**Основы микропроцессорной техники**

**Лекция 1. Философия микропроцессорной техники**

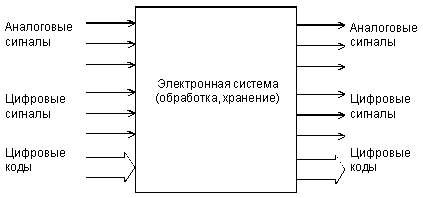
В этой главе рассматриваются базовые концепции, которые лежат в основе любой *микропроцессорной системы* — от простейшего *микроконтроллера* до сложного компьютера. Именно в этом смысле здесь используется термин "философия".

Для начала несколько основных определений.

* **Электронная система** — в данном случае это любой электронный узел, блок, прибор или комплекс, производящий обработку информации.
* **Задача** — это набор функций, выполнение которых требуется от электронной системы.
* **Быстродействие** — это показатель скорости выполнения электронной системой ее функций.
* **Гибкость** — это способность системы подстраиваться под различные задачи.
* **Избыточность** — это показатель степени соответствия возможностей системы решаемой данной системой задаче.
* **Интерфейс** — соглашение об обмене информацией, правила обмена информацией, подразумевающие электрическую, логическую и конструктивную совместимость устройств, участвующих в обмене. Другое название — **сопряжение**.

***Микропроцессорная система*** может рассматриваться как частный случай электронной системы, предназначенной для обработки входных сигналов и выдачи выходных сигналов (рис. 1.1). В качестве входных и выходных сигналов при этом могут использоваться аналоговые сигналы, одиночные цифровые сигналы, цифровые коды, последовательности цифровых кодов. Внутри системы может производиться хранение, накопление сигналов (или информации), но суть от этого не меняется. Если система цифровая (а *микропроцессорные системы* относятся к разряду цифровых), то входные аналоговые сигналы преобразуются в последовательности кодов выборок с помощью *АЦП*, а выходные аналоговые сигналы формируются из последовательности кодов выборок с помощью *ЦАП*. Обработка и хранение информации производятся в цифровом виде.

Характерная особенность традиционной цифровой системы состоит в том, что алгоритмы обработки и хранения информации в ней *жестко связаны* со схемотехникой системы. То есть изменение этих алгоритмов возможно только путем изменения структуры системы, замены электронных узлов, входящих в систему, и/или связей между ними. Например, если нам нужна дополнительная операция суммирования, то необходимо добавить в структуру системы лишний *сумматор*. Или если нужна дополнительная *функция* хранения кода в течение одного такта, то мы должны добавить в структуру еще один *регистр*. Естественно, это практически невозможно сделать в процессе эксплуатации, обязательно нужен новый производственный цикл проектирования, изготовления, отладки всей системы. Именно поэтому традиционная цифровая система часто называется системой на "жесткой логике".



**Рис. 1.1.**Электронная система.

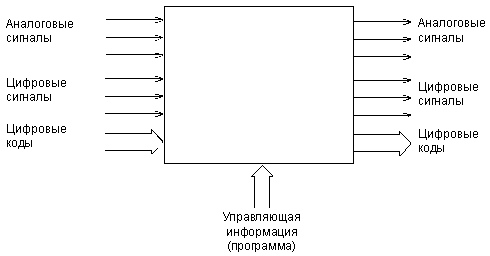
Любая система на "жесткой логике" обязательно представляет собой специализированную систему, настроенную исключительно на одну задачу или (реже) на несколько близких, заранее известных задач. Это имеет свои бесспорные преимущества.

Во-первых, специализированная система (в отличие от универсальной) никогда не имеет аппаратурной избыточности, то есть каждый ее элемент обязательно работает в полную силу (конечно, если эта система грамотно спроектирована).

Во-вторых, именно специализированная система может обеспечить максимально высокое *быстродействие*, так как скорость выполнения алгоритмов обработки информации определяется в ней только быстродействием отдельных логических элементов и выбранной схемой путей прохождения информации. А именно логические элементы всегда обладают максимальным на данный момент быстродействием.

Но в то же время большим недостатком цифровой системы на "жесткой логике" является то, что для каждой новой задачи ее надо проектировать и изготавливать заново. Это процесс длительный, дорогостоящий, требующий высокой квалификации исполнителей. А если решаемая задача вдруг изменяется, то вся аппаратура должна быть полностью заменена. В нашем быстро меняющемся мире это довольно расточительно.

*Путь* преодоления этого недостатка довольно очевиден: надо построить такую систему, которая могла бы легко адаптироваться под любую задачу, перестраиваться с одного алгоритма работы на другой без изменения аппаратуры. И задавать тот или иной *алгоритм* мы тогда могли бы путем ввода в систему некой дополнительной управляющей информации, **программы** работы системы (рис. 1.2). Тогда система станет универсальной, или **программируемой**, не жесткой, а гибкой. Именно это и обеспечивает *микропроцессорная система*.



**Рис. 1.2.**Программируемая (она же универсальная) электронная система.

Но любая *универсальность* обязательно приводит к избыточности. Ведь решение максимально трудной задачи требует гораздо больше средств, чем решение максимально простой задачи. Поэтому сложность универсальной системы должна быть такой, чтобы обеспечивать решение самой трудной задачи, а при решении простой задачи система будет работать далеко не в полную силу, будет использовать не все свои ресурсы. И чем проще решаемая задача, тем больше *избыточность*, и тем менее оправданной становится *универсальность*. *Избыточность* ведет к увеличению стоимости системы, снижению ее надежности, увеличению потребляемой мощности и т.д.

Кроме того, *универсальность*, как правило, приводит к существенному снижению быстродействия. Оптимизировать универсальную систему так, чтобы каждая новая задача решалась максимально быстро, попросту невозможно. *Общее правило* таково: чем больше *универсальность*, гибкость, тем меньше *быстродействие*. Более того, для универсальных систем не существует таких задач (пусть даже и самых простых), которые бы они решали с максимально возможным быстродействием. За все приходится платить.

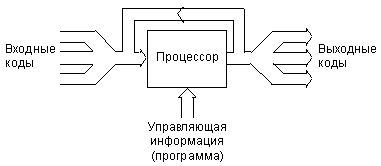
Таким образом, можно сделать следующий *вывод*. Системы на "жесткой логике" хороши там, где решаемая задача не меняется длительное время, где требуется самое высокое *быстродействие*, где алгоритмы обработки информации предельно просты. А универсальные, программируемые системы хороши там, где часто меняются решаемые задачи, где высокое *быстродействие* не слишком важно, где алгоритмы обработки информации сложные. То есть любая система хороша на своем месте.

Однако за последние десятилетия *быстродействие* универсальных ( *микропроцессорных* ) систем сильно выросло (на несколько порядков). К тому же большой объем выпуска микросхем для этих систем привел к резкому снижению их стоимости. В результате область применения систем на "жесткой логике" резко сузилась. Более того, высокими темпами развиваются сейчас программируемые системы, предназначенные для решения одной задачи или нескольких близких задач. Они удачно совмещают в себе как достоинства систем на "жесткой логике", так и программируемых систем, обеспечивая сочетание достаточно высокого быстродействия и необходимой гибкости. Так что *вытеснение* "жесткой логики" продолжается.

**1.1. Что такое микропроцессор?**

Ядром любой *микропроцессорной системы* является *микропроцессор* или просто *процессор* (от английского *processor*). Перевести на русский язык это *слово* правильнее всего как "обработчик", так как именно ***микропроцессор*** — это тот узел, блок, который производит всю обработку информации внутри*микропроцессорной системы* . Остальные узлы выполняют всего лишь *вспомогательные функции*: хранение информации (в том числе и управляющей информации, то есть программы), связи с внешними устройствами, связи с пользователем и т.д. *Процессор* заменяет практически всю "жесткую логику", которая понадобилась бы в случае традиционной цифровой системы. Он выполняет *арифметические функции* (*сложение*, *умножение* и т.д.), логические функции (сдвиг, сравнение, маскирование кодов и т.д.), временное хранение кодов (во внутренних регистрах), пересылку кодов между узлами *микропроцессорной системы* и многое другое. Количество таких элементарных операций, выполняемых процессором, может достигать нескольких сотен. *Процессор* можно сравнить с мозгом системы.

Но при этом надо учитывать, что все свои *операции* *процессор* выполняет **последовательно**, то есть одну за другой, *по* очереди. Конечно, существуют процессоры с параллельным выполнением некоторых операций, встречаются также *микропроцессорные системы*, в которых несколько процессоров работают над одной задачей параллельно, но это редкие исключения. С одной стороны, последовательное выполнение операций — несомненное достоинство, так как позволяет с помощью всего одного процессора выполнять любые, самые сложные алгоритмы обработки информации. Но, с другой стороны, последовательное выполнение операций приводит к тому, что *время выполнения* алгоритма зависит от его сложности. Простые алгоритмы выполняются быстрее сложных. То есть *микропроцессорная система* способна сделать все, но работает она не слишком быстро, ведь все информационные потоки приходится пропускать через один-единственный узел — *микропроцессор* (рис. 1.3). В традиционной цифровой системе можно легко организовать *параллельную обработку* всех потоков информации, правда, ценой усложнения схемы.

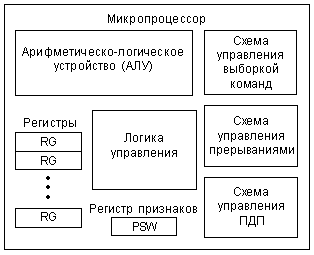


**Рис. 1.3.**Информационные потоки в микропроцессорной системе.

Итак, *микропроцессор* способен выполнять множество операций. Но откуда он узнает, какую операцию ему надо выполнять в данный момент? Именно это определяется управляющей информацией, программой. *Программа* представляет собой набор **команд (инструкций)**, то есть цифровых кодов, расшифровав которые,*процессор* узнает, что ему надо делать. *Программа* от начала и до конца составляется человеком, программистом, а *процессор* выступает в роли послушного исполнителя этой программы, никакой инициативы он не проявляет (если, конечно, исправен). Поэтому сравнение процессора с мозгом не слишком корректно. Он всего лишь *исполнитель* того алгоритма, который заранее составил для него человек. Любое отклонение от этого алгоритма может быть вызвано только неисправностью процессора или каких-нибудь других узлов *микропроцессорной системы*.

Все команды, выполняемые процессором, образуют **систему команд** процессора. Структура и объем системы команд процессора определяют его *быстродействие*, гибкость, *удобство использования*. Всего команд у процессора может быть от нескольких десятков до нескольких сотен. *Система команд* может быть рассчитана на узкий круг решаемых задач (у специализированных процессоров) или на максимально широкий круг задач (у универсальных процессоров). Коды команд могут иметь различное количество разрядов (занимать от одного до нескольких *байт*). Каждая *команда* имеет свое *время выполнения*, поэтому *время выполнения* всей программы зависит не только от количества команд в программе, но и от того, какие именно команды используются.

Для выполнения команд в структуру процессора входят внутренние регистры, *арифметико-логическое устройство* (*АЛУ*, *ALU* — *Arithmetic* *Logic Unit*) , мультиплексоры, буферы, регистры и другие узлы. Работа всех узлов синхронизируется общим внешним тактовым сигналом процессора. То есть *процессор* представляет собой довольно сложное цифровое устройство (рис. 1.4).



**Рис. 1.4.**Пример структуры простейшего процессора.

Впрочем, для разработчика *микропроцессорных систем* *информация* о тонкостях внутренней структуры процессора не слишком важна. Разработчик должен рассматривать *процессор* как "*черный ящик*", который в ответ на входные и *управляющие коды* производит ту или иную операцию и выдает выходные сигналы. Разработчику необходимо знать систему команд, режимы работы процессора, а также правила взаимодействия процессора с внешним миром или, как их еще называют, протоколы обмена информацией. О внутренней структуре процессора надо знать только то, что необходимо для выбора той или иной команды, того или иного режима работы.

**1.2. Шинная структура связей**

Для достижения максимальной универсальности и упрощения *протоколов обмена* информацией в *микропроцессорных системах* применяется так называемая шинная структура связей между отдельными устройствами, входящими в систему. Суть шинной структуры связей сводится к следующему.

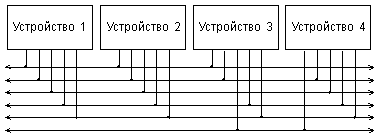


**Рис. 1.5.**Классическая структура связей (каждый с каждым).

При классической структуре связей (рис. 1.5) все сигналы и коды между устройствами передаются *по* отдельным линиям связи. Каждое устройство, входящее в систему, передает свои сигналы и коды независимо от других устройств. При этом в системе получается очень много линий связи и разных *протоколов обмена*информацией.

При шинной структуре связей (рис. 1.6) все сигналы между устройствами передаются *по* одним и тем же линиям связи, но в разное время (это называется мультиплексированной передачей). Причем передача *по* всем линиям связи может осуществляться в обоих направлениях (так называемая двунаправленная передача). В результате количество линий связи существенно сокращается, а правила обмена (протоколы) упрощаются. Группа линий связи, по которым передаются сигналы или коды как раз и называется ***шиной*** (англ. bus).

Понятно, что при шинной структуре связей легко осуществляется пересылка всех информационных потоков в нужном направлении, например, их можно пропустить через один *процессор*, что очень важно для *микропроцессорной системы*. Однако при шинной структуре связей вся *информация* передается *по* линиям связи последовательно во времени, *по* очереди, что снижает *быстродействие* системы *по* сравнению с классической структурой связей.



**Рис. 1.6.**Шинная структура связей.

Большое достоинство шинной структуры связей состоит в том, что все устройства, подключенные к *шине*, должны принимать и передавать информацию *по* одним и тем же правилам (протоколам обмена информацией *по* *шине* ). Соответственно, все узлы, отвечающие за обмен с *шиной* в этих устройствах, должны быть единообразны, унифицированы.

Существенный недостаток шинной структуры связан с тем, что все устройства подключаются к каждой линии связи параллельно. Поэтому любая неисправность любого устройства может вывести из строя всю систему, если она портит линию связи. *По* этой же причине *отладка* системы с шинной структурой связей довольно сложна и обычно требует специального оборудования.

В системах с шинной структурой связей применяют все три существующие разновидности выходных каскадов цифровых микросхем:

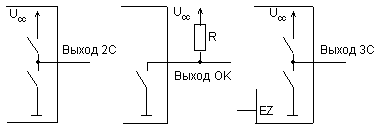
* стандартный выход или выход с двумя состояниями (обозначается 2С, 2S, реже ТТЛ, *TTL*);
* выход с открытым коллектором (обозначается ОК, *OC*);
* выход с тремя состояниями или (что то же самое) с возможностью отключения (обозначается 3С, 3S).

Упрощенно эти три типа выходных каскадов могут быть представлены в виде схем на рис. 1.7.

У выхода 2С два ключа замыкаются *по* очереди, что соответствует уровням логической единицы (верхний *ключ* замкнут) и логического нуля (нижний *ключ* замкнут).

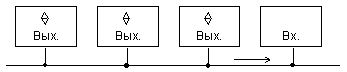
У выхода ОК замкнутый *ключ* формирует уровень логического нуля, разомкнутый — логической единицы.

У выхода 3С ключи могут замыкаться *по* очереди (как в случае 2С), а могут размыкаться одновременно, образуя третье, высокоимпедансное состояние. Переход в третье состояние (Z-состояние) управляется сигналом на специальном входе EZ.

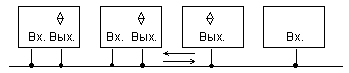


**Рис. 1.7.**Три типа выходов цифровых микросхем.

Выходные каскады типов 3С и ОК позволяют объединять несколько выходов микросхем для получения мультиплексированных (рис. 1.8) или двунаправленных (рис. 1.9) линий.



**Рис. 1.8.**Мультиплексированная линия.

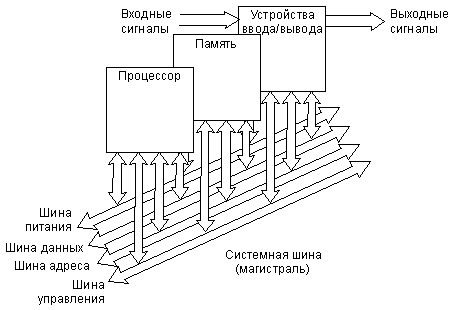


**Рис. 1.9.**Двунаправленная линия.

При этом в случае выходов 3С необходимо обеспечить, чтобы на линии всегда работал только один *активный* *выход*, а все остальные выходы находились бы в это время в третьем состоянии, иначе возможны конфликты. Объединенные выходы ОК могут работать все одновременно, без всяких конфликтов.

Типичная структура *микропроцессорной системы* приведена на рис. 1.10. Она включает в себя три основных типа устройств:

* процессор;
* *память*, включающую оперативную ***память*** ( ОЗУ, RAM — *Random Access* Memory ) и постоянную *память* ( ПЗУ, ROM —Read Only Memory ), которая служит для хранения данных и программ;
* ***устройства ввода/вывода*** ( УВВ, I/O — Input/*Output Devices* ), служащие для связи *микропроцессорной системы* с внешними устройствами, для приема (ввода, чтения, Read) входных сигналов и выдачи (вывода, записи, Write) выходных сигналов.



**Рис. 1.10.**Структура микропроцессорной системы.

Все устройства *микропроцессорной системы* объединяются общей системной ***шиной*** (она же называется еще **системной магистралью** или **каналом** ). Системная *магистраль* включает в себя четыре основные *шины* нижнего уровня:

* *шина* адреса (*Address Bus*);
* *шина* данных (*Data Bus*);
* *шина* управления (*Control Bus*);
* *шина* питания (*Power Bus*).

*Шина* адреса служит для определения адреса (номера) устройства, с которым *процессор* обменивается информацией в данный момент. Каждому устройству (кроме процессора), каждой ячейке *памяти* в *микропроцессорной системе* присваивается собственный *адрес*. Когда код какого-то адреса выставляется процессором на *шине* адреса, устройство, которому этот *адрес* приписан, понимает, что ему предстоит *обмен информацией*. *Шина* адреса может быть однонаправленной или двунаправленной.

*Шина* данных — это основная *шина*, которая используется для передачи информационных кодов между всеми устройствами *микропроцессорной системы*. Обычно в пересылке информации участвует *процессор*, который передает код данных в какое-то устройство или в ячейку *памяти* или же принимает код данных из какого-то устройства или из ячейки *памяти*. Но возможна также и передача информации между устройствами без участия процессора. *Шина* данных всегда двунаправленная.

*Шина* управления в отличие от *шины* адреса и *шины* данных состоит из отдельных управляющих сигналов. Каждый из этих сигналов во время обмена информацией имеет свою функцию. Некоторые сигналы служат для стробирования передаваемых или принимаемых данных (то есть определяют моменты времени, когда информационный код выставлен на *шину* данных). Другие *управляющие* сигналы могут использоваться для подтверждения приема данных, для сброса всех устройств в исходное состояние, для тактирования всех устройств и т.д. Линии *шины* управления могут быть однонаправленными или двунаправленными.

Наконец, *шина* питания предназначена не для пересылки информационных сигналов, а для питания системы. Она состоит из линий питания и общего провода. В *микропроцессорной системе* может быть один источник питания (чаще +5 В) или несколько источников питания (обычно еще –5 В, +12 В и –12 В). Каждому напряжению питания соответствует своя *линия связи*. Все устройства подключены к этим линиям параллельно.

Если в *микропроцессорную систему* надо ввести *входной* код (или *входной* сигнал), то *процессор* *по* *шине* адреса обращается к нужному *устройству ввода/вывода* и принимает *по* *шине* данных *входную информацию*. Если из *микропроцессорной системы* надо вывести выходной код (или выходной сигнал), то *процессор*обращается *по* *шине* адреса к нужному *устройству ввода/вывода* и передает ему *по* *шине* данных *выходную информацию*.

Если *информация* должна пройти сложную многоступенчатую обработку, то *процессор* может хранить промежуточные результаты в системной оперативной *памяти*. Для обращения к любой ячейке *памяти* *процессор* выставляет ее *адрес* на *шину* адреса и передает в нее информационный код *по* *шине* данных или же принимает из нее информационный код *по* *шине* данных. В *памяти* (оперативной и постоянной) находятся также и *управляющие коды* (команды выполняемой процессором программы), которые *процессор* также читает *по* *шине* данных с адресацией *по* *шине* адреса. Постоянная *память* используется в основном для хранения программы начального пуска *микропроцессорной системы*, которая выполняется каждый раз после включения питания. *Информация* в нее заносится изготовителем раз и навсегда.

Таким образом, в *микропроцессорной системе* все информационные коды и коды команд передаются *по* *шинам* последовательно, *по* очереди. Это определяет сравнительно невысокое *быстродействие* *микропроцессорной системы*. Оно ограничено обычно даже не быстродействием процессора (которое тоже очень важно) и не скоростью обмена *по* системной *шине* (магистрали), а именно последовательным характером передачи информации *по* системной *шине* (магистрали).

Важно учитывать, что *устройства ввода/вывода* чаще всего представляют собой устройства на "жесткой логике". На них может быть возложена часть функций, выполняемых *микропроцессорной системой*. Поэтому у разработчика всегда имеется возможность перераспределять функции системы между аппаратной и программной реализациями оптимальным образом. Аппаратная реализация ускоряет выполнение функции, но имеет недостаточную гибкость. *Программная реализация* значительно медленнее, но обеспечивает высокую гибкость. Аппаратная реализация функций увеличивает *стоимость* системы и ее *энергопотребление*, программная — не увеличивает. Чаще всего применяется комбинирование аппаратных и программных функций.

Иногда *устройства ввода/вывода* имеют в своем составе *процессор*, то есть представляют собой небольшую специализированную *микропроцессорную систему*. Это позволяет переложить часть программных функций на *устройства ввода/вывода*, разгрузив центральный *процессор* системы.

**1.3. Режимы работы микропроцессорной системы**

Как уже отмечалось, *микропроцессорная система* обеспечивает большую гибкость работы, она способна настраиваться на любую задачу. Гибкость эта обусловлена прежде всего тем, что функции, выполняемые системой, определяются программой (программным обеспечением, *software*), которую выполняет *процессор*. Аппаратура (*аппаратное обеспечение*, *hardware*) остается неизменной при любой задаче. Записывая в *память* системы программу, можно заставить *микропроцессорную систему* выполнять любую задачу, поддерживаемую данной аппаратурой. К тому же шинная *организация связей* *микропроцессорной системы* позволяет довольно легко заменять аппаратные модули, например, заменять *память* на новую большего объема или более высокого быстродействия, добавлять или модернизировать *устройства ввода/вывода*, наконец, заменять *процессор* на более мощный. Это также позволяет увеличить гибкость системы, продлить ее жизнь при любом изменении требований к ней.

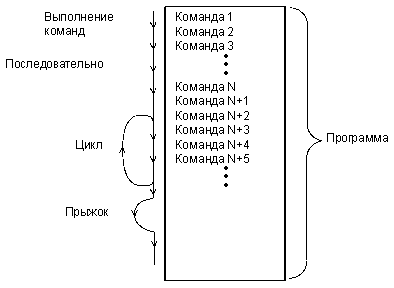
Но гибкость *микропроцессорной системы* определяется не только этим. Настраиваться на задачу помогает еще и выбор режима работы системы, то есть режима обмена информацией *по* системной магистрали ( *шине* ).

**Практически любая развитая*****микропроцессорная система* (в том числе и*****компьютер*) поддерживает три основных режима обмена*****по* магистрали:**

* **программный обмен информацией;**
* **обмен с использованием прерываний (*****Interrupts*);**
* **обмен с использованием прямого доступа к*****памяти* (*****ПДП*,*****DMA* —*****Direct Memory Access*).**

**Программный обмен информацией** является основным в любой *микропроцессорной системе*. Он предусмотрен всегда, без него невозможны другие режимы обмена. В этом режиме *процессор* является единоличным хозяином (или задатчиком, Master) системной магистрали. Все *операции* (циклы) обмена информацией в данном случае инициируются только процессором, все они выполняются строго в порядке, предписанном исполняемой программой.

*Процессор* читает (выбирает) из *памяти* коды команд и исполняет их, читая данные из *памяти* или из *устройства ввода/вывода*, обрабатывая их, записывая данные в *память* или передавая их в *устройство ввода/вывода*. *Путь* процессора *по* программе может быть линейным, циклическим, может содержать переходы (прыжки), но он всегда непрерывен и полностью находится под контролем процессора. Ни на какие внешние события, не связанные с программой, *процессор* не реагирует (рис. 1.11). Все сигналы на магистрали в данном случае контролируются процессором.



**Рис. 1.11.**Программный обмен информацией.

**Обмен по прерываниям** используется тогда, когда необходима *реакция* *микропроцессорной системы* на какое-то внешнее событие, на приход внешнего сигнала. В случае компьютера внешним событием может быть, например, нажатие на клавишу клавиатуры или приход *по* локальной сети пакета данных. *Компьютер* должен реагировать на это, соответственно, выводом символа на экран или же чтением и обработкой принятого *по* сети пакета.

В общем случае организовать реакцию на внешнее событие можно тремя различными путями:

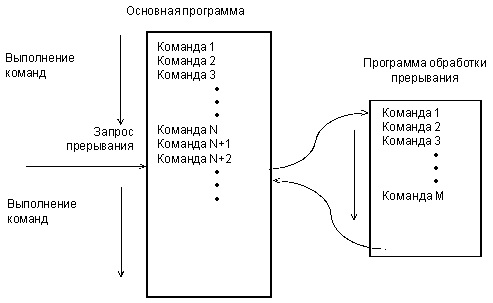
* с помощью постоянного программного контроля факта наступления события (так называемый метод опроса флага или *polling*);
* с помощью прерывания, то есть насильственного перевода процессора с выполнения текущей программы на выполнение экстренно необходимой программы;
* с помощью прямого доступа к *памяти*, то есть без участия процессора при его отключении от системной магистрали.

Проиллюстрировать эти три способа можно следующим простым примером. Допустим, вы готовите себе завтрак, поставив на плиту кипятиться молоко. Естественно, на закипание молока надо реагировать, причем срочно. Как это организовать? Первый *путь* — постоянно следить за молоком, но тогда вы ничего другого не сможете делать. Правильнее будет регулярно поглядывать на молоко, делая одновременно что-то другое. Это программный режим с опросом флага. Второй *путь* — установить на кастрюлю с молоком датчик, который подаст звуковой сигнал при закипании молока, и спокойно заниматься другими делами. Услышав сигнал, вы выключите молоко. Правда, возможно, вам придется сначала закончить то, что вы начали делать, так что ваша *реакция* будет медленнее, чем в первом случае. Наконец, третий *путь* состоит в том, чтобы соединить датчик на кастрюле с управлением плитой так, чтобы при закипании молока горелка была выключена без вашего участия (правда, *аналогия* с *ПДП* здесь не очень точная, так как в данном случае на момент выполнения действия вас не отвлекают от работы).

Первый случай с опросом флага реализуется в *микропроцессорной системе* постоянным чтением информации процессором из *устройства ввода/вывода*, связанного с тем внешним устройством, на поведение которого необходимо срочно реагировать.

Во втором случае в режиме прерывания *процессор*, получив *запрос прерывания* от внешнего устройства (часто называемый *IRQ* — *Interrupt ReQuest*), заканчивает выполнение текущей команды и переходит к программе обработки прерывания. Закончив выполнение программы обработки прерывания, он возвращается к прерванной программе с той точки, где его прервали (рис. 1.12).

Здесь важно то, что вся работа, как и в случае программного режима, осуществляется самим процессором, внешнее событие просто временно отвлекает его. *Реакция* на внешнее событие *по* прерыванию в общем случае медленнее, чем при программном режиме. Как и в случае программного обмена, здесь все сигналы на магистрали выставляются процессором, то есть он полностью контролирует *магистраль*. Для обслуживания прерываний в систему иногда вводится специальный *модуль* *контроллера прерываний*, но он в обмене информацией не участвует. Его задача состоит в том, чтобы упростить работу процессора с внешними*запросами прерываний*. Этот *контроллер* обычно программно управляется процессором *по* системной магистрали.



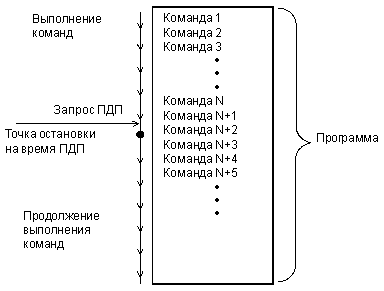
**Рис. 1.12.**Обслуживание прерывания.

Естественно, никакого ускорения работы системы *прерывание* не дает. Его применение позволяет только отказаться от постоянного опроса флага внешнего события и временно, до наступления внешнего события, занять *процессор* выполнением каких-то других задач.

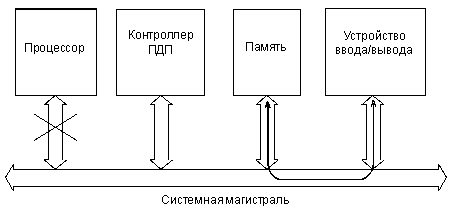
**Прямой доступ к памяти (ПДП, DMA)** — это режим, принципиально отличающийся от двух ранее рассмотренных режимов тем, что обмен *по* системной *шине* идет без участия процессора. *Внешнее устройство*, требующее обслуживания, сигнализирует процессору, что режим *ПДП* необходим, в ответ на это *процессор*заканчивает выполнение текущей команды и отключается от всех *шин*, сигнализируя запросившему устройству, что обмен в режиме *ПДП* можно начинать.

Операция *ПДП* сводится к пересылке информации из *устройства ввода/вывода* в *память* или же из *памяти* в *устройство ввода/вывода*. Когда пересылка информации будет закончена, *процессор* вновь возвращается к прерванной программе, продолжая ее с той точки, где его прервали (рис. 1.13). Это похоже на режим обслуживания прерываний, но в данном случае *процессор* не участвует в обмене. Как и в случае прерываний, *реакция* на внешнее событие при *ПДП* существенно медленнее, чем при программном режиме.

Понятно, что в этом случае требуется введение в систему дополнительного устройства (контроллера *ПДП*), которое будет осуществлять полноценный обмен *по* системной магистрали без всякого участия процессора. Причем *процессор* предварительно должен сообщить этому контроллеру *ПДП*, откуда ему следует брать информацию и/или куда ее следует помещать. *Контроллер* *ПДП* может считаться специализированным процессором, который отличается тем, что сам не участвует в обмене, не принимает в себя информацию и не выдает ее (рис. 1.14).



**Рис. 1.13.**Обслуживание ПДП.



**Рис. 1.14.**Информационные потоки в режиме ПДП.

В принципе *контроллер* *ПДП* может входить в состав *устройства ввода/вывода*, которому необходим режим *ПДП* или даже в состав нескольких *устройств ввода/вывода*. Теоретически обмен с помощью прямого доступа к *памяти* может обеспечить более высокую *скорость передачи* информации, чем программный обмен, так как *процессор* передает данные медленнее, чем специализированный *контроллер* *ПДП*. Однако на практике это преимущество реализуется далеко не всегда. Скорость обмена в режиме *ПДП* обычно ограничена возможностями магистрали. К тому же необходимость программного задания режимов контроллера *ПДП*может свести на нет выигрыш от более высокой скорости пересылки данных в режиме *ПДП*. Поэтому режим *ПДП* применяется редко.

Если в системе уже имеется самостоятельный *контроллер* *ПДП*, то это может в ряде случаев существенно упростить аппаратуру *устройств ввода/вывода*, работающих в режиме *ПДП*. В этом, пожалуй, состоит единственное бесспорное преимущество режима *ПДП*.

**1.4. Архитектура микропроцессорных систем**

До сих пор мы рассматривали только один тип *архитектуры*   *микропроцессорных систем* — *архитектуру* с общей, единой *шиной* для данных и команд (одношинную, или **принстонскую**, **фон-неймановскую**   *архитектуру* ). Соответственно, в составе системы в этом случае присутствует одна общая *память*, как для данных, так и для команд (рис. 1.15).



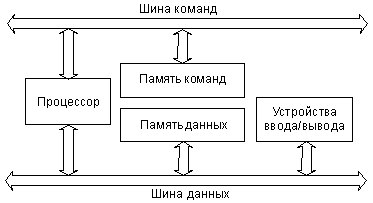
**Рис. 1.15.**Архитектура с общей шиной данных и команд.

Но существует также и альтернативный тип *архитектуры* *микропроцессорной системы* — это *архитектура* с раздельными *шинами* данных и команд (двухшинная, или гарвардская, *архитектура* ). Эта *архитектура* предполагает наличие в системе отдельной *памяти* для данных и отдельной *памяти* для команд (рис. 1.16). Обмен процессора с каждым из двух типов *памяти* происходит *по* своей *шине*.

*Архитектура* с общей *шиной* распространена гораздо больше, она применяется, например, в персональных компьютерах и в сложных *микрокомпьютерах*. *Архитектура* с раздельными *шинами* применяется в основном в однокристальных *микроконтроллерах*.

Рассмотрим некоторые достоинства и недостатки обоих архитектурных решений.

*Архитектура* с общей *шиной* (принстонская, фон-неймановская) проще, она не требует от процессора одновременного обслуживания двух *шин*, контроля обмена *по* двум *шинам* сразу. Наличие единой *памяти* данных и команд позволяет гибко распределять ее объем между кодами данных и команд. Например, в некоторых случаях нужна большая и сложная *программа*, а данных в *памяти* надо хранить не слишком много. В других случаях, наоборот, *программа* требуется простая, но необходимы большие объемы хранимых данных. Перераспределение *памяти* не вызывает никаких проблем, главное — чтобы *программа* и данные вместе помещались в *памяти* системы. Как правило, в системах с такой *архитектурой**память* бывает довольно большого объема (до десятков и сотен *мегабайт*). Это позволяет решать самые сложные задачи.



**Рис. 1.16.**Архитектура с раздельными шинами данных и команд.

*Архитектура* с раздельными *шинами* данных и команд сложнее, она заставляет *процессор* работать одновременно с двумя потоками кодов, обслуживать обмен *по* двум *шинам* одновременно. *Программа* может размещаться только в *памяти* команд, данные — только в *памяти* данных. Такая узкая специализация ограничивает круг задач, решаемых системой, так как не дает возможности гибкого перераспределения *памяти*. *Память* данных и *память* команд в этом случае имеют не слишком большой объем, поэтому применение систем с данной *архитектурой* ограничивается обычно не слишком сложными задачами.

**В чем же преимущество*****архитектуры* с двумя*****шинами* (гарвардской)? В первую*****очередь*, в быстродействии.**

Дело в том, что при единственной *шине* команд и данных *процессор* вынужден *по* одной этой *шине* принимать данные (из *памяти* или *устройства ввода/вывода* ) и передавать данные (в *память* или в *устройство ввода/вывода* ), а также читать команды из *памяти*. Естественно, одновременно эти пересылки кодов *по* магистрали происходить не могут, они должны производиться *по* очереди. Современные процессоры способны совместить во времени выполнение команд и проведение циклов обмена *по* системной *шине*. Использование конвейерных технологий и быстрой кэш-памяти позволяет им ускорить процесс взаимодействия со сравнительно медленной системной *памятью*. Повышение тактовой частоты и совершенствование структуры процессоров дают возможность сократить *время выполнения* команд. Но дальнейшее увеличение быстродействия системы возможно только при совмещении пересылки данных и чтения команд, то есть при переходе к *архитектуре* с двумя *шинами*.

В случае двухшинной *архитектуры* обмен *по* обеим *шинам* может быть независимым, параллельным во времени. Соответственно, структуры *шин* (количество разрядов кода адреса и кода данных, порядок и скорость обмена информацией и т.д.) могут быть выбраны оптимально для той задачи, которая решается каждой *шиной*. Поэтому при прочих равных условиях переход на двухшинную *архитектуру* ускоряет работу *микропроцессорной системы*, хотя и требует дополнительных затрат на аппаратуру, усложнения структуры процессора. *Память* данных в этом случае имеет свое распределение адресов, а *память* команд — свое.

Проще всего преимущества двухшинной *архитектуры* реализуются внутри одной микросхемы. В этом случае можно также существенно уменьшить влияние недостатков этой *архитектуры*. Поэтому основное ее применение — в *микроконтроллерах*, от которых не требуется решения слишком сложных задач, но зато необходимо максимальное *быстродействие* при заданной тактовой частоте.

**1.5. Типы микропроцессорных систем**

*Диапазон* применения микропроцессорной техники сейчас очень широк, требования к *микропроцессорным системам* предъявляются самые разные. Поэтому сформировалось несколько типов *микропроцессорных систем*, различающихся мощностью, универсальностью, быстродействием и структурными отличиями. Основные типы следующие:

* микроконтроллеры — наиболее простой тип *микропроцессорных систем*, в которых все или большинство узлов системы выполнены в виде одной микросхемы;
* контроллеры — управляющие *микропроцессорные системы*, выполненные в виде отдельных модулей(на платах);
* микрокомпьютеры (на платах)— более мощные *микропроцессорные системы* с развитыми средствами сопряжения с внешними устройствами.
* компьютеры (в том числе персональные) — самые мощные и наиболее универсальные *микропроцессорные системы*.

Четкую границу между этими типами иногда провести довольно сложно. *Быстродействие* всех типов *микропроцессоров* постоянно растет, и нередки ситуации, когда новый *микроконтроллер* оказывается быстрее, например, устаревшего персонального компьютера. Но кое-какие принципиальные отличия все-таки имеются.

Микроконтроллеры представляют собой универсальные устройства, которые практически всегда используются не сами *по* себе, а в составе более сложных устройств, в том числе и контроллеров. Системная *шина* *микроконтроллера* скрыта от пользователя внутри микросхемы. Возможности подключения внешних устройств к микроконтроллеру ограничены. Устройства на *микроконтроллерах* обычно предназначены для решения одной задачи.

Контроллеры, как правило, создаются для решения какой-то отдельной задачи или группы близких задач. Они обычно не имеют возможностей подключения дополнительных узлов и устройств, например, большой *памяти*, средств ввода/вывода. Их системная *шина* чаще всего недоступна пользователю. Структура контроллера проста и оптимизирована под максимальное *быстродействие*. В большинстве случаев выполняемые программы хранятся в постоянной *памяти* и не меняются. Конструктивно контроллеры выпускаются в одноплатном варианте.

Микрокомпьютеры отличаются от контроллеров более открытой структурой, они допускают подключение к системной *шине* нескольких дополнительных устройств. Производятся микрокомпьютеры в каркасе, корпусе с разъемами системной магистрали, доступными пользователю. Микрокомпьютеры могут иметь средства хранения информации на магнитных носителях (например, магнитные диски) и довольно развитые средства связи с пользователем (видеомонитор, клавиатура). Микрокомпьютеры рассчитаны на широкий круг задач, но в отличие от контроллеров, к каждой новой задаче его надо приспосабливать заново. Выполняемые микрокомпьютером программы можно легко менять.

Наконец, компьютеры и самые распространенные из них — персональные компьютеры — это самые универсальные из *микропроцессорных систем*. Они обязательно предусматривают возможность модернизации, а также широкие возможности подключения новых устройств. Их системная *шина*, конечно, доступна пользователю. Кроме того, внешние устройства могут подключаться к компьютеру через несколько встроенных портов связи (количество портов доходит иногда до 10). *Компьютер* всегда имеет сильно развитые средства связи с пользователем, средства длительного хранения информации большого объема, средства связи с другими компьютерами *по* информационным сетям. Области применения компьютеров могут быть самыми разными: математические расчеты, обслуживание доступа к базам данных, управление работой сложных электронных систем, компьютерные игры, подготовка документов и т.д.

Любую задачу в принципе можно выполнить с помощью каждого из перечисленных типов *микропроцессорных систем*. Но при выборе типа надо *по* возможности избегать избыточности и предусматривать необходимую для данной задачи гибкость системы.

В настоящее время при разработке новых *микропроцессорных систем* чаще всего выбирают *путь* использования микроконтроллеров (примерно в 80% случаев). При этом микроконтроллеры применяются или самостоятельно, с минимальной дополнительной аппаратурой, или в составе более сложных контроллеров с развитыми средствами ввода/вывода.

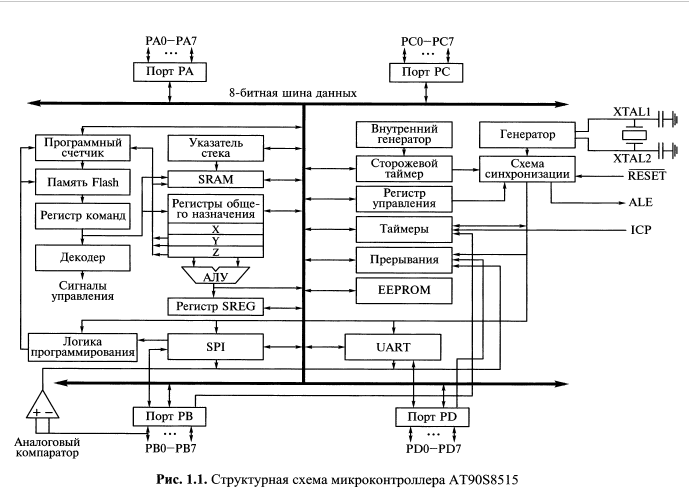
Классические *микропроцессорные системы* на базе микросхем процессоров и микропроцессорных комплектов выпускаются сейчас довольно редко, в первую *очередь*, из-за сложности процесса разработки и отладки этих систем. Данный тип *микропроцессорных систем* выбирают в основном тогда, когда микроконтроллеры не могут обеспечить требуемых характеристик.

Наконец, заметное *место* занимают сейчас *микропроцессорные системы* на основе персонального компьютера. Разработчику в этом случае нужно только оснастить *персональный компьютер* дополнительными устройствами сопряжения, а *ядро* *микропроцессорной системы* уже готово. *Персональный компьютер* имеет развитые средства программирования, что существенно упрощает задачу разработчика. К тому же он может обеспечить самые сложные алгоритмы обработки информации. Основные недостатки персонального компьютера — большие размеры корпуса и аппаратурная *избыточность* для простых задач. Недостатком является и неприспособленность большинства персональных компьютеров к работе в сложных условиях (запыленность, высокая влажность, вибрации, высокие температуры и т.д.). Однако выпускаются и специальные персональные компьютеры, приспособленные к различным условиям эксплуатации.

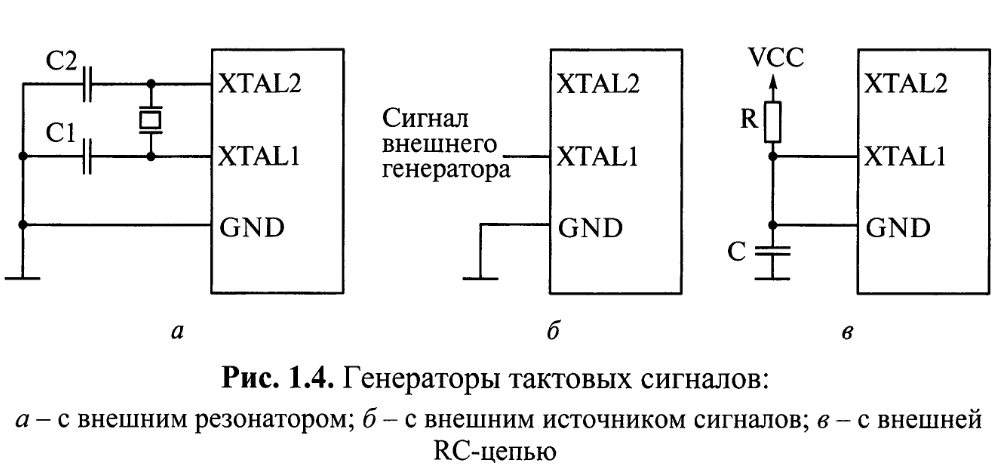
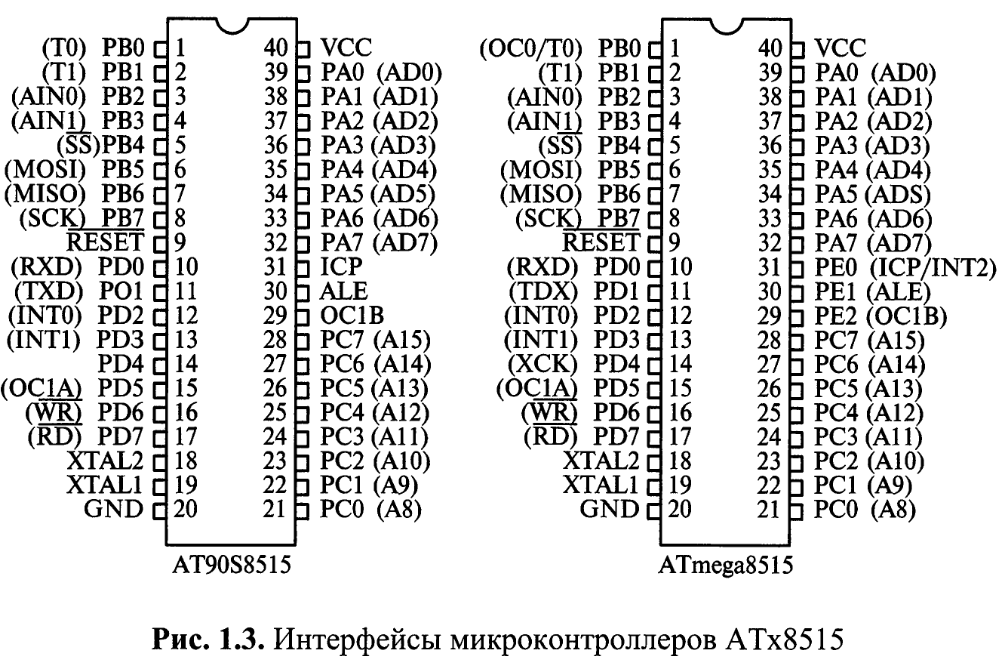
**Далее (перед шинами):**

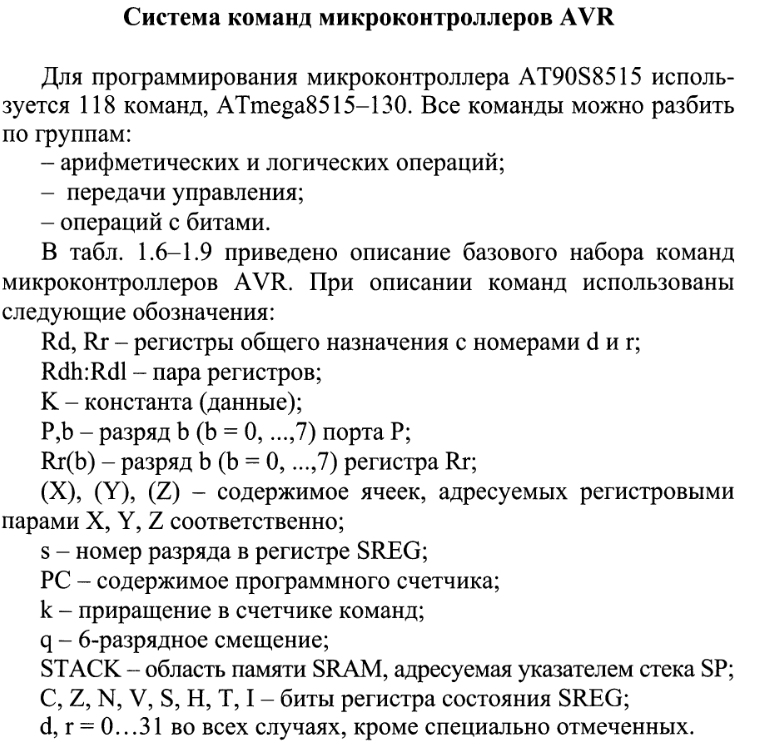
**Лекция 1(дополнение). Архитектура МК на основе ядра Intel MCS51**

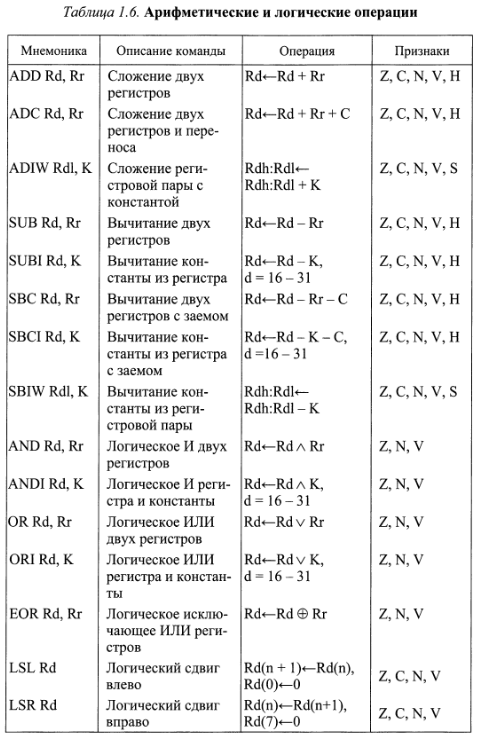
***Система команд RISC (на примере микроконтроллера AVR (AT90S8515, рис.1.1, с.10, см. в Хартов В.Я., Микроконтроллеры AVR, 2012)***

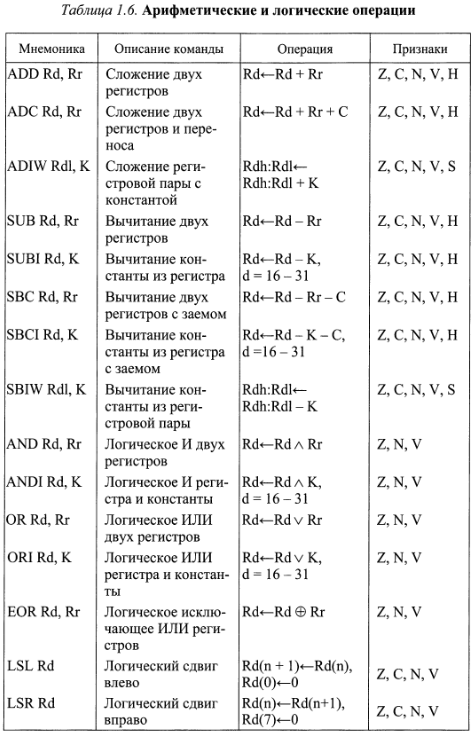
****

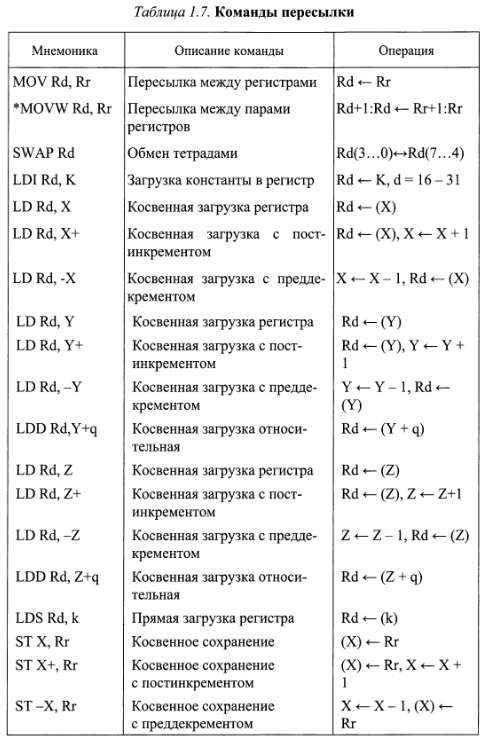
****

****

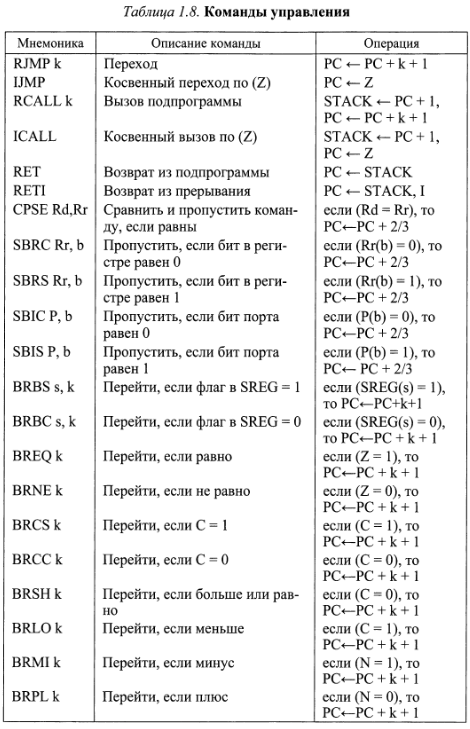
****

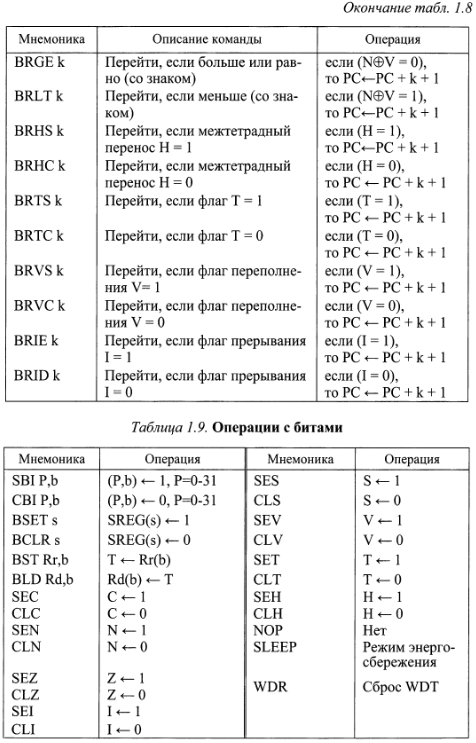
****

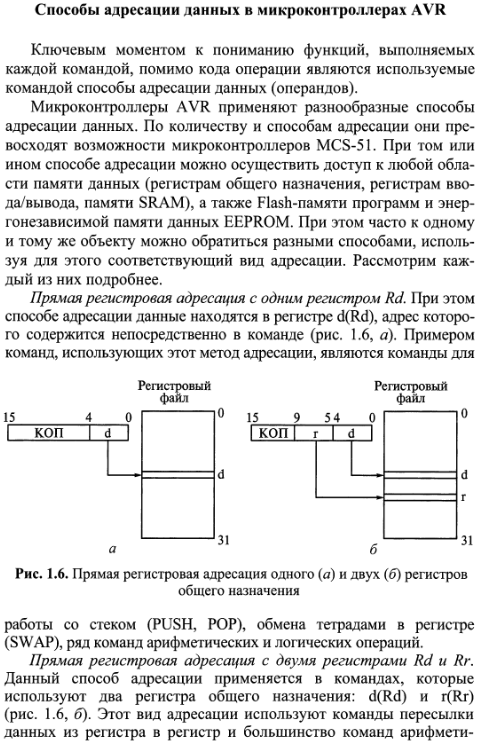
****

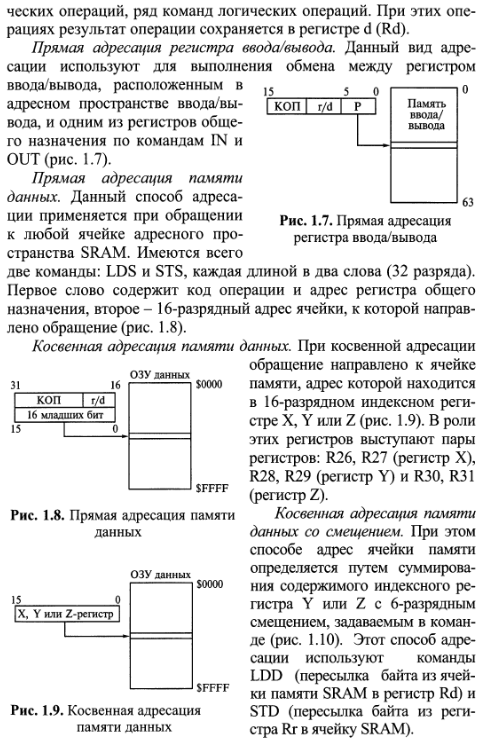
****

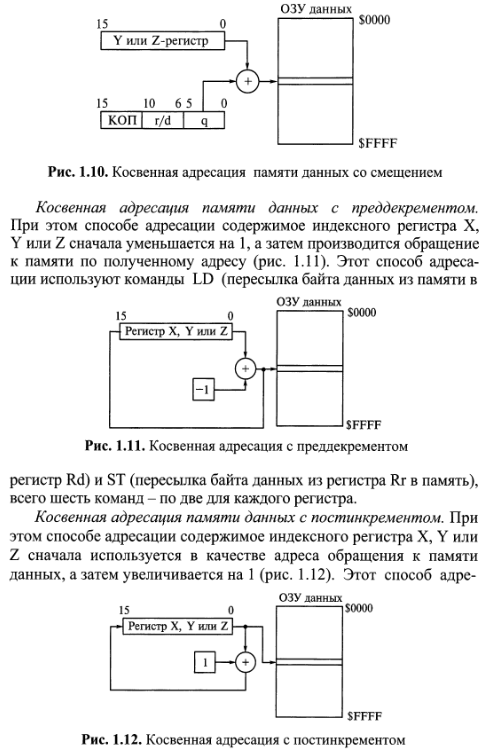
****

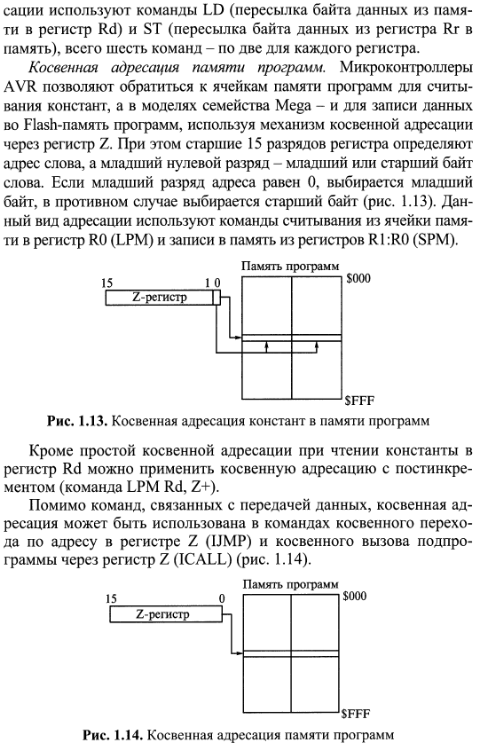
****

****

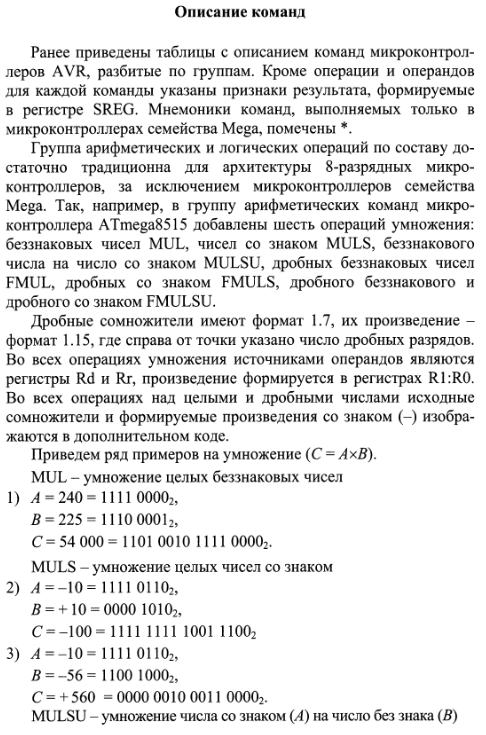
****

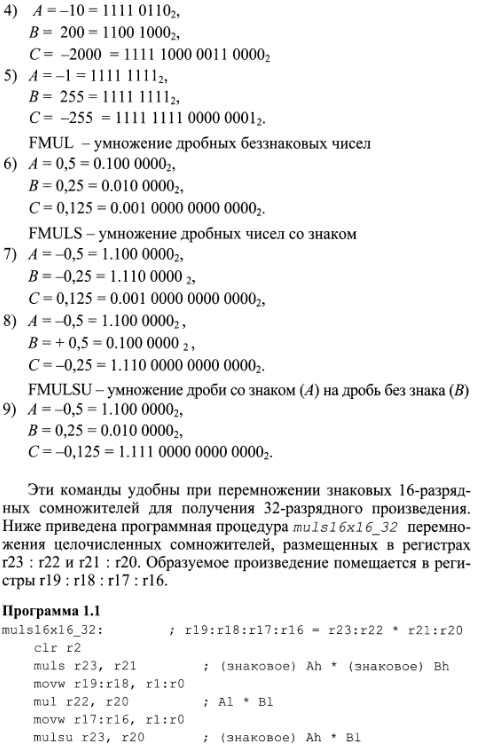
****

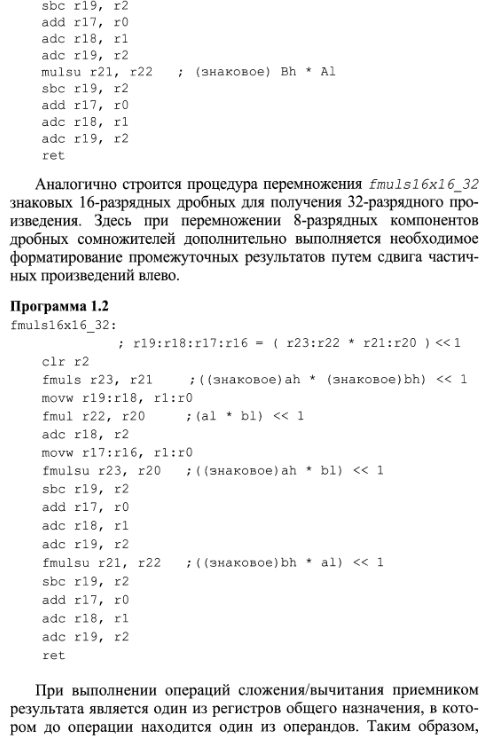
****

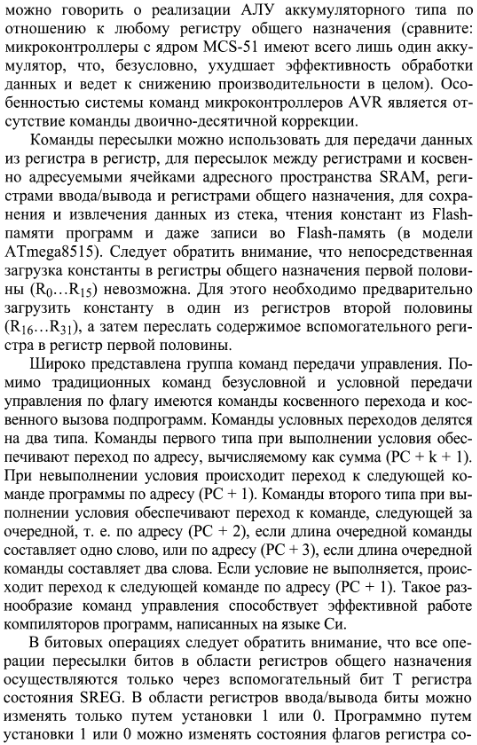
****

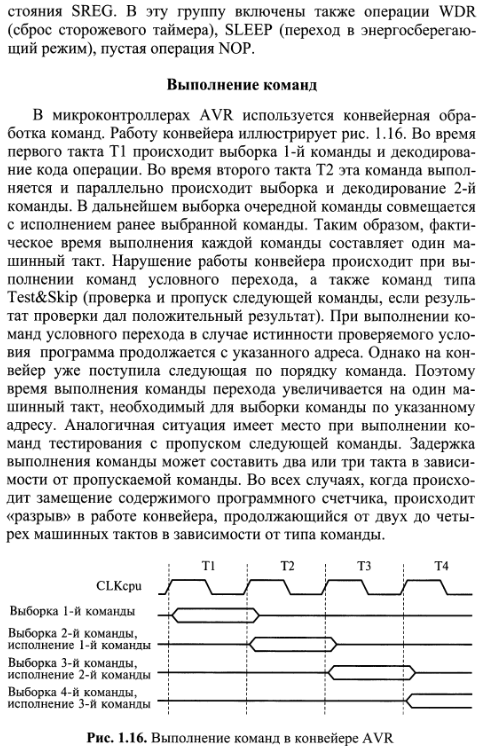
****

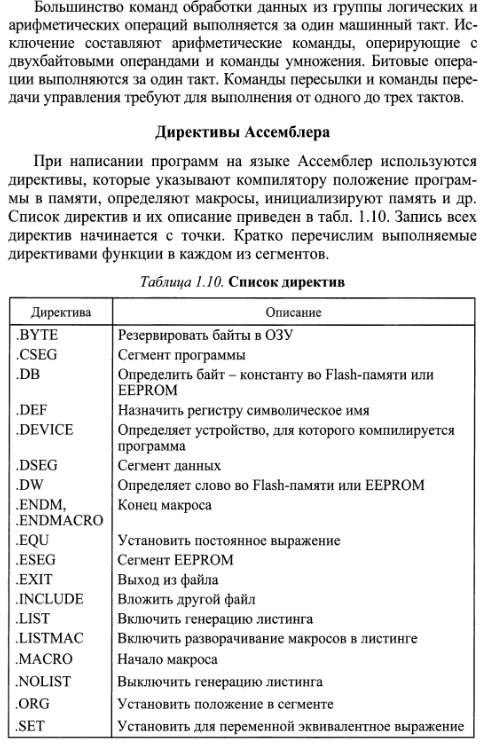
****

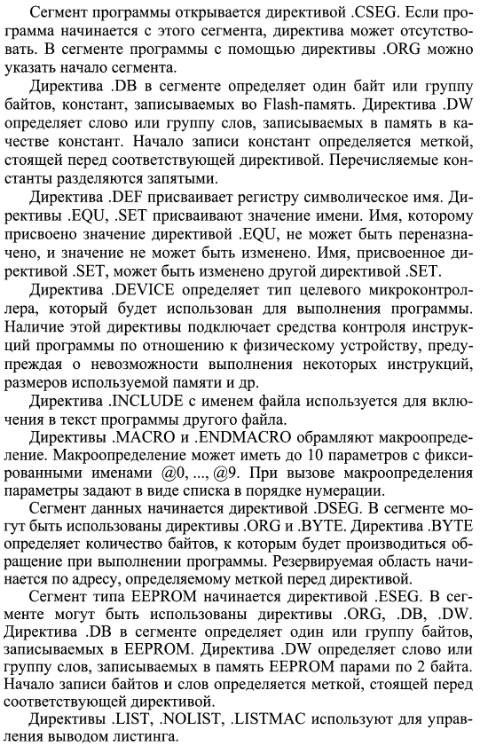
****

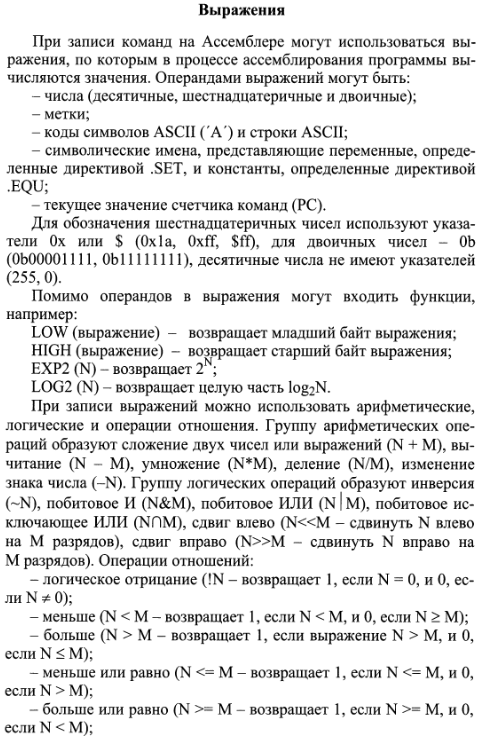
****

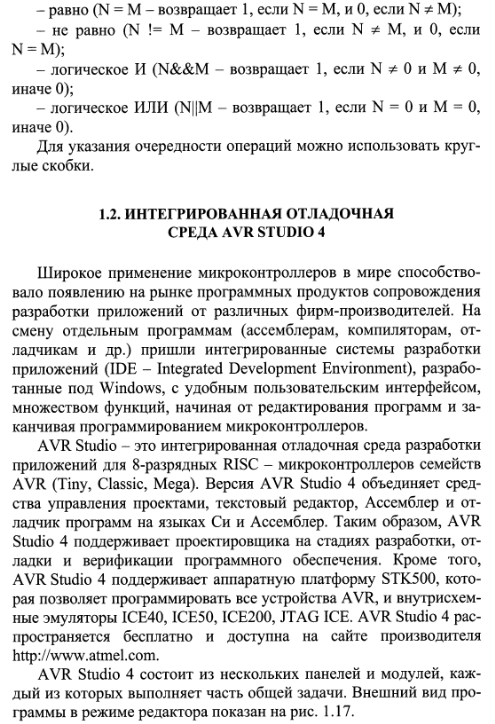
****

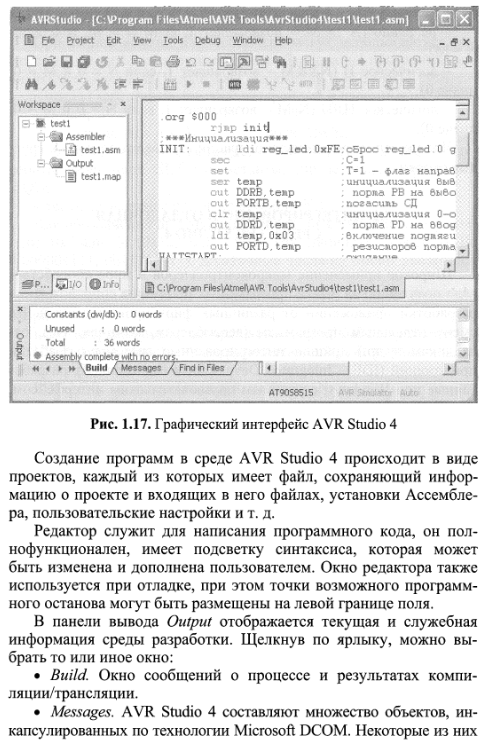
****

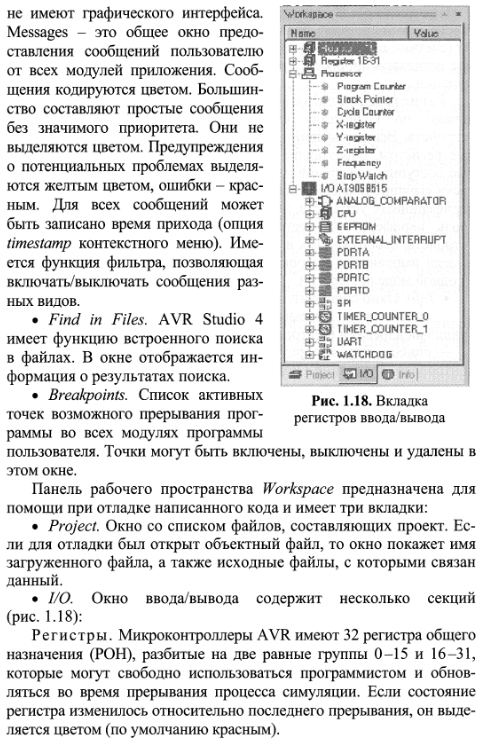
****

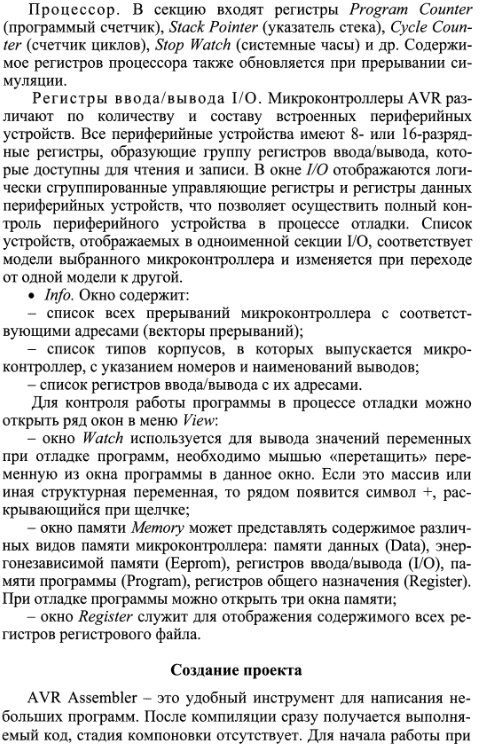
****

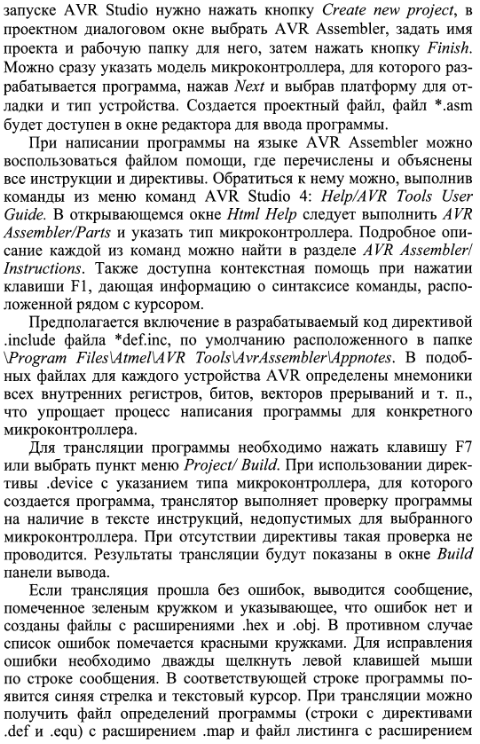
****

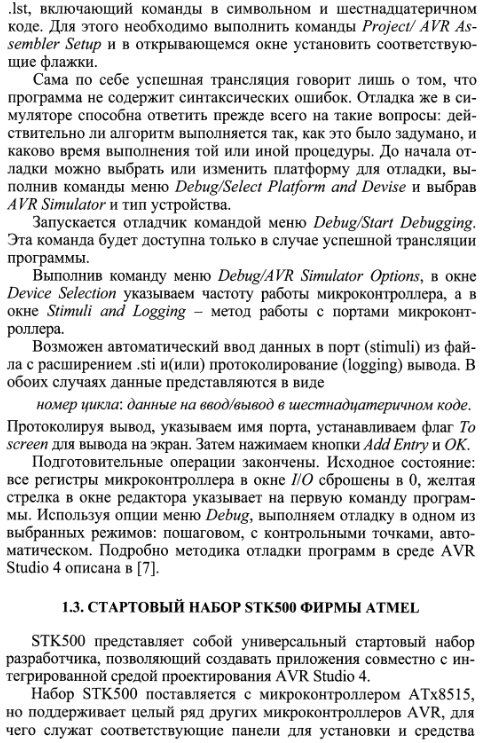
****

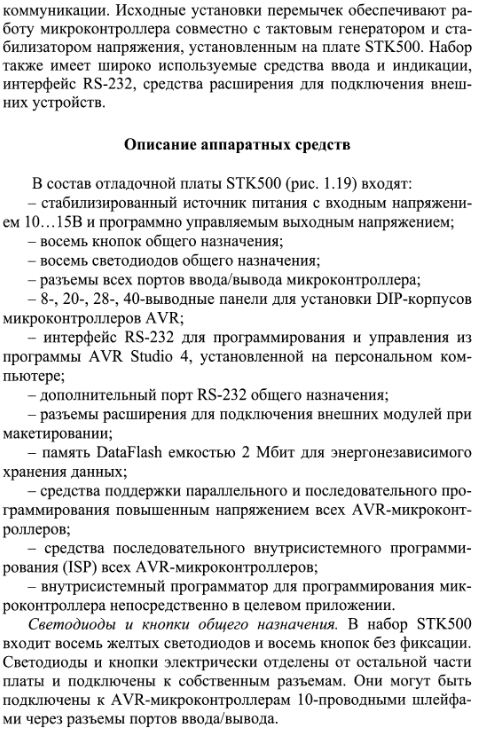
****

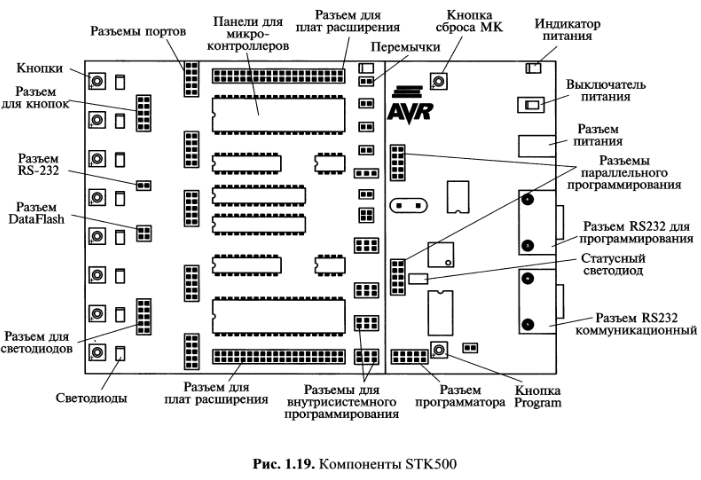
****

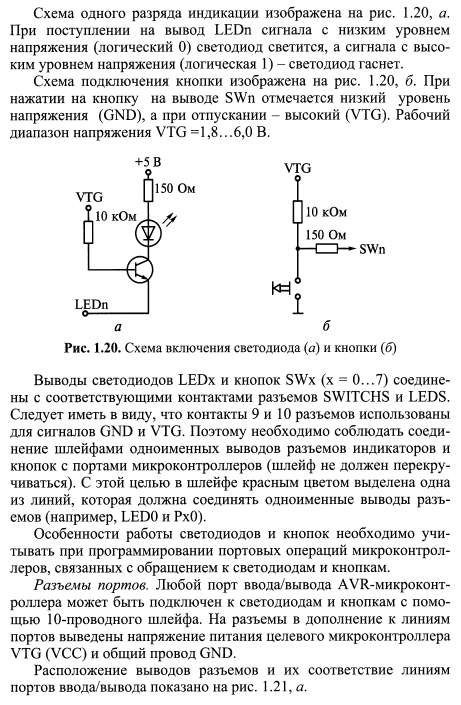
****

****

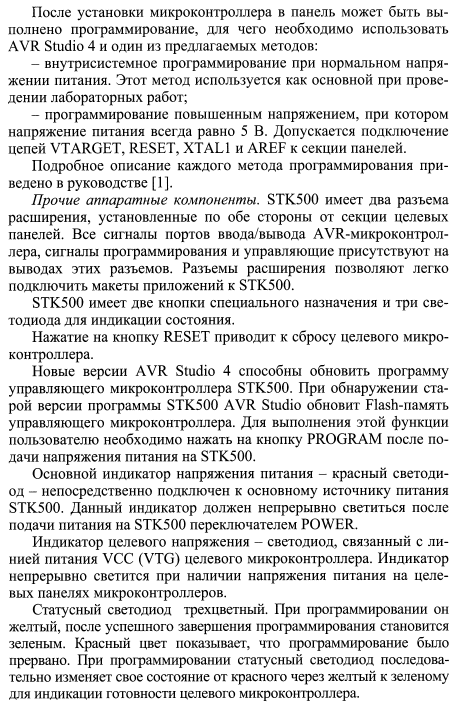
****

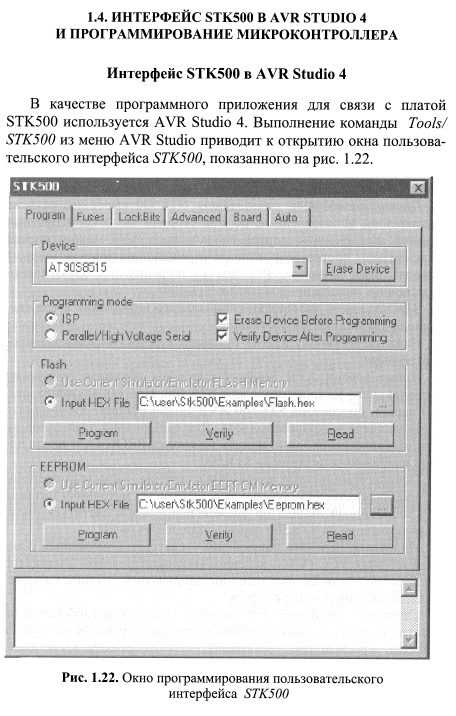
****

****

****

****

****

****

**Л2. Шины микропроцессорной системы и циклы обмена**

Самое главное, что должен знать разработчик микропроцессорных систем — это принципы организации обмена информацией *по* шинам таких систем. Без этого невозможно разработать аппаратную часть системы, а без аппаратной части не будет работать никакое *программное обеспечение*.

За более чем 30 лет, прошедших с момента появления первых микропроцессоров, были выработаны определенные правила обмена, которым следуют и разработчики новых микропроцессорных систем. Правила эти не слишком сложны, но твердо знать и неукоснительно соблюдать их для успешной работы необходимо. Как показала практика, принципы организации обмена *по* шинам гораздо важнее, чем особенности конкретных микропроцессоров. Стандартные системные магистрали живут гораздо дольше, чем тот или иной *процессор*. Разработчики новых процессоров ориентируются на уже существующие стандарты магистрали. Более того, некоторые системы на основе совершенно разных процессоров используют одну и ту же системную *магистраль*. То есть *магистраль* оказывается самым главным системообразующим фактором в микропроцессорных системах.

*Обмен информацией* в микропроцессорных системах происходит в циклах обмена информацией. Под циклом обмена информацией понимается временной *интервал*, в течение которого происходит выполнение одной элементарной *операции* обмена *по* шине. Например, пересылка кода данных из процессора в *память* или же пересылка кода данных из устройства ввода/вывода в *процессор*. В пределах одного *цикла* также может передаваться и несколько кодов данных, даже *целый* *массив* данных, но это встречается реже.

Циклы обмена информацией делятся на два основных типа:

* ***Цикл записи (вывода)***, в котором процессор записывает (выводит) информацию ;
* ***Цикл чтения (ввода)***, в котором процессор читает (вводит) информацию.

В некоторых микропроцессорных системах существует также цикл "чтение-модификация-*запись*" или же "ввод-пауза-*вывод*". В этих циклах *процессор* сначала читает информацию из памяти или устройства ввода/вывода, затем как-то преобразует ее и снова записывает *по* тому же адресу. Например, *процессор* может прочитать код из ячейки памяти, увеличить его на единицу и снова записать в эту же ячейку памяти. Наличие или отсутствие данного типа *цикла* связано с особенностями используемого процессора.

Особое *место* занимают **циклы прямого доступа к памяти (если режим *ПДП* в системе предусмотрен) и циклы запроса и предоставления прерывания (если прерывания в системе есть).** Когда в дальнейшем речь пойдет о таких циклах, это будет специально оговорено.

Во время каждого *цикла* устройства, участвующие в обмене информацией, передают друг другу информационные и *управляющие* сигналы в строго установленном порядке или, как еще говорят, в соответствии с принятым **протоколом обмена информацией**.

Длительность *цикла* обмена может быть постоянной или переменной, но она всегда включает в себя несколько периодов сигнала тактовой частоты системы. То есть даже в идеальном случае частота чтения информации процессором и частота записи информации оказываются в несколько раз меньше тактовой частоты системы.

**Чтение кодов команд из памяти системы также производится с помощью циклов чтения.** Поэтому в случае одношинной архитектуры на системной магистрали чередуются циклы чтения команд и циклы пересылки (чтения и записи) данных, но протоколы обмена остаются неизменными независимо от того, что передается — данные или команды. В случае двухшинной архитектуры циклы чтения команд и записи или чтения данных разделяются *по* разным шинам и могут выполняться одновременно.

### 2.1. Шины микропроцессорной системы

Прежде чем переходить к особенностям циклов обмена, остановимся подробнее на составе и назначении различных шин микропроцессорной системы.

Как уже упоминалось, в системную *магистраль* (системную шину) микропроцессорной системы входит три основные информационные шины: *адреса*, *данных* и *управления*.

***Шина данных*** — это основная шина, ради которой и создается вся система. **Количество ее разрядов (линий связи) определяет скорость и эффективность информационного обмена, а также максимально возможное количество команд.**

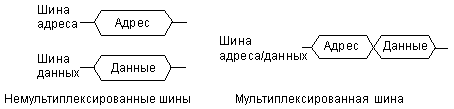
*Шина данных* всегда двунаправленная, так как предполагает передачу информации в обоих направлениях. Наиболее часто встречающийся тип выходного каскада для линий этой шины — *выход* с тремя состояниями.

Обычно *шина данных* имеет 8, 16, 32 или 64 разряда. Понятно, что за один цикл обмена *по* 64-разрядной шине может передаваться 8 *байт* информации, а *по* 8-разрядной — только один *байт*. *Разрядность* *шины данных* определяет и *разрядность* всей магистрали. Например, когда говорят о 32-разрядной системной магистрали, подразумевается, что она имеет 32-разрядную *шину данных*.

***Шина адреса*** — вторая по важности шина, которая определяет максимально возможную сложность микропроцессорной системы, то есть допустимый объем памяти и, следовательно, максимально возможный размер программы и максимально возможный объем запоминаемых данных. Количество адресов, обеспечиваемых*шиной адреса*, определяется как 2N, где N — количество разрядов. Например, 16-разрядная *шина адреса* обеспечивает 65 536 адресов. *Разрядность* *шины адреса* обычно кратна 4 и может достигать 32 и даже 64. *Шина адреса* может быть однонаправленной (когда магистралью всегда управляет только *процессор*) или двунаправленной (когда *процессор* может временно передавать управление магистралью другому устройству, например контроллеру *ПДП* ). Наиболее часто используются типы выходных каскадов с тремя состояниями или обычные ТТЛ (с двумя состояниями).

Как в *шине данных*, так и в *шине адреса* может использоваться **положительная логика** или **отрицательная логика**. При положительной логике высокий уровень напряжения соответствует логической единице на соответствующей линии связи, низкий — логическому нулю. При отрицательной логике — наоборот. В большинстве случаев уровни сигналов на шинах — ТТЛ.

Для снижения общего количества линий связи магистрали часто применяется ***мультиплексирование***   *шин адреса* и *данных*. То есть одни и те же линии связи используются в разные моменты времени для передачи как адреса, так и данных (в начале цикла — адрес, в конце цикла — данные). Для фиксации этих моментов (стробирования) служат специальные сигналы на *шине управления*. Понятно, что мультиплексированная *шина адреса* / *данных* обеспечивает меньшую скорость обмена, требует более длительного *цикла* обмена (рис. 2.1). *По* типу *шины адреса* и *шины данных* все магистрали также делятся на мультиплексированные и немультиплексированные.



**Рис. 2.1.**Мультиплексирование шин адреса и данных.

В некоторых мультиплексированных магистралях после одного кода адреса передается несколько кодов данных (*массив* данных). Это позволяет существенно повысить *быстродействие* магистрали. Иногда в магистралях применяется частичное *мультиплексирование*, то есть часть разрядов данных передается *по* немультиплексированным линиям, а другая часть — *по* мультиплексированным с адресом линиям.

***Шина управления*** — это вспомогательная шина, управляющие сигналы на которой определяют тип текущего цикла и фиксируют моменты времени, соответствующие разным частям или стадиям цикла. Кроме того, управляющие сигналы обеспечивают согласование работы процессора (или другого хозяина магистрали, задатчика, master) с работой памяти или устройства ввода/вывода (устройства-исполнителя, *slave*). *Управляющие* сигналы также обслуживают *запрос* и предоставление прерываний, *запрос* и предоставление прямого доступа.

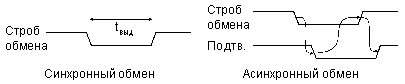
Сигналы *шины управления* могут передаваться как в положительной логике (реже), так и в отрицательной логике (чаще). Линии *шины управления* могут быть как однонаправленными, так и двунаправленными. Типы выходных каскадов могут быть самыми разными: с двумя состояниями (для однонаправленных линий), с тремя состояниями (для *двунаправленных линий*), с открытым коллектором (для двунаправленных и мультиплексированных линий).

Самые главные *управляющие* сигналы — это стробы обмена, то есть сигналы, формируемые процессором и определяющие моменты времени, в которые производится пересылка данных *по* *шине данных*, *обмен данными*. Чаще всего в магистрали используются два различных строба обмена:

* Строб записи (вывода), который определяет момент времени, когда устройство-исполнитель может принимать данные, выставленные процессором на *шину данных* ;
* Строб чтения (ввода), который определяет момент времени, когда устройство-исполнитель должно выдать на *шину данных* код данных, который будет прочитан процессором.

При этом большое *значение* имеет то, как *процессор* заканчивает обмен в пределах *цикла*, в какой момент он снимает свой строб обмена. Возможны два пути решения (рис. 2.2):

* При ***синхронном обмене*** процессор заканчивает обмен данными самостоятельно, через раз и навсегда установленный временной интервал выдержки (tвыд), то есть без учета интересов устройства-исполнителя;
* При ***асинхронном обмене*** процессор заканчивает обмен только тогда, когда устройство-исполнитель подтверждает выполнение операции специальным сигналом (так называемый режим *handshake* — рукопожатие).



**Рис. 2.2.**Синхронный обмен и асинхронный обмен.

Достоинства *синхронного обмена* — более простой *протокол обмена*, меньшее количество управляющих сигналов. Недостатки — отсутствие гарантии, что *исполнитель* выполнил требуемую операцию, а также высокие требования к быстродействию исполнителя.

Достоинства *асинхронного обмена* — более надежная пересылка данных, возможность работы с самыми разными *по* быстродействию исполнителями. Недостаток — необходимость формирования сигнала подтверждения всеми исполнителями, то есть дополнительные аппаратурные *затраты*.

Какой тип обмена быстрее, *синхронный* или *асинхронный*? Ответ на этот вопрос неоднозначен. С одной стороны, при *асинхронном обмене* требуется какое-то время на выработку, передачу дополнительного сигнала и на его обработку процессором. С другой стороны, при *синхронном обмене* приходится искусственно увеличивать длительность строба обмена для соответствия требованиям большего числа исполнителей, чтобы они успевали обмениваться информацией в темпе процессора. Поэтому иногда в магистрали предусматривают возможность как *синхронного*, так и *асинхронного обмена*, причем *синхронный обмен* является основным и довольно быстрым, а *асинхронный* применяется только для медленных исполнителей.

*По* используемому типу обмена магистрали микропроцессорных систем также делятся на синхронные и асинхронные.

### 2.2. Циклы обмена информацией

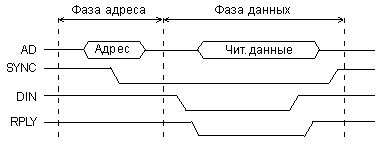
#### 2.2.1. Циклы программного обмена

Рассмотрим для примера два довольно типичных случая программного обмена по магистрали микропроцессорной системы.

Первый пример — это обмен по мультиплексированной асинхронной магистрали Q-bus, предложенной фирмой DEC и широко применявшейся в *микрокомпьютерах* и промышленных контроллерах. Упрощенные временные диаграммы циклов чтения (ввода) и записи (вывода) по этой магистрали приведены на рис. 2.3 и 2.4.

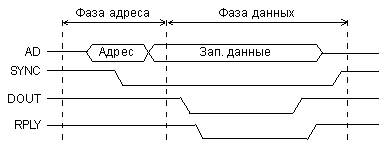
Отметим, что в дальнейшем тексте знак "минус" перед названием сигнала говорит о том, что *активный уровень сигнала* низкий, пассивный — высокий, то есть сигнал отрицательный. Если минуса перед названием сигнала нет, то сигнал положительный, его низкий уровень пассивный, а высокий — активный.

На *шине адреса* / *данных* ( AD ) в начале цикла обмена (в фазе адреса) процессор (задатчик) выставляет код адреса. На этой шине используется отрицательная логика. Средний уровень сигналов на шине AD обозначает, что состояния сигналов на шине в данные временные интервалы не важны. Для стробирования адреса используется отрицательный синхросигнал -*SYNC*, выставляемый также процессором. Его передний (отрицательный) фронт соответствует действительности кода адреса на шине AD. Фаза адреса одинакова в обоих циклах записи и чтения.



**Рис. 2.3.**Цикл чтения на магистрали Q-bus.

Получив (распознав) свой код адреса, устройство ввода/вывода или память (исполнитель) готовится к проведению обмена. Через некоторое время после начала (*отрицательного фронта) сигнала* -*SYNC* процессор снимает адрес и начинает фазу данных.



**Рис. 2.4.**Цикл записи на магистрали Q-bus.

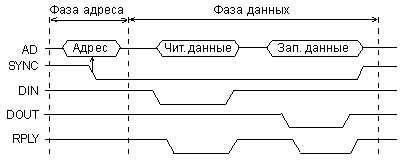
В фазе данных цикла чтения (рис. 2.3) процессор выставляет сигнал строба чтения данных -*DIN*, в ответ на который устройство, к которому обращается процессор (исполнитель), должно выставить свой код данных (читаемые данные). Одновременно это устройство должно подтвердить выполнение операции сигналом подтверждения обмена -RPLY.

Для сигнала -RPLY используется тип выходного каскада ОК, чтобы не было конфликтов между устройствами-исполнителями. Процессор, получив сигнал -RPLY, заканчивает цикл обмена. Для этого он снимает сигнал -*DIN* и сигнал -*SYNC*. Устройство-исполнитель в ответ на снятие сигнала -*DIN* должно снять код данных с шины AD и закончить сигнал подтверждения -RPLY. После этого процессор снимает сигнал -*SYNC*.

В фазе данных цикла записи (рис. 2.4) процессор выставляет на шину AD код записываемых данных и сопровождает его отрицательным сигналом строба записи данных -DOUT. Устройство-исполнитель должно по этому сигналу принять данные от процессора и сформировать сигнал подтверждения обмена -RPLY. Процессор, получив сигнал -RPLY, заканчивает цикл обмена. Для этого он снимает код данных с шины AD и сигнал -DOUT. Устройство-исполнитель в ответ на снятие сигнала -DOUT должно закончить сигнал подтверждения -RPLY. После этого процессор снимает сигнал -*SYNC*.

То есть на данной магистрали адрес передается синхронно (без подтверждения его получения исполнителем), а данные передаются асинхронно, с обязательным подтверждением их выдачи или приема исполнителем. Отсутствие сигнала подтверждения -RPLY в течение заданного времени воспринимается процессором как аварийная ситуация. В принципе возможна и асинхронная передача адреса, что увеличивает надежность обмена, хотя может снижать его скорость.

Помимо циклов чтения и записи на магистрали Q-bus используются также и циклы типа "ввод-пауза-вывод" ("чтение-модификация-запись"). Упрощенная временная диаграмма этого цикла представлена на рис. 2.5.



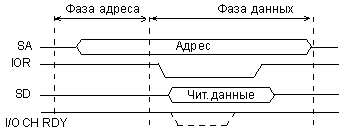
**Рис. 2.5.**Цикл "ввод-пауза-вывод" на магистрали Q-bus.

В этом цикле адресная фаза производится точно так же, как и в циклах чтения (ввода) и записи (вывода). Но в фазе данных процессор производит сначала чтение из заданного в адресной фазе адреса, а потом запись в тот же самый адрес. Для чтения используется строб чтения -*DIN*, а для записи – строб записи -DOUT. В ответ на сигнал -*DIN* устройство-исполнитель выдает свои данные на шину AD, а по сигналу -DOUT – принимает данные с шины AD. Как и в циклах чтения и записи, устройство-исполнитель подтверждает выполнение каждой операции сигналом подтверждения -RPLY. Понятно, что цикл "ввод-пауза-вывод" требует больше времени, чем каждый из циклов чтения или записи, но меньше времени, чем два последовательно произведенных цикла чтения и записи (так как для него нужна только одна адресная фаза). Сигнал -*SYNC* вырабатывается процессором в начале цикла "ввод-пауза-вывод" и держится до окончания всего цикла.

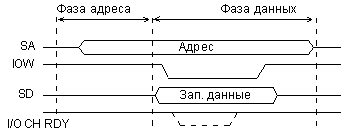
В качестве второго примера рассмотрим циклы обмена на синхронной немультиплексированной магистрали *ISA* (*Industrial Standard* Architecture), предложенной фирмой IBM и широко используемой в персональных компьютерах. Упрощенные циклы записи в устройство ввода/вывода и чтения из устройства ввода/вывода приведены на рис. 2.6 и 2.7.

Оба цикла начинаются с выставления процессором (задатчиком) кода адреса на *шину адреса* SA (логика на этой шине положительная). Адрес остается на шине SA до конца цикла. Фаза адреса, одинаковая для обоих циклов, заканчивается с началом строба обмена данными -IOR или -*IOW*. В течение фазы адреса устройство-исполнитель должно принять код адреса и распознать или не распознать его. Если адрес распознан, исполнитель готовится к обмену.

В фазе данных цикла чтения (рис. 2.6) процессор выставляет *отрицательный сигнал* чтения данных из устройства ввода/вывода -IOR. В ответ на него устройство-исполнитель должно выдать на *шину данных*   *SD* свой код данных (читаемые данные). Логика на *шине данных* положительная. Через установленное время строб обмена -IOR снимается процессором, после чего снимается также и код адреса с шины SA. Цикл заканчивается без учета быстродействия исполнителя.



**Рис. 2.6.**Цикл чтения из УВВ на магистрали ISA.



**Рис. 2.7.**Цикл записи в УВВ на магистрали ISA.

Но так происходит только в случае основного, *синхронного обмена*. Кроме него на магистрали *ISA* также предусмотрена возможность *асинхронного обмена*. Для этого применяется сигнал готовности канала (магистрали) I/O CH RDY. Тип выходного каскада для данного сигнала — ОК, для предотвращения конфликтов между устройствами-исполнителями. При *синхронном обмене* сигнал I/O CH RDY всегда положительный. Но медленное устройство-исполнитель, не успевающее работать в темпе процессора, может этот сигнал снять, то есть сделать нулевым сразу после начала строба обмена. Тогда процессор до того момента, пока сигнал I/O CH RDY не станет снова положительным, приостанавливает завершение цикла, продлевает строб обмена. Конечно, слишком большая длительность этого сигнала рассматривается как аварийная ситуация. Для простоты понимания можно считать, что устройство-исполнитель формирует в данном случае*отрицательный сигнал* неготовности завершить обмен. На время этого сигнала обмен на магистрали приостанавливается.

Принципиальное отличие *асинхронного обмена* по магистрали *ISA* от *асинхронного обмена* по магистрали Q-bus состоит в следующем. Если в случае Q-bus сигнал подтверждения обязателен, и его должен формировать каждый исполнитель, то в случае *ISA* сигнал о неготовности исполнитель может не формировать, если он успевает работать в темпе процессора. Зато в случае Q-bus к концу цикла обмена процессор всегда уверен, что устройство-исполнитель выполнило требуемую операцию, а в случае *ISA* такой уверенности нет.

В фазе данных цикла записи по магистрали *ISA* (рис. 2.7) процессор выставляет на *шину данных*   *SD* код записываемых данных и сопровождает их стробом записи данных в устройство ввода/вывода -*IOW*. Получив этот сигнал, устройство-исполнитель должно принять с шины *SD* код записываемых данных. Если оно не успевает сделать это в темпе процессора, то оно может снять на нужное время сигнал I/O CH RDY после получения переднего фронта сигнала -*IOW*. Тогда процессор приостановит окончание цикла записи.

Рассмотренные примеры, конечно, не раскрывают всех тонкостей обмена по упомянутым магистралям. Они всего лишь иллюстрируют главные принципы обмена по ним.

#### 2.2.2. Циклы обмена по прерываниям

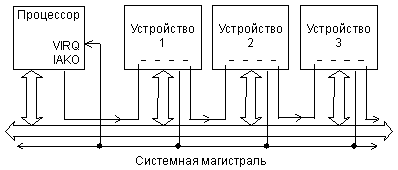
Циклы обмена в режиме прерываний строятся по тем же принципам, что и циклы программного обмена, но имеют ряд специфических особенностей.

Прерывания в микропроцессорных системах бывают двух основных типов:

* *векторные прерывания*, которые требуют проведения цикла чтения по магистрали;
* *радиальные прерывания*, которые не требуют никакого цикла обмена по магистрали.

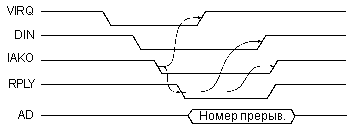
Дело в том, что прерываний в микропроцессорной системе обычно бывает много. Поэтому процессору необходима информация о номере (или, как еще говорят, об адресе вектора) конкретного прерывания. Эта информация может быть передана процессору двумя путями.

При ***векторном прерывании*** код номера прерывания передается процессору тем устройством ввода/вывода, которое данное прерывание запросило. Для этого процессор проводит цикл чтения по магистрали, и по *шине данных* получает код номера прерывания. *Шина адреса* в данном цикле обычно не используется, так как устройство, запросившее прерывание, и так знает, что процессор будет обращаться именно к нему. В этом случае в магистрали достаточно всего одной линии *запроса прерывания* для всех устройств ввода/вывода. Так организованы прерывания, например, в магистрали Q-bus.



**Рис. 2.8.**Сигналы запроса и предоставления прерывания в магистрали Q-bus.

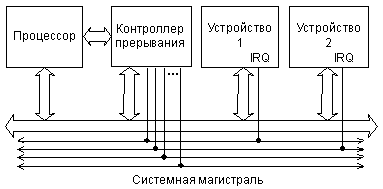
Схема распространения сигналов, участвующих в прерываниях на магистрали Q-bus, показана на рис. 2.8. Упрощенная временная диаграмма цикла запроса и предоставления магистрали представлена на рис. 2.9.



**Рис. 2.9.**Цикл запроса/предоставления векторного прерывания на магистрали Q-bus.

*Запрос прерывания* осуществляется отрицательным сигналом -VIRQ, который может формироваться каждым из устройств, запрашивающих прерывание. Тип выходного каскада для этого сигнала — ОК, чтобы избежать конфликтов между запрашивающими прерывания устройствами. Получив сигнал -VIRQ, процессор предоставляет прерывание (закончив предварительно выполнение текущей команды). Для этого он выставляет сигнал чтения данных -*DIN* и сигнал предоставления прерывания IAKO. Этот сигнал IAKO последовательно проходит через все устройства, которые могут запрашивать прерывания. Если устройство запросило прерывание, то оно не пропускает через себя этот сигнал. В результате получается, что если прерывания одновременно запросили два или более устройств, то сигнал предоставления прерывания получит только одно устройство, а именно то, которое ближе к процессору. Такой механизм разрешения конфликтов называется иногда географическим приоритетом (или цепочечным приоритетом, Daisy Chain). Получив сигнал IAKO, устройство, запросившее прерывание, должно снять свой сигнал -VIRQ.

Затем процессор проводит цикл безадресного чтения номера прерывания. В ответ на полученные сигналы -*DIN* и IAKO устройство, которому предоставлено прерывание, должно выдать на *шину адреса* / *данных*   AD код номера прерывания (адреса вектора прерывания) и выставить сигнал подтверждения -RPLY. Процессор читает код номера прерывания и заканчивает цикл безадресного чтения снятием сигналов -*DIN* и IAKO.



**Рис. 2.10.**Структура связей для организации радиальных прерываний на магистрали ISA.

При ***радиальном прерывании*** в магистрали имеется столько линий *запроса прерывания*, сколько всего может быть разных прерываний. То есть каждое устройство ввода/вывода, желающее использовать прерывание, подает сигнал *запроса прерывания* по своей отдельной линии. Процессор узнает о номере прерывания по номеру линии, по которой пришел сигнал *запроса прерывания*. Никаких циклов обмена по магистрали при этом не требуется. В случае *радиальных прерываний* в систему обычно включается дополнительная микросхема контроллера прерываний, обрабатывающая сигналы *запросов прерываний*. Именно так организованы прерывания, например, в магистрали *ISA*.

Упрощенная структура связей между устройствами, участвующими в обмене по прерываниям, на магистрали *ISA* показана на рис. 2.10. Процессор общается с контроллером прерываний как по магистрали (чтобы задать ему режимы работы), так и вне магистрали (при обработке *запросов на прерывание* ). Сигналы *запросов прерываний*   IRQ распределяются между всеми устройствами магистрали. На каждую линию IRQ приходится одно устройство. Тип выходного каскада для этих линий — 2С, так как конфликты здесь не предусмотрены. *Запросом прерывания* является передний, положительный фронт сигнала IRQ. При одновременном поступлении сигналов IRQ от нескольких устройств порядок их обслуживания определяется контроллером прерываний.

Какой тип прерываний лучше — *векторный* или *радиальный*?

*Векторные прерывания* обеспечивают системе большую гибкость, в системе их может быть очень много. Но зато они требуют дополнительных аппаратурных узлов во всех устройствах, запрашивающих прерывания, для обслуживания циклов безадресного чтения.

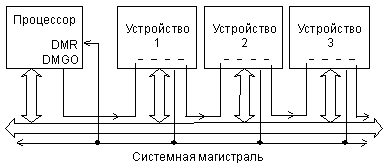
*Радиальных прерываний* в системе обычно не очень много (от 1 до 16). При этом типе прерываний, как правило, требуется введение в систему специального контроллера прерываний. Каждое *радиальное прерывание* требует введения дополнительной линии в *шину управления* системной магистрали. Но работать с*радиальными прерываниями* проще, так как все сводится только к выработке единственного сигнала IRQ, и никаких циклов обмена по магистрали не требуется.

#### 2.2.3. Циклы обмена в режиме ПДП

Циклы обмена в режиме прямого доступа к памяти выполняются по тем же правилам, что и циклы программного обмена, и циклы предоставления прерываний.

Прежде чем начать обмен в режиме *ПДП*, устройство, которому необходим *ПДП*, должно запросить *ПДП* и получить его. Процедура запроса и предоставления *ПДП* очень похожа на процедуру запроса и предоставления прерывания. В обоих случаях устройство, требующее обслуживания, посылает сигнал запроса процессору. Однако в случае *ПДП* процессор обязательно должен предоставить *ПДП* запросившему устройству с помощью специальных сигналов, так как на время *ПДП* процессор отключается от магистрали. А при *радиальных прерываниях* предоставления прерывания от процессора не требуется.

На магистрали Q-bus запрос и предоставление *ПДП* организуются подобно запросу и предоставлению прерывания. Упрощенная структура связей устройств, участвующих в *ПДП*, показана на рис. 2.11. Временная диаграмма запроса/предоставления *ПДП* очень близка к временной диаграмме запроса/предоставления прерывания (см. рис. 2.9).



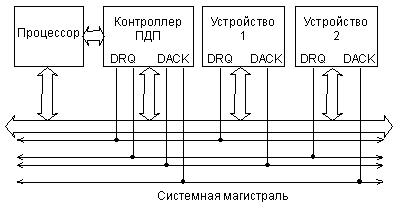
**Рис. 2.11.**Структура связей запроса/предоставления ПДП на магистрали Q-bus.

Сигнал запроса *ПДП*, называемый -DMR, передается всеми устройствами, нуждающимися в *ПДП*, по одной линии магистрали. Тип выходного каскада на этой линии — ОК. Процессор, получив сигнал -DMR, выдает сигнал предоставления *ПДП*   DMGO, аналогичный сигналу IAKO. Этот сигнал также проходит через все устройства последовательно, в результате чего *ПДП* получает только то устройство, которое находится ближе к процессору (географический приоритет). А затем устройство, получившее *ПДП*, проводит циклы обмена по магистрали, аналогично циклам программного обмена. В циклах *ПДП* информация читается из памяти и записывается в устройство ввода/вывода, или наоборот — читается из устройства ввода/вывода и передается в память.

На магистрали *ISA* запрос/предоставление *ПДП* очень напоминает организацию *радиальных прерываний* (рис. 2.12). Точно так же в системе существует контроллер *ПДП*, к которому сходятся сигналы запроса *ПДП*, называемые DRQ, и от которого расходятся сигналы предоставления *ПДП*, называемые -DACK. К каждому каналу *ПДП* (пара сигналов DRQ и -DACK ) подключается только одно устройство, запрашивающее *ПДП*. Тип выходных каскадов для этих сигналов —2С. Устройство, нуждающееся в *ПДП*, посылает сигнал запроса DRQ и получает в ответ сигнал предоставления -DACK. После этого контроллер *ПДП* проводит циклы обмена по магистрали между устройством ввода/вывода и памятью.

Упрощенная временная диаграмма циклов *ПДП* на магистрали *ISA* показана на рис. 2.13.

На магистрали *ISA* используются раздельные стробы записи в память ( -MEMW ) и записи в устройства ввода/вывода ( -*IOW* ), а также раздельные стробы чтения из памяти ( -MEMR ) и чтения из устройств ввода/вывода ( -IOR ). Это позволяет за один цикл обмена *ПДП* читать информацию из памяти и записывать ее в устройство ввода/вывода или же читать информацию из устройства ввода/вывода и записывать ее в память. При этом на *шине адреса* выставляется адрес памяти, а адрес устройства ввода/вывода заменяется одним-единственным сигналом AEN. Естественно, в цикле обмена в режиме *ПДП* участвует только то устройство ввода/вывода, которое предварительно запросило *ПДП* и которому *ПДП* было предоставлено. Поэтому никаких конфликтов между устройствами ввода/вывода из-за такой упрощенной адресации не возникает.



**Рис. 2.12.**Структура связей запроса/предоставления ПДП на магистрали ISA.



**Рис. 2.13.**Цикл ПДП на магистрали ISA.

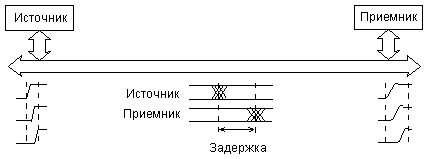
### 2.3. Прохождение сигналов по магистрали

При организации обмена *по* магистралям и шинам разработчику необходимо учитывать несколько важных моментов, связанных как с особенностью распространения сигналов *по* шинам, так и с самой природой шин. В противном случае *микропроцессорная система* может попросту не работать или работать неустойчиво, хотя вся логика цифровых устройств, входящих в систему, будет спроектирована безошибочно.

В случае, когда системная *шина* (*магистраль*) микропроцессорной системы является внешней, а не скрыта внутри микросхемы, необходимо учитывать особенности распространения сигналов *по* длинным линиям. Хотя в большинстве случаев *длина* магистрали не слишком велика, не превышает 1—2 десятков сантиметров, это все равно оказывает большое влияние на синхронизацию обмена.

На прохождение сигналов *по* магистрали влияют следующие факторы:

* конечная величина задержки распространения сигналов по линиям магистрали;
* различие задержек распространения сигналов по разным линиям шины;
* неодновременное выставление сигналов на линии шины;
* искажение фронтов сигналов, проходящих по линиям магистрали;
* отражение сигналов от концов линий связи (рис. 2.14).



**Рис. 2.14.**Прохождение сигналов по шине.

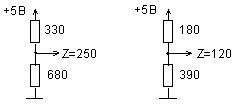
Для учета всех этих факторов разработчики стандартных магистралей обмена и стандартных протоколов обмена всегда закладывают необходимые задержки между сигналами, участвующими в обмене. Кроме того, задержки между сигналами выбираются таким образом, чтобы устройство, которому адресован тот или иной сигнал, имело достаточно времени для его обработки. Если разрабатывается новая *магистраль*, все это тоже надо учитывать.

Поэтому пытаться "модернизировать" какой-то *стандартный протокол* и ускорять обмен *по* магистрали путем уменьшения задержек, предусмотренных стандартом, очень опасно. Точно так же опасно, не изменяя протокола обмена, пытаться увеличить длину магистрали, увеличивая тем самым задержки распространения сигналов *по* линиям и шинам. Особенно чувствительны к такого рода "модернизациям" синхронные магистрали, в которых не предусмотрено обязательное подтверждение выполнения каждой *операции*.

Например, длительность фазы адреса в цикле обмена выбирается таким образом. В течение адресной фазы все сигналы всех разрядов кода адреса, пусть даже и сформированные процессором не одновременно, должны дойти до устройства-исполнителя *по* своим проводам шины. А устройство-*исполнитель* должно этот код адреса принять и обработать (то есть отличить свой *адрес* от чужого). Естественно, для гарантии в длительность адресной фазы еще добавляется небольшая дополнительная задержка.

Точно так же длительность фазы данных в цикле чтения должна выбираться такой, чтобы устройство-*исполнитель* успело получить строб чтения и выдать код читаемых данных на *шину данных*. Затем этот код должен успеть дойти до процессора и *процессор* должен успеть его прочитать. После чего *процессор* снимает сигнал строба чтения, этот задний *фронт сигнала* доходит с задержкой до устройства-исполнителя, которое также с задержкой снимает свой код данных. Аналогично и в цикле записи.

Для улучшения формы сигналов, распространяющихся *по* магистрали, иногда применяют оконечные согласователи (терминаторы) на концах линий магистрали. Особенно важно их применение в случае, когда допустимая *длина* магистрали превышает несколько метров. Например, в случае магистрали Q-bus применяются два типа согласователей: 120-омный и 250-омный (рис. 2.15).



**Рис. 2.15.**Оконечные согласователи на магистрали Q-bus.

Включение согласователей предъявляет дополнительные требования к нагрузочной способности передатчиков, работающих на линии магистрали. В магистрали *ISA* подобные согласователи не используются, хотя к некоторым линиям присоединены резисторы, соединенные другим своим выводом с шиной питания (прежде всего это линии, тип выходного каскада для которых — ОК).

В любом случае выходные каскады передатчиков, работающих на линии магистрали, должны обеспечивать высокие выходные токи, так как к магистрали может подключаться несколько устройств, каждое из которых потребляет *входной* ток. Типичные величины требуемых выходных токов магистральных передатчиков находятся в пределах 20—30 мА. В то же время входные токи магистральных приемников должны быть малыми, чтобы не перегружать передатчики. Типичные величины допустимых входных токов магистральных приемников лежат в пределах 0,2—0,8 мА.

**Лекция 3. Функции устройств магистрали**

### 3.1. Функции устройств магистрали

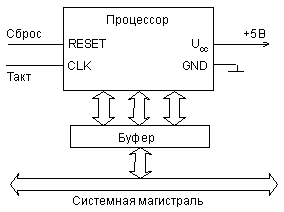
Рассмотрим теперь, как взаимодействуют на магистрали основные устройства *микропроцессорной системы*: *процессор*, *память* (оперативная и постоянная), устройства ввода/вывода.

#### 3.1.1. Функции процессора

**Процессор (рис. 3.1) обычно представляет собой отдельную микросхему или же часть микросхемы (в случае *микроконтроллера*).** В прежние годы процессор иногда выполнялся на комплектах из нескольких микросхем, но сейчас от такого подхода уже практически отказались. Микросхема процессора обязательно имеет выводы трех шин: *шины адреса*, *шины данных* и шины управления. Иногда некоторые сигналы и шины мультиплексируются, чтобы уменьшить количество выводов микросхемы процессора.

Важнейшие характеристики процессора — это количество разрядов его *шины данных*, количество разрядов его *шины адреса* и количество управляющих сигналов в *шине управления*. Разрядность *шины данных* определяет скорость работы системы. Разрядность *шины адреса* определяет допустимую сложность системы. Количество линий управления определяет разнообразие режимов обмена и эффективность обмена процессора с другими устройствами системы.

Кроме выводов для сигналов трех основных шин процессор всегда имеет вывод (или два вывода) для подключения внешнего *тактового сигнала* или кварцевого резонатора (*CLK*), так как процессор всегда представляет собой тактируемое устройство. Чем больше тактовая частота процессора, тем он быстрее работает, то есть тем быстрее выполняет команды. Впрочем, быстродействие процессора определяется не только тактовой частотой, но и особенностями его структуры. Современные процессоры выполняют большинство команд за один такт и имеют средства для параллельного выполнения нескольких команд. Тактовая частота процессора не связана прямо и жестко со скоростью обмена по магистрали, так как скорость обмена по магистрали ограничена *задержками распространения* сигналов и искажениями сигналов на магистрали. То есть тактовая частота процессора определяет только его внутреннее быстродействие, а не внешнее. Иногда тактовая частота процессора имеет нижний и верхний пределы. При превышении верхнего предела частоты возможно перегревание процессора, а также сбои, причем, что самое неприятное, возникающие не всегда и нерегулярно. Так что с изменением этой частоты надо быть очень осторожным.



**Рис. 3.1.**Схема включения процессора.

Еще один важный сигнал, который имеется в каждом процессоре, — это сигнал начального сброса RESET. При включении питания, при аварийной ситуации или зависании процессора подача этого сигнала приводит к инициализации процессора, заставляет его приступить к выполнению программы начального запуска. Аварийная ситуация может быть вызвана помехами по цепям питания и "земли", сбоями в работе памяти, внешними ионизирующими излучениями и еще множеством причин. В результате процессор может потерять контроль над выполняемой программой и остановиться в каком-то адресе. Для выхода из этого состояния как раз и используется сигнал начального сброса. Этот же вход начального сброса может использоваться для оповещения процессора о том, что напряжение питания стало ниже установленного предела. В таком случае процессор переходит к выполнению программы сохранения важных данных. По сути, этот вход представляет собой особую разновидность *радиального прерывания*.

Иногда у микросхемы процессора имеется еще один-два входа *радиальных прерываний* для обработки особых ситуаций (например, для прерывания от внешнего таймера).

Шина питания современного процессора обычно имеет одно напряжение питания (+5В или +3,3В) и общий провод ("землю"). Первые процессоры нередко требовали нескольких напряжений питания. В некоторых процессорах предусмотрен режим пониженного *энергопотребления*. Вообще, современные микросхемы процессоров, особенно с высокими тактовыми частотами, потребляют довольно большую мощность. В результате для поддержания нормальной рабочей температуры корпуса на них нередко приходится устанавливать радиаторы, вентиляторы или даже специальные микрохолодильники.

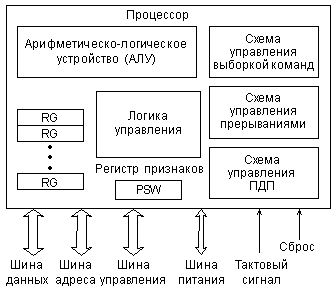
Для подключения процессора к магистрали используются буферные микросхемы, обеспечивающие, если необходимо, демультиплексирование сигналов и электрическое буферирование сигналов магистрали. Иногда *протоколы обмена* по системной магистрали и по шинам процессора не совпадают между собой, тогда буферные микросхемы еще и согласуют эти протоколы друг с другом. Иногда в микропроцессорной системе используется несколько *магистралей* (системных и локальных), тогда для каждой из *магистралей* применяется свой буферный узел. Такая структура характерна, например, для персональных компьютеров.

После включения питания процессор переходит в первый адрес программы начального пуска и выполняет эту программу. Данная программа предварительно записана в постоянную (энергонезависимую) память. После завершения программы начального пуска процессор начинает выполнять основную программу, находящуюся в постоянной или оперативной памяти, для чего выбирает по очереди все команды. От этой программы процессор могут отвлекать внешние прерывания или запросы на *ПДП*. Команды из памяти процессор выбирает с помощью циклов чтения по магистрали. При необходимости процессор записывает данные в память или в устройства ввода/вывода с помощью циклов записи или же читает данные из памяти или из устройств ввода/вывода с помощью циклов чтения.

Таким образом, основные функции любого процессора следующие:

* выборка (чтение) выполняемых команд;
* ввод (чтение) данных из памяти или устройства ввода/вывода;
* вывод (запись) данных в память или в устройства ввода/вывода;
* обработка данных (операндов), в том числе арифметические операции над ними;
* адресация памяти, то есть задание адреса памяти, с которым будет производиться обмен;
* обработка прерываний и режима прямого доступа.

Упрощенно структуру микропроцессора можно представить в следующем виде (рис. 3.2).



**Рис. 3.2.**Внутренняя структура микропроцессора.

Основные функции показанных узлов следующие.

**Схема управления выборкой команд** выполняет чтение команд из памяти и их дешифрацию. В первых микропроцессорах было невозможно одновременное выполнение предыдущей команды и выборка следующей команды, так как процессор не мог совмещать эти операции. Но уже в 16-разрядных процессорах появляется так называемый конвейер (очередь) команд, позволяющий выбирать несколько следующих команд, пока выполняется предыдущая. Два процесса идут параллельно, что ускоряет работу процессора. Конвейер представляет собой небольшую внутреннюю память процессора, в которую при малейшей возможности (при освобождении внешней шины) записывается несколько команд, следующих за исполняемой. Читаются эти команды процессором в том же порядке, что и записываются в конвейер (это память типа FIFO, First In — First Out, первый вошел — первый вышел). Правда, если выполняемая команда предполагает переход не на следующую ячейку памяти, а на удаленную (с меньшим или большим адресом), конвейер не помогает, и его приходится сбрасывать. Но такие команды встречаются в программах сравнительно редко.

Развитием идеи конвейера стало использование внутренней кэш-памяти процессора, которая заполняется командами, пока процессор занят выполнением предыдущих команд. Чем больше объем кэш-памяти, тем меньше вероятность того, что ее содержимое придется сбросить при команде перехода. Понятно, что обрабатывать команды, находящиеся во внутренней памяти, процессор может гораздо быстрее, чем те, которые расположены во внешней памяти. В кэш-памяти могут храниться и данные, которые обрабатываются в данный момент, это также ускоряет работу. Для большего ускорения выборки команд в современных процессорах применяют совмещение выборки и дешифрации, одновременную дешифрацию нескольких команд, несколько параллельных конвейеров команд, предсказание *команд переходов* и некоторые другие методы.

***Арифметико-логическое устройство*** (или *АЛУ*, *ALU* ) предназначено для обработки информации в соответствии с полученной процессором командой. Примерами обработки могут служить логические операции (типа логического "И", "ИЛИ", "Исключающего ИЛИ" и т.д.) то есть *побитные операции* над операндами, а также арифметические операции (типа сложения, вычитания, умножения, деления и т.д.). Над какими кодами производится операция, куда помещается ее результат — определяется выполняемой командой. Если команда сводится всего лишь к пересылке данных без их обработки, то *АЛУ* не участвует в ее выполнении.

Быстродействие *АЛУ* во многом определяет производительность процессора. Причем важна не только частота *тактового сигнала*, которым тактируется *АЛУ*, но и количество тактов, необходимое для выполнения той или иной команды. Для повышения производительности разработчики стремятся довести время выполнения команды до одного такта, а также обеспечить работу *АЛУ* на возможно более высокой частоте. Один из путей решения этой задачи состоит в уменьшении количества выполняемых *АЛУ* команд, создание процессоров с уменьшенным набором команд (так называемые *RISC*-процессоры). Другой путь повышения производительности процессора — использование нескольких параллельно работающих *АЛУ*.

Что касается операций над числами с плавающей точкой и других специальных сложных операций, то в системах на базе первых процессоров их реализовали последовательностью более простых команд, специальными подпрограммами, однако затем были разработаны специальные вычислители — *математические сопроцессоры*, которые заменяли основной процессор на время выполнения таких команд. В современных микропроцессорах *математические сопроцессоры* входят в структуру как составная часть.

*Регистры* процессора представляют собой по сути ячейки очень быстрой памяти и служат для временного хранения различных кодов: данных, адресов, служебных *кодов. Операции* с этими кодами выполняются предельно быстро, поэтому, в общем случае, чем больше внутренних *регистров*, тем лучше. Кроме того, на быстродействие процессора сильно влияет разрядность *регистров*. Именно разрядность *регистров* и *АЛУ* называется **внутренней разрядностью процессора**, которая может не совпадать с **внешней разрядностью**.

По отношению к назначению внутренних *регистров* существует два основных подхода. Первого придерживается, например, компания Intel, которая каждому *регистру* отводит строго определенную функцию. С одной стороны, это упрощает организацию процессора и уменьшает время выполнения команды, но с другой — снижает гибкость, а иногда и замедляет работу программы. Например, некоторые арифметические операции и обмен с устройствами ввода/вывода проводятся только через один *регистр* — **аккумулятор**, в результате чего при выполнении некоторых процедур может потребоваться несколько дополнительных пересылок между *регистрами*. Второй подход состоит в том, чтобы все (или почти все) *регистры* сделать равноправными, как , например, в 16-разрядных процессорах Т-11 фирмы DEC. При этом достигается высокая гибкость, но необходимо усложнение структуры процессора. Существуют и промежуточные решения, в частности, в процессоре MC68000 фирмы Motorola половина *регистров* использовалась для данных, и они были взаимозаменяемы, а другая половина — для адресов, и они также взаимозаменяемы.

*Регистр* признаков ( *регистр* состояния) занимает особое место, хотя он также является внутренним *регистром* процессора. Содержащаяся в нем информация — это не данные, не адрес, а слово состояния процессора (*ССП*, *PSW* — Processor Status Word). Каждый бит этого слова (флаг) содержит информацию о результате предыдущей команды. Например, есть бит нулевого результата, который устанавливается в том случае, когда результат выполнения предыдущей команды — нуль, и очищается в том случае, когда результат выполнения команды отличен от нуля. Эти биты (флаги) используются командами *условных переходов*, например, командой перехода в случае нулевого результата. В этом же *регистре* иногда содержатся флаги управления, определяющие режим выполнения некоторых команд.

**Схема управления прерываниями обрабатывает** поступающий на процессор *запрос прерывания*, определяет адрес начала программы обработки прерывания (адрес вектора прерывания), обеспечивает переход к этой программе после выполнения текущей команды и сохранения в памяти (в *стеке* ) текущего состояния*регистров* процессора. По окончании программы обработки прерывания процессор возвращается к прерванной программе с восстановленными из памяти (из *стека* ) значениями внутренних *регистров*. Подробнее о *стеке* будет рассказано в следующем разделе.

**Схема управления прямым доступом к памяти** служит для временного отключения процессора от внешних шин и приостановки работы процессора на время предоставления прямого доступа запросившему его устройству.

**Логика управления** организует взаимодействие всех узлов процессора, перенаправляет данные, синхронизирует работу процессора с внешними сигналами, а также реализует процедуры ввода и вывода информации.

Таким образом, в ходе работы процессора схема выборки команд выбирает последовательно команды из памяти, затем эти команды выполняются, причем в случае необходимости обработки данных подключается *АЛУ*. На входы *АЛУ* могут подаваться обрабатываемые данные из памяти или из внутренних *регистров*. Во внутренних *регистрах* хранятся также коды адресов обрабатываемых данных, расположенных в памяти. Результат обработки в *АЛУ* изменяет состояние *регистра* признаков и записывается во внутренний *регистр* или в память (как источник, так и приемник данных указывается в составе кода команды). При необходимости информация может переписываться из памяти (или из устройства ввода/вывода) во внутренний *регистр* или из внутреннего *регистра* в память (или в устройство ввода/вывода).

Внутренние *регистры* любого микропроцессора обязательно выполняют две служебные функции:

* определяют адрес в памяти, где находится выполняемая в данный момент команда (функция **счетчика команд** или **указателя команд** );
* определяют текущий адрес *стека* (функция указателя *стека* ).

В разных процессорах для каждой из этих функций может отводиться один или два внутренних *регистра*. Эти два указателя отличаются от других не только своим специфическим, служебным, системным назначением, но и особым способом изменения содержимого. Их содержимое программы могут менять только в случае крайней необходимости, так как любая ошибка при этом грозит нарушением работы компьютера, зависанием и порчей содержимого памяти.

Содержимое указателя (*счетчика) команд* изменяется следующим образом. В начале работы системы (при включении питания) в него заносится раз и навсегда установленное значение. Это первый адрес программы начального запуска. Затем после выборки из памяти каждой следующей команды значение указателя команд автоматически увеличивается (инкрементируется) на единицу (или на два в зависимости от *формата команд* и типа процессора). То есть следующая команда будет выбираться из следующего по порядку адреса памяти. При выполнении команд перехода, нарушающих *последовательный перебор* адресов памяти, в *указатель команд* принудительно записывается новое значение — новый адрес в памяти, начиная с которого адреса команд опять же будут перебираться последовательно. Такая же смена содержимого указателя команд производится при вызове подпрограммы и возврате из нее или при начале обработки прерывания и после его окончания.

О *стеке* будет подробнее рассказано в следующем разделе.

#### 3.1.2. Функции памяти

Память *микропроцессорной системы* выполняет функцию временного или постоянного хранения данных и команд. Объем памяти определяет допустимую сложность выполняемых системой алгоритмов, а также в некоторой степени и скорость работы системы в целом. Модули памяти выполняются на микросхемах памяти (оперативной или постоянной). Все чаще в составе микропроцессорных систем используется *флэш-память* (англ. — *flash memory*), которая представляет собой энергонезависимую память с возможностью многократной перезаписи содержимого.

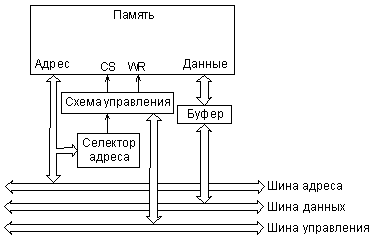
Информация в памяти хранится в ячейках, количество разрядов которых равно количеству разрядов *шины данных* процессора. Обычно оно кратно восьми (например, 8, 16, 32, 64). Допустимое количество ячеек памяти определяется количеством разрядов *шины адреса* как 2N, где N — количество разрядов *шины адреса*. Чаще всего объем памяти измеряется в байтах независимо от разрядности ячейки памяти. Используются также следующие более крупные единицы объема памяти: килобайт — 210 или 1024 байта (обозначается Кбайт), мегабайт — 220 или 1 048 576 байт (обозначается Мбайт), гигабайт — 230 байт (обозначается Гбайт), терабайт — 240 (обозначается Тбайт) Например, если память имеет 65 536 ячеек, каждая из которых 16-разрядная, то говорят, что память имеет объем 128 Кбайт. Совокупность ячеек памяти называется обычно **пространством памяти** системы.

Для *подключения модуля* памяти к системной магистрали используются блоки сопряжения, которые включают в себя *дешифратор* *(селектор) адреса*, схему обработки управляющих сигналов магистрали и *буферы* данных (рис. 3.3).

Оперативная память общается с системной магистралью в циклах чтения и записи, *постоянная память* — только в циклах чтения. Обычно в составе системы имеется несколько модулей памяти, каждый из которых работает в своей области пространства памяти. *Селектор адреса* как раз и определяет, какая область адресов пространства памяти отведена данному модулю памяти. Схема управления вырабатывает в нужные моменты сигналы разрешения работы памяти (CS) и сигналы разрешения записи в память (WR). *Буферы* данных передают данные от памяти к магистрали или от магистрали к памяти.

В пространстве памяти *микропроцессорной системы* обычно выделяются несколько особых областей, которые выполняют специальные функции.

**Память программы начального запуска** всегда выполняется на *ПЗУ* или флэш-памяти. Именно с этой области процессор начинает работу после включения питания и после сброса его с помощью сигнала RESET.



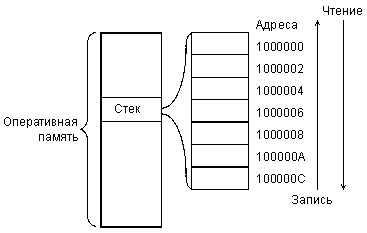
**Рис. 3.3.**Структура модуля памяти.

Память для *стека* или ***стек*** (Stack) — это часть оперативной памяти, предназначенная для временного хранения данных в режиме *LIFO* (Last In — First Out).

Особенность *стека* по сравнению с другой оперативной памятью — это заданный и неизменяемый *способ адресации*. При записи любого числа (кода) в *стек* число записывается по адресу, определяемому как содержимое *регистра* указателя *стека*, предварительно уменьшенное (декрементированное) на единицу (или на два, если 16-разрядные слова расположены в памяти по четным адресам). При чтении из *стека* число читается из адреса, определяемого содержимым указателя *стека*, после чего это содержимое указателя *стека* увеличивается (инкрементируется) на единицу (или на два). В результате получается, что число, записанное последним, будет прочитано первым, а число, записанное первым, будет прочитано последним. Такая память называется *LIFO* или памятью магазинного типа (например, в магазине автомата патрон, установленный последним, будет извлечен первым).

Принцип действия *стека* показан на рис. 3.4 (адреса ячеек памяти выбраны условно).

Пусть, например, текущее состояние указателя *стека* 1000008, и в него надо записать два числа (слова). Первое слово будет записано по адресу 1000006 (перед записью указатель *стека* уменьшится на два). Второе — по адресу 1000004. После записи содержимое указателя *стека* — 1000004. Если затем прочитать из *стека*два слова, то первым будет прочитано слово из адреса 1000004, а после чтения указатель стека *станет* равным 1000006. Вторым будет прочитано слово из адреса 1000006, а указатель *стека* станет равным 1000008. Все вернулось к исходному состоянию. Первое записанное слово читается вторым, а второе — первым.



**Рис. 3.4.**Принцип работы стека.

Необходимость такой адресации становится очевидной в случае многократно вложенных подпрограмм. Пусть, например, выполняется основная программа, и из нее вызывается подпрограмма 1. Если нам надо сохранить значения данных и внутренних *регистров* основной программы на время выполнения подпрограммы, мы перед вызовом подпрограммы сохраним их в *стеке* (запишем в *стек* ), а после ее окончания извлечем (прочитаем) их из *стека*. Если же из подпрограммы 1 вызывается подпрограмма 2, то ту же самую операцию мы проделаем с данными и содержимым внутренних *регистров* подпрограммы 1. Понятно, что внутри подпрограммы 2 крайними в *стеке* (читаемыми в первую очередь) будут данные из подпрограммы 1, а данные из основной программы будут глубже. При этом в случае чтения из *стека* автоматически будет соблюдаться нужный порядок читаемой информации. То же самое будет и в случае, когда таких уровней вложения подпрограмм гораздо больше. То есть то, что надо хранить подольше, прячется поглубже, а то, что скоро может потребоваться — с краю.

В системе команд любого процессора для обмена информацией со *стеком* предусмотрены специальные команды записи в *стек* (PUSH) и чтения из *стека* (POP). В *стеке* можно прятать не только содержимое всех внутренних *регистров* процессоров, но и содержимое *регистра* признаков (слово состояния процессора, *PSW*). Это позволяет, например, при возвращении из подпрограммы контролировать результат последней команды, выполненной непосредственно перед вызовом этой подпрограммы. Можно также хранить в *стеке* и данные, для того чтобы удобнее было передавать их между программами и подпрограммами. В общем случае, чем больше область памяти, отведенная под *стек*, тем больше свободы у программиста и тем более сложные программы могут выполняться.

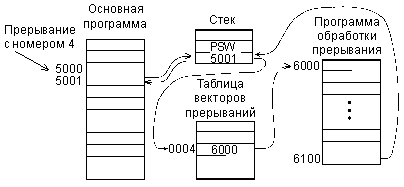
Следующая специальная область памяти — это **таблица векторов прерываний**.

Вообще, понятие прерывания довольно многозначно. Под прерыванием в общем случае понимается не только обслуживание запроса внешнего устройства, но и любое нарушение последовательной работы процессора. Например, может быть предусмотрено прерывание по факту некорректного выполнения арифметической операции типа деления на ноль. Или же прерывание может быть программным, когда в программе используется команда перехода на какую-то подпрограмму, из которой затем последует возврат в основную программу. В последнем случае общее с истинным прерыванием только то, как осуществляется переход на подпрограмму и возврат из нее.

Любое прерывание обрабатывается через таблицу векторов (указателей) прерываний. В этой таблице в простейшем случае находятся адреса начала программ обработки прерываний, которые и называются векторами. Длина таблицы может быть довольно большой (до нескольких сот элементов). Обычно *таблица векторов прерываний* располагается в начале пространства памяти (в ячейках памяти с малыми адресами). Адрес каждого вектора (или адрес начального элемента каждого вектора) представляет собой номер прерывания.

В случае аппаратных прерываний номер прерывания или задается устройством, запросившим прерывание (при векторных прерываниях), или же задается номером линии запроса прерываний (при радиальных прерываниях). Процессор, получив аппаратное прерывание, заканчивает выполнение текущей команды и обращается к памяти в область таблицы векторов прерываний, в ту ее строку, которая определяется номером запрошенного прерывания. Затем процессор читает содержимое этой строки (код вектора прерывания) и переходит в адрес памяти, задаваемый этим вектором. Начиная с этого адреса в памяти должна располагаться программа обработки прерывания с данным номером. В конце программы обработки прерываний обязательно должна располагаться команда выхода из прерывания, выполнив которую, процессор возвращается к выполнению прерванной основной программы. Параметры процессора на время выполнения программы обработки прерывания сохраняются в *стеке*.

Пусть, например, процессор (рис.3.5) выполнял основную программу и команду, находящуюся в адресе памяти 5000 (условно). В этот момент он получил *запрос прерывания* с номером (адресом вектора) 4. Процессор заканчивает выполнение команды из адреса 5000. Затем он сохраняет в *стеке* текущее значение счетчика команд (5001) и текущее значение *PSW*. После этого процессор читает из адреса 4 памяти код вектора прерывания. Пусть этот код равен 6000. Процессор переходит в адрес памяти 6000 и приступает к выполнению программы обработки прерывания, начинающейся с этого адреса. Пусть эта программа заканчивается в адресе 6100. Дойдя до этого адреса, процессор возвращается к выполнению прерванной программы. Для этого он извлекает из *стека* значение адреса (5001), на котором его прервали, и бывшее в тот момент *PSW*. Затем процессор читает команду из адреса 5001 и дальше последовательно выполняет команды основной программы.



**Рис. 3.5.**Упрощенный алгоритм обработки прерывания.

Прерывание в случае аварийной ситуации обрабатывается точно так же, только адрес вектора прерывания (номер строки в таблице векторов) жестко привязан к данному типу аварийной ситуации.

Программное прерывание тоже обслуживается через таблицу векторов прерываний, но номер прерывания указывается в составе команды, вызывающей прерывание.

Такая сложная, на первый взгляд, организация прерываний позволяет программисту легко менять программы обработки прерываний, располагать их в любой области памяти, делать их любого размера и любой сложности.

Во время выполнения программы обработки прерывания может поступить новый запрос на прерывание. В этом случае он обрабатывается точно так же, как описано, но основной программой считается прерванная программа обработки предыдущего прерывания. Это называется многократным вложением прерываний. Механизм *стека* позволяет без проблем обслуживать это многократное вложение, так как первым из *стека* извлекается тот код, который был сохранен последним, то есть возврат из обработки данного прерывания происходит в программу обработки предыдущего прерывания.

Отметим, что в более сложных случаях в таблице векторов прерываний могут находиться не адреса начала программ обработки прерываний, а так называемые дескрипторы (описатели) прерываний. Но конечным результатом обработки этого дескриптора все равно будет адрес начала программы обработки прерываний.

Наконец, еще одна специальная область памяти *микропроцессорной системы* — это **память устройств, подключенных к системной шине**. Такое решение встречается нечасто, но иногда оно очень удобно. То есть процессор получает возможность обращаться к внутренней памяти устройств ввода/вывода или каких-то еще подключенных к системной шине устройств, как к своей собственной системной памяти. Обычно окно в пространстве памяти, выделяемое для этого, не слишком большое.

Все остальные части пространства памяти, как правило, имеют универсальное назначение. В них могут располагаться как данные, так и программы (конечно, в случае одношинной архитектуры). Иногда это пространство памяти используется как единое целое, без всяких границ. А иногда пространство памяти делится на сегменты с программно изменяемым адресом начала сегмента и с установленным размером сегмента. Оба подхода имеют свои плюсы и минусы. Например, использование сегментов позволяет защитить область программ или данных, но зато границы сегментов могут затруднять размещение больших программ и массивов данных.

В заключение остановимся на проблеме разделения адресов памяти и адресов устройств ввода/вывода. Существует два основных подхода к решению этой проблемы:

* выделение в общем адресном пространстве системы специальной области адресов для устройств ввода/вывода;
* полное разделение адресных пространств памяти и устройств ввода/вывода.

Первый подход хорош тем, что при обращении к устройствам ввода/вывода процессор может использовать те же команды, которые служат для взаимодействия с памятью. Но адресное пространство памяти должно быть уменьшено на величину адресного пространства устройств ввода/вывода. Например, при 16-разрядной шине адреса всего может быть 64К адресов. Из них 56К адресов отводится под адресное пространство памяти, а 8К адресов — под адресное пространство устройств ввода/вывода.

Преимущество второго подхода состоит в том, что память занимает все адресное пространство *микропроцессорной системы*. Для общения с устройствами ввода/вывода применяются специальные команды и специальные стробы обмена на магистрали. Именно так сделано, например, в персональных компьютерах. Но возможности взаимодействия с устройствами ввода/вывода в данном случае существенно ограничены по сравнению с возможностями общения с памятью.

#### 3.1.3. Функции устройств ввода/вывода

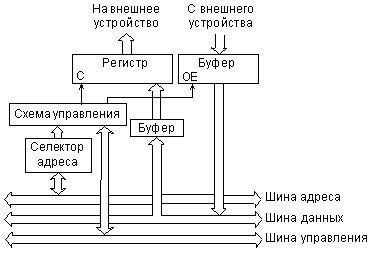
Устройства ввода/вывода обмениваются информацией с магистралью по тем же принципам, что и память. Наиболее существенное отличие с точки зрения организации обмена состоит в том, что модуль памяти имеет в адресном пространстве системы много адресов (до нескольких десятков миллионов), а устройство ввода/вывода обычно имеет немного адресов (обычно до десяти), а иногда и всего один адрес.

Но модули памяти системы обмениваются информацией только с магистралью, с процессором, а устройства ввода/вывода взаимодействуют еще и с внешними устройствами, цифровыми или аналоговыми. Поэтому разнообразие устройств ввода/вывода неизмеримо больше, чем модулей памяти. Часто используются еще и другие названия для устройств ввода/вывода: устройства сопряжения, контроллеры, карты расширения, интерфейсные модули и т.д.

Объединяют все устройства ввода/вывода общие принципы обмена с магистралью и, соответственно, общие принципы организации узлов, которые осуществляют сопряжение с магистралью. Упрощенная структура устройства ввода/вывода (точнее, его интерфейсной части) приведена на [рис. 3.](http://www.intuit.ru/studies/courses/3/3/lecture/66?page=3#image.2.21)6. Как и в случае модуля памяти, она обязательно содержит схему *селектора адреса*, схему управления для обработки стробов обмена и *буферы* данных.

Самые простейшие устройства ввода/вывода выдают на внешнее устройство код данных в параллельном формате и принимают из внешнего устройства код данных в параллельном формате. Такие устройства ввода/вывода часто называют параллельными портами ввода/вывода. Они наиболее универсальны, то есть удовлетворяют потребности сопряжения с большим числом внешних устройств, поэтому их часто вводят в состав *микропроцессорной системы* в качестве стандартных устройств. Параллельные порты обычно имеются в составе микроконтроллеров. Именно через параллельные порты микроконтроллер связывается с внешним миром.

Входной порт (порт ввода) в простейшем случае представляет собой параллельный *регистр*, в который процессор может записывать информацию. Выходной порт (порт вывода) обычно представляет собой просто однонаправленный *буфер*, через который процессор может читать информацию от внешнего устройства. Именно такие порты показаны для примера на рис. 3.6. Порт может быть и двунаправленным (входным/выходным). В этом случае процессор пишет информацию во внешнее устройство и читает информацию из внешнего устройства по одному и тому же адресу в адресном пространстве системы. Входные и выходные линии для связи с внешним устройством при этом могут быть объединены поразрядно, образуя двунаправленные линии.



**Рис. 3.6.**Структура простейшего устройства ввода/вывода.

При обращении со стороны магистрали *селектор адреса* распознает адрес, приписанный данному устройству ввода/вывода. Схема управления выдает внутренние стробы обмена в ответ на магистральные стробы обмена. Входной *буфер* данных обеспечивает электрическое согласование *шины данных* с этим устройством (*буфер* может и отсутствовать). Данные из *шины данных* записываются в *регистр* по сигналу С и выдаются на внешнее устройство. Выходной *буфер* данных передает входные данные с внешнего устройства на шину данных магистрали в цикле чтения из порта.

Более сложные устройства ввода/вывода (устройства сопряжения) имеют в своем составе внутреннюю буферную оперативную память и даже могут иметь микроконтроллер, на который возложено выполнение функций обмена с внешним устройством.

Каждому устройству ввода/вывода отводится свой адрес в адресном пространстве *микропроцессорной системы*. Дублирование адресов должно быть исключено, за этим должны следить разработчик и пользователь *микропроцессорной системы*.

Устройства ввода/вывода помимо программного обмена могут также поддерживать режим обмена по прерываниям. В этом случае они преобразуют поступающий от внешнего устройства сигнал запроса на прерывание в сигнал запроса прерывания, необходимый для данной магистрали (или в последовательность сигналов при векторном прерывании). Если нужно использовать режим *ПДП*, устройство ввода/вывода должно выдать сигнал запроса *ПДП* на магистраль и обеспечить работу в циклах *ПДП*, принятых для данной магистрали.

В составе микропроцессорных систем, как правило, выделяются три специальные группы устройств ввода/вывода:

* устройства ***интерфейса*** пользователя (ввода информации пользователем и вывода информации для пользователя);
* устройства ввода/вывода для длительного хранения информации;
* таймерные устройства.

К устройствам ввода для ***интерфейса*** пользователя относятся контроллеры клавиатуры, тумблеров, отдельных кнопок, мыши, трекбола, джойстика и т.д.   К устройствам вывода для *интерфейса* пользователя относятся контроллеры светодиодных индикаторов, табло, жидкокристаллических, плазменных и электронно-лучевых экранов и т.д. В простейших случаях управляющих контроллеров или микроконтроллеров эти средства могут отсутствовать. В сложных микропроцессорных системах они есть обязательно. Роль внешнего устройства в данном случае играет человек.

Устройства ввода/вывода для длительного хранения информации обеспечивают сопряжение *микропроцессорной системы* с дисководами (компакт-дисков или магнитных дисков), а также с накопителями на магнитной ленте. Применение таких устройств существенно увеличивает возможности *микропроцессорной системы*в отношении хранения выполняемых программ и накопления массивов данных. В простейших контроллерах эти устройства отсутствуют.

Таймерные устройства отличаются от других устройств ввода/вывода тем, что они могут не иметь внешних выводов для подключения к внешним устройствам. Эти устройства предназначены для того, чтобы микропроцессорная система могла выдерживать заданные временные интервалы, следить за реальным временем, считать импульсы и т.д. В основе любого таймера лежит кварцевый тактовый генератор и многоразрядные двоичные счетчики, которые могут перезапускать друг друга. Процессор может записывать в таймер коэффициенты деления тактовой частоты, количество отсчитываемых импульсов, задавать режим работы счетчиков таймера, а читает процессор выходные коды счетчиков. В принципе выполнить практически все функции таймера можно и программным путем, поэтому иногда таймеры в системе отсутствуют. Но включение в систему таймера позволяет решать более сложные задачи и строить более эффективные алгоритмы.

Еще один важный класс устройств ввода/вывода — это устройства для подключения к информационным сетям (локальным и глобальным). Эти устройства распространены не так широко, как устройства трех перечисленных ранее групп, но их значение с каждым годом становится все больше. Сейчас средства связи с информационными сетями вводятся иногда даже в простые контроллеры.

Иногда устройства ввода/вывода обеспечивают сопряжение с внешними устройствами с помощью аналоговых сигналов. Это бывает очень удобно, поэтому в состав некоторых микроконтроллеров даже вводят внутренние ЦАП и АЦП.

**Лекция 4. Адресация операндов**

Основная *функция* любого процессора, ради которой он и создается, — это выполнение команд. *Система команд*, выполняемых процессором, представляет собой нечто подобное *таблице истинности* логических элементов или таблице режимов работы более сложных логических микросхем. То есть она определяет логику работы процессора и его реакцию на те или иные комбинации внешних событий.

Написание программ для *микропроцессорной системы* — важнейший и часто наиболее трудоемкий этап разработки такой системы. А для создания эффективных программ необходимо иметь хотя бы самое общее *представление* о системе команд используемого процессора. Самые компактные и быстрые программы и подпрограммы создаются на языке *Ассемблер*, использование которого без знания системы команд абсолютно невозможно, ведь язык *Ассемблер* представляет собой символьную *запись* цифровых кодов *машинного языка*, кодов команд процессора. Конечно, для разработки программного обеспечения существуют всевозможные *программные средства*. Пользоваться ими обычно можно и без знания системы команд процессора. Чаще всего применяются языки программирования высокого уровня, такие как *Паскаль* и *Си*. Однако *знание* системы команд и языка *Ассемблер* позволяет в несколько раз повысить эффективность некоторых наиболее важных частей программного обеспечения любой *микропроцессорной системы* — от микроконтроллера до персонального компьютера.

Именно поэтому в данной главе мы рассмотрим основные *типы команд*, имеющиеся у большинства процессоров, и особенности их применения.

Каждая *команда*, выбираемая (читаемая) из памяти процессором, определяет *алгоритм* поведения процессора на ближайшие несколько тактов. Код команды говорит о том, какую операцию предстоит выполнить процессору и с какими ***операндами*** (то есть кодами данных), где взять исходную информацию для выполнения команды и куда поместить результат (если необходимо). Код команды может занимать от одного до нескольких *байт*, причем *процессор* узнает о том, сколько *байт* команды ему надо читать, из первого прочитанного им байта или слова. В процессоре код команды расшифровывается и преобразуется в набор*микроопераций*, выполняемых отдельными узлами процессора. Но разработчику микропроцессорных систем это *знание* не слишком важно, ему важен только результат выполнения той или иной команды.

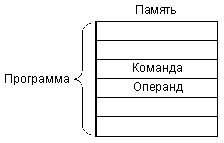
### 4.1. Адресация операндов

Большая часть команд процессора работает с кодами данных ( *операндами* ). Одни команды требуют входных *операндов* (одного или двух), другие выдают выходные *операнды* (чаще один *операнд* ). Входные *операнды* называются еще операндами-источниками, а выходные называются операндами-приемниками. Все эти коды *операндов* (входные и выходные) должны где-то располагаться. Они могут находиться во внутренних регистрах процессора (наиболее удобный и быстрый вариант). Они могут располагаться в системной памяти (самый распространенный вариант). Наконец, они могут находиться в устройствах ввода/вывода (наиболее редкий случай). *Определение* места положения *операндов* производится кодом команды. Причем существуют разные методы, с помощью которых код команды может определить, откуда брать входной *операнд* и куда помещать выходной *операнд*. Эти методы называются ***методами адресации*** . Эффективность выбранных*методов адресации* во многом определяет эффективность работы всего процессора в целом.

#### 4.1.1. Методы адресации

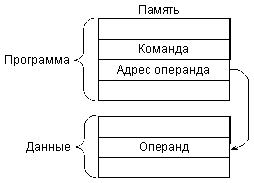
Количество *методов адресации* в различных процессорах может быть от 4 до 16. Рассмотрим несколько типичных *методов адресации операндов*, используемых сейчас в большинстве микропроцессоров.

**Непосредственная адресация** (рис.4.1) предполагает, что *операнд* (входной) находится в памяти непосредственно за кодом команды. *Операнд* обычно представляет собой константу, которую надо куда-то переслать, к чему-то прибавить и т.д. Например, команда может состоять в том, чтобы прибавить число 6 к содержимому какого-то внутреннего регистра процессора. Это число 6 будет располагаться в памяти, внутри программы в адресе, следующем за кодом данной команды сложения.



**Рис. 4.1.**Непосредственная адресация.

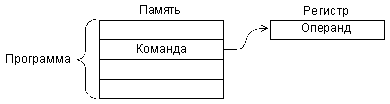
**Прямая (она же абсолютная) адресация** (рис. 4.2) предполагает, что *операнд* (входной или выходной) находится в памяти по адресу, код которого находится внутри программы сразу же за кодом команды. Например, команда может состоять в том, чтобы очистить (сделать нулевым) содержимое ячейки памяти с адресом 1000000. Код этого адреса 1000000 будет располагаться в памяти, внутри программы в следующем адресе за кодом данной команды очистки.



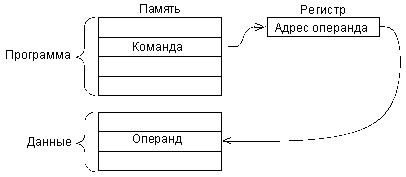
**Рис. 4.2.**Прямая адресация.

**Регистровая адресация** (рис. 4.3) предполагает, что *операнд* (входной или выходной) находится во внутреннем регистре процессора. Например, команда может состоять в том, чтобы переслать число из нулевого регистра в первый. Номера обоих регистров (0 и 1) будут определяться кодом *команды пересылки*.

**Косвенно-регистровая (она же косвенная) адресация** предполагает, что во внутреннем регистре процессора находится не сам *операнд*, а его адрес в памяти (рис. 4.4). Например, команда может состоять в том, чтобы очистить ячейку памяти с адресом, находящимся в нулевом регистре. Номер этого регистра (0) будет определяться кодом команды очистки.



**Рис. 4.3.**Регистровая адресация.



**Рис. 4.4.**Косвенная адресация.

Реже встречаются еще два *метода адресации*.

**Автоинкрементная адресация** очень близка к косвенной адресации, но отличается от нее тем, что после выполнения команды содержимое используемого регистра увеличивается на единицу или на два. Этот *метод адресации* очень удобен, например, при последовательной обработке кодов из массива данных, находящегося в памяти. После обработки какого-то кода адрес в регистре будет указывать уже на следующий код из массива. При использовании косвенной адресации в данном случае пришлось бы увеличивать содержимое этого регистра отдельной командой.

**Автодекрементная адресация** работает похоже на автоинкрементную, но только содержимое выбранного регистра уменьшается на единицу или на два перед выполнением команды. Эта адресация также удобна при обработке массивов данных. Совместное использование автоинкрементной и автодекрементной адресаций позволяет организовать память стекового типа.

Из других распространенных *методов адресации* можно упомянуть об индексных методах, которые предполагают для вычисления адреса *операнда* прибавление к содержимому регистра заданной константы (индекса). Код этой константы располагается в памяти непосредственно за кодом команды.

Отметим, что выбор того или иного *метода адресации* в значительной степени определяет время выполнения команды. Самая быстрая адресация — это регистровая, так как она не требует дополнительных циклов обмена по магистрали. Если же адресация требует обращения к памяти, то время выполнения команды будет увеличиваться за счет длительности необходимых циклов обращения к памяти. Понятно, что чем больше внутренних регистров у процессора, тем чаще и свободнее можно применять регистровую адресацию, и тем быстрее будет работать система в целом.

#### 4.1.2. Сегментирование памяти

Говоря об адресации, нельзя обойти вопрос о *сегментировании* памяти, применяемой в некоторых процессорах, например в процессорах IBM PC-совместимых персональных компьютеров.

В процессоре Intel 8086 *сегментирование* памяти организовано следующим образом.

Вся память системы представляется не в виде непрерывного пространства, а в виде нескольких кусков — сегментов заданного размера (по 64 Кбайта), положение которых в пространстве памяти можно изменять программным путем.

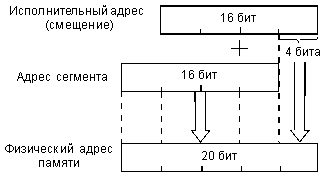
Для хранения кодов адресов памяти используются не отдельные регистры, а пары регистров:

* сегментный регистр определяет адрес начала сегмента (то есть положение сегмента в памяти);
* ***регистр указателя*** (регистр смещения) определяет положение рабочего адреса внутри сегмента.

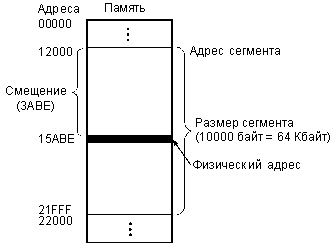
При этом физический 20-разрядный адрес памяти, выставляемый на внешнюю *шину адреса*, образуется так, как показано на рис. 4.5, то есть путем сложения смещения и адреса сегмента со сдвигом на 4 бита. Положение этого адреса в памяти показано на рис. 4.6.

Сегмент может начинаться только на 16-байтной границе памяти (так как адрес начала сегмента, по сути, имеет четыре младших нулевых разряда, как видно из рис. 4.5), то есть с адреса, кратного 16. Эти допустимые границы сегментов называются границами параграфов.

Отметим, что введение *сегментирования*, прежде всего, связано с тем, что внутренние регистры процессора 16-разрядные, а физический адрес памяти 20-разрядный (16-разрядный адрес позволяет использовать память только в 64 Кбайт, что явно недостаточно). В появившемся в то же время процессоре MC68000 фирмы Motorola внутренние регистры 32-разрядные, поэтому там проблемы *сегментирования* памяти не возникает.



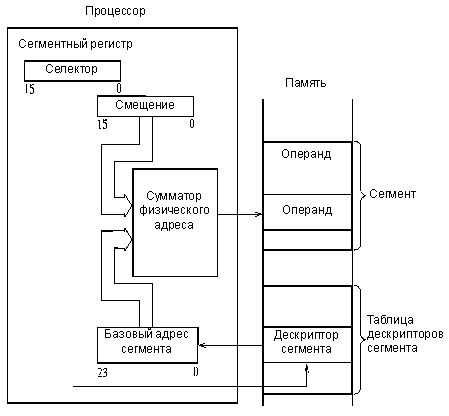
**Рис. 4.5.**Формирование физического адреса памяти из адреса сегмента и смещения.



**Рис. 4.6.**Физический адрес в сегменте (все коды — шестнадцатеричные).

Применяются и более сложные методы *сегментирования* памяти. Например, в процессоре *Intel 80286* в так называемом защищенном режиме адрес памяти вычисляется в соответствии с рис. 4.7.

В *сегментном регистре* в данном случае хранится не базовый (начальный) адрес сегментов, а коды селекторов, определяющие адреса в памяти, по которым хранятся дескрипторы (то есть описатели) сегментов. Область памяти с дескрипторами называется таблицей дескрипторов. Каждый дескриптор сегмента содержит базовый адрес сегмента, размер сегмента (от 1 до 64 Кбайт) и его атрибуты. Базовый адрес сегмента имеет разрядность 24 бит, что обеспечивает адресацию 16 Мбайт физической памяти.

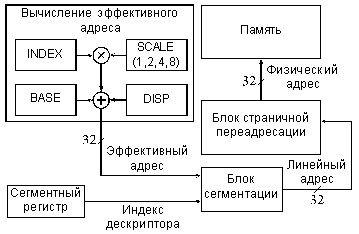


**Рис. 4.7.**Адресация памяти в защищенном режиме процессора Intel 80286.

Таким образом, на *сумматор*, вычисляющий физический адрес памяти, подается не содержимое сегментного регистра, как в предыдущем случае, а базовый адрес сегмента из таблицы дескрипторов.

Еще более сложный *метод адресации* памяти с *сегментированием* использован в процессоре *Intel 80386* и в более поздних моделях процессоров фирмы Intel. Этот метод иллюстрируется рис. 4.8.

Адрес памяти (физический адрес) вычисляется в три этапа. Сначала вычисляется так называемый **эффективный адрес** (32-разрядный) путем суммирования трех компонентов: базы, индекса и смещения (Base, Index, *Displacement*), причем возможно умножение индекса на масштаб (Scale). Эти компоненты имеют следующий смысл:



**Рис. 4.8.**Формирование физического адреса памяти процессора 80386 в защищенном режиме.

* смещение — это 8-, 16- или 32-разрядное число, включенное в команду.
* база — это содержимое базового регистра процессора. Обычно оно используется для указания на начало некоторого массива.
* индекс — это содержимое индексного регистра процессора. Обычно оно используется для выбора одного из элементов массива.
* масштаб — это множитель (он может быть равен 1, 2, 4 или 8), указанный в коде команды, на который перед суммированием с другими компонентами умножается индекс. Он используется для указания размера элемента массива.

Затем специальный блок *сегментации* вычисляет 32-разрядный линейный адрес, который представляет собой сумму базового адреса сегмента из сегментного регистра с эффективным адресом. Наконец, физический 32-битный адрес памяти образуется путем преобразования линейного адреса блоком страничной переадресации, который осуществляет перевод линейного адреса в *физический страницами* по 4 Кбайта.

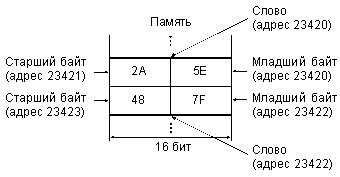
В любом случае *сегментирование* позволяет выделить в памяти один или несколько сегментов для данных и один или несколько сегментов для программ. Переход от одного сегмента к другому сводится всего лишь к изменению содержимого сегментного регистра. Иногда это бывает очень удобно. Но для программиста работать с сегментированной памятью обычно сложнее, чем с непрерывной, несегментированной памятью, так как приходится следить за границами сегментов, за их описанием, переключением и т.д.

#### 4.1.3. Адресация байтов и слов

Многие процессоры, имеющие разрядность 16 или 32, способны адресовать не только целое слово в памяти (16-разрядное или 32-разрядное), но и отдельные байты. Каждому байту в каждом слове при этом отводится свой адрес.

Так, в случае 16-разрядных процессоров все слова в памяти (16-разрядные) имеют четные адреса. А байты, входящие в эти слова, могут иметь как четные адреса, так и нечетные.

Например, пусть 16-разрядная ячейка памяти имеет адрес 23420, и в ней хранится код 2А5Е (рис. 4.9).



**Рис.4.9.**Адресация слов и байтов.

При обращении к целому слову (с содержимым 2А5Е ) процессор выставляет адрес 23420. При обращении к младшему байту этой ячейки (с содержимым 5Е ) процессор выставляет тот же самый адрес 23420, но использует команду, адресующую байт, а не слово. При обращении к старшему байту этой же ячейки (с содержимым 2А ) процессор выставляет адрес 23421 и использует команду, адресующую байт. Следующая по порядку 16-разрядная ячейка памяти с содержимым 487F будет иметь адрес 23422, то есть опять же четный. Ее байты будут иметь адреса 23422 и 23423.

Для различия байтовых и словных циклов обмена на магистрали в *шине управления* предусматривается специальный сигнал байтового обмена. Для работы с байтами в систему команд процессора вводятся специальные команды или предусматриваются методы байтовой адресации.

### 4.2. Регистры процессора

Как уже упоминалось, внутренние регистры процессора представляют собой сверхоперативную *память* небольшого размера, которая предназначена для временного хранения служебной информации или данных. Количество регистров в разных процессорах может быть от 6—8 до нескольких десятков. Регистры могут быть универсальными и специализированными. Специализированные регистры, которые присутствуют в большинстве процессоров, — это *регистр*- *счетчик команд*, *регистр состояния* ( ***PSW*** ), *регистр указателя* стека. Остальные регистры процессора могут быть как универсальными, так и специализированными.

Например, в 16-разрядном процессоре Т-11 фирмы *DEC* было 8 регистров общего назначения (РОН) и один *регистр состояния*. Все регистры имели по 16 разрядов. Из регистров общего назначения один отводился под *счетчик команд*, другой — под *указатель стека*. Все остальные регистры общего назначения полностью взаимозаменяемы, то есть имеют универсальное назначение, могут хранить как данные, так и адреса (указатели), индексы и т.д. Максимально допустимый объем памяти для данного процессора составлял 64 Кбайт (*адрес* памяти 16-разрядный).

В 16-разрядном процессоре MC68000 фирмы Motorola было 19 регистров: 16-разрядный *регистр состояния*, 32-разрядный *регистр* *счетчика команд*, 9 регистров адреса (32-разрядных) и 8 регистров данных (32-разрядных). Два регистра адреса отведены под указатели стека. Максимально допустимый объем адресуемой памяти — 16 Мбайт (внешняя *шина адреса* 24-разрядная). Все 8 регистров данных взаимозаменяемы. 7 регистров адреса – тоже взаимозаменяемы.

В 16-разрядном процессоре Intel 8086, который стал базовым в линии процессоров, используемых в персональных компьютерах, реализован принципиально другой подход. Каждый *регистр* этого процессора имеет свое особое назначение, и заменять друг друга регистры могут только частично или же не могут вообще. Остановимся на особенностях этого процессора подробнее.

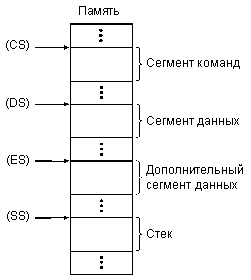
*Процессор* 8086 имеет 14 регистров разрядностью по 16 *бит*. Из них четыре регистра ( AX, BX, CX, DX ) — это регистры данных, каждый из которых помимо хранения *операндов* и результатов операций имеет еще и свое специфическое назначение:

* регистр AX — умножение, деление, обмен с устройствами ввода/вывода (команды ввода и вывода);
* регистр BX — базовый регистр в вычислениях адреса;
* регистр CX — счетчик циклов;
* регистр DX — определение адреса ввода/вывода.

Для регистров данных существует возможность раздельного использования обоих байтов (например, для регистра AX они имеют обозначения AL – младший *байт* и AH — старший *байт*).

Следующие четыре внутренних регистра процессора — это сегментные регистры, каждый из которых определяет положение одного из рабочих сегментов (рис. 4.10):

* регистр CS (Code Segment) соответствует сегменту команд, исполняемых в данный момент;
* регистр DS (Data Segment) соответствует сегменту данных, с которыми работает процессор;
* регистр ES (Extra Segment) соответствует дополнительному сегменту данных;
* регистр SS (Stack Segment) соответствует сегменту стека.



**Рис. 4.10.**Сегменты команд, данных и стека в памяти.

В принципе, все эти *сегменты* могут и перекрываться для оптимального использования пространства памяти. Например, если *программа* занимает только часть сегмента, то сегмент данных может начинаться сразу после завершения работы программы (с точностью 16 *байт*), а не после окончания всего сегмента программы.

Следующие пять регистров процессора ( SP — *Stack Pointer*, BP — *Base* *Pointer*, SI — Source *Index*, DI — *Destination* *Index*, IP —Instruction *Pointer*) служат указателями (то есть определяют смещение в пределах сегмента). Например, *счетчик команд* процессора образуется парой регистров CS и IP, а *указатель стека* — парой регистров SP и SS. Регистры SI, DI используются в строковых операциях, то есть при последовательной обработке нескольких ячеек памяти одной командой.

Последний *регистр* FLAGS — это *регистр состояния* процессора ( *PSW* ). Из его 16 разрядов используются только девять ([рис. 3.11](http://www.intuit.ru/studies/courses/3/3/lecture/68?page=2#image.3.11)): CF (*Carry* *Flag*) — флаг переноса при арифметических операциях, PF (*Parity Flag*) — флаг четности результата, AF (*Auxiliary* *Flag*) — флаг дополнительного переноса, ZF (Zero *Flag*) — флаг нулевого результата, SF (*Sign* *Flag*) — флаг знака (совпадает со старшим битом результата), TF (*Trap* *Flag*) — флаг пошагового режима (используется при отладке), IF (Interrupt-enable *Flag*) — флаг разрешения аппаратных прерываний, DF (*Direction* *Flag*) — флаг направления при строковых операциях, OF(*Overflow* *Flag*) — флаг переполнения.

Регистр состояния процессора 8086.

**Рис. 4.11.**Регистр состояния процессора 8086.

Биты регистра состояния устанавливаются или очищаются в зависимости от результата исполнения предыдущей команды и используются некоторыми командами процессора. Биты регистра состояния могут также устанавливаться и очищаться специальными командами процессора (о системе команд процессора будет рассказано в следующем разделе).

Во многих процессорах выделяется специальный *регистр*, называемый **аккумулятором** (то есть накопителем). При этом, как правило, только этот *регистр*-аккумулятор может участвовать во всех операциях, только через него может производиться взаимодействие с устройствами ввода/вывода. Иногда в него же помещается результат любой выполненной команды (в этом случае говорят даже об "аккумуляторной" архитектуре процессора). Например, в процессоре 8086 *регистр* данных АХ можно считать своеобразным аккумулятором, так как именно он обязательно участвует в командах умножения и деления, а также только через него можно пересылать данные в *устройство ввода/вывода* и из устройства ввода/вывода. Выделение специального регистра-аккумулятора упрощает структуру процессора и ускоряет пересылки кодов внутри процессора, но в некоторых случаях замедляет работу системы в целом, так как весь *поток* информации должен пройти через один *регистр*-аккумулятор. В случае, когда несколько регистров процессора полностью взаимозаменяемы, таких проблем не возникает.

**Лекция 5. Система команд процессора**

### 5.1.Система команд процессора

В общем случае *система команд* процессора включает в себя следующие четыре основные группы команд:

* *команды пересылки* данных;
* *арифметические команды* ;
* *логические команды* ;
* *команды переходов*.

***Команды пересылки*** данных не требуют выполнения никаких операций над операндами. Операнды просто пересылаются (точнее, копируются) из источника (Source) в приемник (*Destination*). Источником и приемником могут быть внутренние регистры процессора, ячейки памяти или устройства ввода/вывода. *АЛУ* в данном случае не используется.

***Арифметические команды*** выполняют операции сложения, вычитания, умножения, деления, увеличения на единицу (инкрементирования), уменьшения на единицу (декрементирования) и т.д. Этим командам требуется один или два входных операнда. Формируют команды один выходной *операнд*.

***Логические команды*** производят над операндами логические операции, например, логическое И, *логическое ИЛИ*, исключающее ИЛИ, очистку, инверсию, разнообразные сдвиги (вправо, влево, арифметический сдвиг, циклический сдвиг). Этим командам, как и *арифметическим*, требуется один или два входных операнда, и формируют они один выходной *операнд*.

Наконец, ***команды переходов*** предназначены для изменения обычного порядка последовательного выполнения команд. С их помощью организуются переходы на *подпрограммы* и возвраты из них, всевозможные циклы, *ветвления* программ, пропуски фрагментов программ и т.д. *Команды переходов* всегда меняют содержимое счетчика *команд. Переходы* могут быть условными и безусловными. Именно эти команды позволяют строить сложные алгоритмы обработки информации.

В соответствии с результатом каждой выполненной команды устанавливаются или очищаются биты *регистра состояния* процессора ( *PSW* ). Но надо помнить, что не все команды изменяют все имеющиеся в *PSW* флаги. Это определяется особенностями каждого конкретного процессора.

У разных процессоров системы команд существенно различаются, но в основе своей они очень похожи. Количество команд у процессоров также различно. Например, у упоминавшегося уже процессора МС68000 всего 61 *команда*, а у процессора 8086 — 133 команды. У современных мощных процессоров количество команд достигает нескольких сотен. В то же время существуют процессоры с *сокращенным набором команд* (так называемые RISC-процессоры), в которых за счет максимального сокращения количества команд достигается увеличение эффективности и скорости их выполнения.

Рассмотрим теперь особенности четырех выделенных групп команд процессора более подробно.

#### 5.1.1. Команды пересылки данных

*Команды пересылки* данных занимают очень важное место в системе команд любого процессора. Они выполняют следующие важнейшие функции:

* загрузка (запись) содержимого во внутренние регистры процессора;
* сохранение в памяти содержимого внутренних регистров процессора;
* копирование содержимого из одной области памяти в другую;
* запись в устройства ввода/вывода и чтение из устройств ввода/вывода.

В некоторых процессорах (например, Т-11) все эти функции выполняются одной единственной командой MOV (для байтовых пересылок — MOVB ) но с различными *методами адресации* операндов.

В других процессорах помимо команды MOV имеется еще несколько команд для выполнения перечисленных функций. Например, для загрузки регистров могут использоваться команды загрузки, причем для разных регистров — разные команды (их обозначения обычно строятся с использованием слова LOAD — загрузка). Часто выделяются специальные команды для сохранения в стеке и для извлечения из стека ( PUSH — сохранить в стеке, POP — извлечь из стека). Эти команды выполняют пересылку с автоинкрементной и с автодекрементной адресацией (даже если эти режимы адресации не предусмотрены в процессоре в явном виде).

Иногда в систему команд вводится специальная команда *MOVS* для строчной (или цепочечной) пересылки данных (например, в процессоре 8086). Эта команда пересылает не одно слово или байт, а заданное количество слов или байтов ( MOVSB ), то есть инициирует не один цикл обмена по магистрали, а несколько. При этом адрес памяти, с которым происходит взаимодействие, увеличивается на 1 или на 2 после каждого обращения или же уменьшается на 1 или на 2 после каждого обращения. То есть в неявном виде применяется автоинкрементная или автодекрементная адресация.

В некоторых процессорах (например, в процессоре 8086) специально выделяются функции обмена с устройствами ввода/вывода. Команда IN используется для ввода (чтения) информации из устройства ввода/вывода, а команда OUT используется для вывода (записи) в устройство ввода/вывода. Обмен информацией в этом случае производится между регистром-аккумулятором и устройством ввода/вывода. В более продвинутых процессорах этого же семейства (начиная с процессора 80286) добавлены команды строчного (цепочечного) ввода (команда INS ) и строчного вывода (команда OUTS ). Эти команды позволяют пересылать целый массив (строку) данных из памяти в устройство ввода/вывода ( OUTS ) или из устройства ввода/вывода в память ( INS ). Адрес памяти после каждого обращения увеличивается или уменьшается (как и в случае с командой *MOVS* ).

Также к *командам пересылки* данных относятся команды обмена информацией (их обозначение строится на основе слова Exchange ). Может быть предусмотрен обмен информацией между внутренними регистрами, между двумя половинами одного регистра ( SWAP ) или между регистром и ячейкой памяти.

#### 5.1.2. Арифметические команды

*Арифметические команды* рассматривают коды операндов как числовые двоичные или двоично-десятичные коды. Эти команды могут быть разделены на пять основных групп:

* команды операций с *фиксированной запятой* (сложение, вычитание, умножение, деление);
* команды операций с *плавающей запятой* (сложение, вычитание, умножение, деление);
* команды очистки;
* команды инкремента и декремента;
* команда сравнения.

Команды операций с *фиксированной запятой* работают с кодами в регистрах процессора или в памяти как с обычными двоичными кодами. Команда сложения ( ADD ) вычисляет сумму двух кодов. Команда вычитания ( SUB ) вычисляет разность двух кодов. Команда умножения ( MUL ) вычисляет произведение двух кодов (разрядность результата вдвое больше разрядности сомножителей). Команда деления ( DIV ) вычисляет частное от деления одного кода на другой. Причем все эти команды могут работать как с числами со знаком, так и с числами без знака.

Команды операций с *плавающей запятой* (точкой) используют формат представления чисел с порядком и *мантиссой* (обычно эти числа занимают две последовательные ячейки памяти). В современных мощных процессорах набор команд с *плавающей запятой* не ограничивается только четырьмя арифметическими действиями, а содержит и множество других более сложных команд, например, вычисление тригонометрических функций, логарифмических функций, а также сложных функций, необходимых при обработке звука и изображения.

Команды очистки ( CLR ) предназначены для записи нулевого кода в регистр или ячейку памяти. Эти команды могут быть заменены *командами пересылки* нулевого кода, но специальные команды очистки обычно выполняются быстрее, чем *команды пересылки*. Команды очистки иногда относят к группе *логических команд*, но суть их от этого не меняется.

Команды инкремента (увеличения на единицу, INC ) и декремента (уменьшения на единицу, DEC ) также бывают очень удобны. Их можно в принципе заменить командами суммирования с единицей или вычитания единицы, но инкремент и декремент выполняются быстрее, чем суммирование и вычитание. Эти команды требуют одного входного операнда, который одновременно является и выходным операндом.

Наконец, команда сравнения (обозначается *CMP* ) предназначена для сравнения двух входных операндов. По сути, она вычисляет разность этих двух операндов, но выходного операнда не формирует, а всего лишь изменяет биты в регистре состояния процессора ( *PSW* ) по результату этого вычитания. Следующая за командой сравнения команда (обычно это *команда перехода* ) будет анализировать биты в регистре состояния процессора и выполнять действия в зависимости от их значений (о *командах перехода* речь идет в разделе 3.3.4). В некоторых процессорах предусмотрены команды цепочечного сравнения двух последовательностей операндов, находящихся в памяти (например, в процессоре 8086 и совместимых с ним).

#### 5.1.3. Логические команды

*Логические команды* выполняют над операндами логические (побитовые) операции, то есть они рассматривают коды операндов не как единое число, а как набор отдельных битов. Этим они отличаются от *арифметических команд*. *Логические команды* выполняют следующие основные операции:

* логическое И, *логическое ИЛИ*, сложение по модулю 2 (Исключающее ИЛИ);
* логические, арифметические и циклические сдвиги;
* проверка битов и операндов;
* установка и очистка битов (флагов) *регистра состояния* процессора ( *PSW* ).

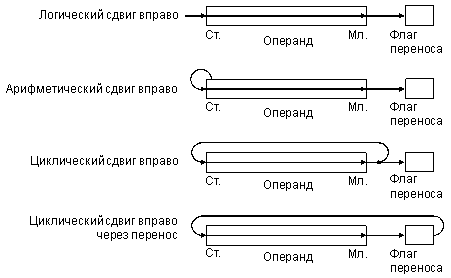
Команды логических операций позволяют побитно вычислять основные логические функции от двух входных операндов. Кроме того, операция И ( AND ) используется для принудительной очистки заданных битов (в качестве одного из операндов при этом используется код маски, в котором разряды, требующие очистки, установлены в нуль). Операция ИЛИ ( OR ) применяется для принудительной установки заданных битов (в качестве одного из операндов при этом используется код маски, в котором разряды, требующие установки в единицу, равны единице). Операция "Исключающее ИЛИ" ( XOR ) используется для инверсии заданных битов (в качестве одного из операндов при этом применяется код маски, в котором биты, подлежащие инверсии, установлены в единицу). Команды требуют двух входных операндов и формируют один выходной операнд.

Команды сдвигов позволяют побитно сдвигать код операнда вправо (в сторону младших разрядов) или влево (в сторону старших разрядов). Тип сдвига (логический, арифметический или циклический) определяет, каково будет новое значение старшего бита (при сдвиге вправо) или младшего бита (при сдвиге влево), а также определяет, будет ли где-то сохранено прежнее значение старшего бита (при сдвиге влево) или младшего бита (при сдвиге вправо). Например, при логическом сдвиге вправо в старшем разряде кода операнда устанавливается нуль, а младший разряд записывается в качестве флага переноса в регистр состояния процессора. А при арифметическом сдвиге вправо значение старшего разряда сохраняется прежним (нулем или единицей), младший разряд также записывается в качестве флага переноса.

Циклические сдвиги позволяют сдвигать биты кода операнда по кругу (по часовой стрелке при сдвиге вправо или против часовой стрелки при сдвиге влево). При этом в кольцо сдвига может входить или не входить флаг переноса. В бит флага переноса (если он используется) записывается значение старшего бита при циклическом сдвиге влево и младшего бита при циклическом сдвиге вправо. Соответственно, значение бита флага переноса будет переписываться в младший разряд при циклическом сдвиге влево и в старший разряд при циклическом сдвиге вправо.

Для примера на рис. 5.1 показаны действия, выполняемые командами сдвигов вправо.

Команды проверки битов и операндов предназначены для установки или очистки битов *регистра состояния* процессора в зависимости от значения выбранных битов или всего операнда в целом. Выходного операнда команды не формируют. Команда проверки операнда ( *TST* ) проверяет весь код операнда в целом на равенство нулю и на знак (на значение старшего бита), она требует только одного входного операнда. Команда проверки бита ( BIT ) проверяет только отдельные биты, для выбора которых в качестве второго операнда используется код маски. В коде маски проверяемым битам основного операнда должны соответствовать единичные разряды.



**Рис. 5.1.**Команды сдвигов вправо.

Наконец, команды установки и очистки битов *регистра состояния* процессора (то есть флагов) позволяют установить или очистить любой флаг, что бывает очень удобно. Каждому флагу обычно соответствуют две команды, одна из которых устанавливает его в единицу, а другая сбрасывает в нуль. Например, флагу переноса C (от *Carry*) будут соответствовать команды CLC (очистка) и SEC или STC (установка).

#### 5.1.4. Команды переходов

*Команды переходов* предназначены для организации всевозможных циклов, *ветвлений*, вызовов *подпрограмм* и т.д., то есть они нарушают последовательный ход выполнения программы. Эти команды записывают в регистр-*счетчик команд* новое значение и тем самым вызывают переход процессора не к следующей по порядку команде, а к любой другой команде в памяти программ. Некоторые *команды переходов* предусматривают в дальнейшем возврат назад, в точку, из которой был сделан переход, другие не предусматривают этого. Если возврат предусмотрен, то текущие параметры процессора сохраняются в стеке. Если возврат не предусмотрен, то текущие параметры процессора не сохраняются.

*Команды переходов* без возврата делятся на две группы:

* команды *безусловных переходов*;
* команды *условных переходов*.

В обозначениях этих команд используются слова *Branch* ( *ветвление* ) и *Jump* (прыжок).

Команды *безусловных переходов* вызывают переход в новый адрес независимо ни от чего. Они могут вызывать переход на указанную величину смещения (вперед или назад) или же на указанный адрес памяти. Величина смещения или новое значение адреса указываются в качестве входного операнда.

Команды *условных переходов* вызывают переход не всегда, а только при выполнении заданных условий. В качестве таких условий обычно выступают значения флагов в регистре состояния процессора ( *PSW* ). То есть условием перехода является результат предыдущей операции, меняющей значения флагов. Всего таких условий перехода может быть от 4 до 16. Несколько примеров команд *условных переходов*:

* переход, если равно нулю;
* переход, если не равно нулю;
* переход, если есть переполнение;
* переход, если нет переполнения;
* переход, если больше нуля;
* переход, если меньше или равно нулю.

Если условие перехода выполняется, то производится загрузка в регистр-*счетчик команд* нового значения. Если же условие перехода не выполняется, *счетчик команд* просто наращивается, и процессор выбирает и выполняет следующую по порядку команду.

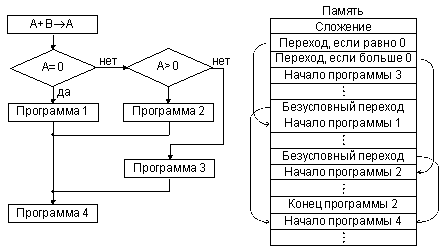
Специально для проверки условий перехода применяется команда сравнения ( *CMP* ), предшествующая команде *условного перехода* (или даже нескольким командам *условных переходов*). Но флаги могут устанавливаться и любой другой командой, например *командой пересылки* данных, любой *арифметической* или*логической командой*. Отметим, что сами *команды переходов* флаги не меняют, что как раз и позволяет ставить несколько *команд переходов* одну за другой.

Совместное использование нескольких команд условных и *безусловных переходов* позволяет процессору выполнять разветвленные алгоритмы любой сложности. Для примера на рис. 5.2  показано разветвление программы на две ветки с последующим соединением, а на рис. 5.3 — разветвление на три ветки с последующим соединением.

*Команды переходов* с дальнейшим возвратом в точку, из которой был произведен переход, применяются для выполнения *подпрограмм*, то есть вспомогательных программ. Эти команды называются также командами вызова *подпрограмм* (распространенное название — CALL ). Использование *подпрограмм* позволяет упростить структуру основной программы, сделать ее более логичной, гибкой, легкой для написания и отладки. В то же время надо учитывать, что широкое использование *подпрограмм*, как правило, увеличивает время выполнения программы.



**Рис. 5.2.**Реализация разветвления на две ветки.



**Рис. 5.3.**Реализация разветвления на три ветки.

Все *команды переходов* с возвратом предполагают безусловный переход (они не проверяют никаких флагов). При этом они требуют одного входного операнда, который может указывать как абсолютное значение нового адреса, так и смещение, складываемое с текущим значением адреса. Текущее значение *счетчика команд*(текущий адрес) сохраняется перед выполнением перехода в стеке.

Для обратного возврата в точку вызова *подпрограммы* (точку перехода) используется специальная команда возврата ( *RET* или *RTS* ). Эта команда извлекает из стека значение адреса *команды перехода* и записывает его в регистр-*счетчик команд*.

Особое место среди *команд перехода* с возвратом занимают команды прерываний (распространенное название — INT ). Эти команды в качестве входного операнда требуют номер прерывания (адрес вектора). Обслуживание таких переходов осуществляется точно так же, как и *аппаратных прерываний*. То есть для выполнения данного перехода процессор обращается к таблице векторов прерываний и получает из нее по номеру прерывания адрес памяти, в который ему необходимо перейти. Адрес вызова прерывания и содержимое *регистра состояния* процессора ( *PSW* ) сохраняются в стеке. Сохранение *PSW* — важное отличие команд прерывания от *команд переходов* с возвратом.

Команды прерываний во многих случаях оказываются удобнее, чем обычные *команды переходов* с возвратом. Сформировать *таблицу векторов прерываний* можно один раз, а потом уже обращаться к ней по мере необходимости. Номер прерывания соответствует номеру *подпрограммы*, то есть номеру функции, выполняемой *подпрограммой*. Поэтому команды прерывания гораздо чаще включаются в системы команд процессоров, чем обычные *команды переходов* с возвратом.

Для возврата из *подпрограммы*, вызванной командой прерывания, используется команда возврата из прерывания ( *IRET* или *RTI* ). Эта команда извлекает из стека сохраненное там значение *счетчика команд* и *регистра состояния* процессора ( *PSW* ).

Отметим, что у некоторых процессоров предусмотрены также команды условных прерываний, например, команда прерывания при переполнении.

Конечно, в данном разделе мы рассмотрели только основные команды, наиболее часто встречающиеся в процессорах. У конкретных процессоров могут быть и многие другие команды, не относящиеся к перечисленным группам команд. Но изучать их надо уже после того, как выбран тип процессора, подходящий для задачи, решаемой данной *микропроцессорной системой*.

### 5.2. Быстродействие процессора

*Быстродействие* процессора — это одна из важнейших его характеристик, определяющая эффективность работы всей микропроцессорной системы в целом. *Быстродействие* процессора зависит от *множества* факторов, что затрудняет сравнение быстродействия даже разных процессоров внутри одного семейства, не говоря уже о процессорах разных фирм и разного назначения.

Выделим важнейшие факторы, влияющие на *быстродействие* процессора.

Прежде всего, *быстродействие* зависит от тактовой частоты процессора. Все *операции* внутри процессора выполняются синхронно, тактируются единым тактовым сигналом. Понятно, что чем больше *тактовая частота*, тем быстрее работает *процессор*, причем, например, двукратное увеличение тактовой частоты какого-то процессора снижает вдвое *время выполнения* команд этим процессором.

Однако надо учитывать, что разные процессоры выполняют одинаковые команды за разное количество тактов, причем количество тактов, затрачиваемых на команду, может изменяться от одного такта до десятков или даже сотен. В некоторых процессорах за счет распараллеливания *микроопераций* на команду тратится даже меньше одного такта.

Количество тактов, затрачиваемых на выполнение команды, зависит от сложности этой команды и от методов адресации операндов. Например, быстрее всего (за меньшее число тактов) выполняются *команды пересылки* данных между внутренними регистрами процессора. Медленнее всего (за большое число тактов) выполняются сложные *арифметические команды* с *плавающей запятой*, операнды которых хранятся в памяти.

Первоначально для количественной оценки производительности процессоров применялась *единица* измерения *MIPS* (Mega *Instruction* *Per* *Second*), соответствовавшая количеству миллионов выполняемых инструкций (команд) за секунду. Естественно, изготовители микропроцессоров старались ориентироваться на самые быстрые команды. Понятно, что подобный показатель не слишком удачен. Для измерения производительности при выполнении вычислений с *плавающей запятой* (точкой) чуть позже была предложена *единица* *FLOPS* (*Floating point Operations* *Per* *Second*), но она по определению узкоспециальная, так как в некоторых системах *операции* с *плавающей запятой* просто не используются.

Другой аналогичный показатель быстродействия процессора — *время выполнения* коротких (быстрых) операций. Для примера в таблице 5.1 представлены показатели быстродействия нескольких 8-разрядных и 16-разрядных процессоров. В настоящее время этот показатель практически не используется, как и *MIPS*.

*Время выполнения* команд — важный, но далеко не единственный фактор, определяющий *быстродействие*. Большое *значение* имеет также *структура системы* команд процессора. Например, некоторым процессорам для выполнения какой-то *операции* понадобится одна *команда*, а другим процессорам — несколько команд. Какие-то процессоры имеют систему команд, позволяющую быстро решать задачи одного типа, а какие-то — задачи другого типа. Важны и *методы адресации*, разрешенные в данном процессоре, и наличие *сегментирования* памяти, и способы взаимодействия процессора с устройствами ввода/вывода и т.д.

Существенно влияет на *быстродействие* системы в целом и то, как *процессор* "общается" с памятью команд и памятью данных, применяется ли совмещение выборки команд из памяти с выполнением ранее выбранных команд.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Таблица 5.1. Параметры некоторых процессоров. | | | | |
| **Процессор** | **8085** | **6800** | **68000** | **8086** |
| **Фирма** | Intel | Motorola | Motorola | Intel |
| **Разрядность** | 8 | 8 | 16 | 16 |
| **Количество команд** | 80 | 72 | 61 | 133 |
| **Тактовая частота, МГц** | 3 | 1 | 8 | 5 |
| **Время выполнения коротких операций, мкс** | 1,3 | 2 | 0,5 | 0,4 |

*Быстродействие* системы в целом определяется также и разрядностью процессора. Например, 8-разрядный *процессор* будет медленнее пересылать и обрабатывать большие массивы данных, чем 16-разрядный *процессор*. Точно так же 16-разрядный *процессор* будет значительно медленнее работать с большими числами (большими, чем 65536), чем 32-разрядный *процессор*.

При высокой сложности решаемых задач *быстродействие* системы зависит и от общего объема системной памяти. Ведь если системной памяти мало, системе приходится сохранять данные во внешней памяти (например, на магнитном диске), а это очень сильно (на несколько порядков) замедляет работу. Так что*разрядность* *шины адреса* процессора тоже важна.

Поэтому количественные показатели производительности процессоров очень условны, они лишь косвенно характеризуют *быстродействие* системы на базе этого процессора. Тем не менее, некоторые производители предлагают количественные показатели для своих процессоров, которые характеризуют *время выполнения*специально составленных *тестовых программ*, содержащих самые различные команды в тех или иных соотношениях.

Так, для сравнения производительности 32-разрядных процессоров *фирма* Intel, производящая процессоры для персональных компьютеров, в 1992 году предложила свою единицу измерения *iCOMP* *Index* (Intel COmparative *Microprocessor* *Performance*). Для вычисления этого показателя используется смесь 16- и 32-битных целочисленных команд, команд с плавающей точкой, команд обработки графики и видео. В качестве базового взят *процессор* i486SX-25, чей *индекс* принят равным 100. В Таблице 3.2 приведены индексы *iCOMP* для некоторых процессоров фирмы Intel. Как видно из таблицы, за счет более развитой архитектуры процессоры семейства 486 всегда быстрее процессоров семейства 386, а любой Pentium быстрее любого процессора из семейства 486. *Тактовая частота* (указана в таблице через черточку) определяет *производительность* только в пределах одного семейства. В 1996 году разработчиками Intel был предложен другой показатель — *iCOMP* *Index* 2.0, для вычисления которого не используются 16-разрядные команды, зато введен мультимедийный тест, а за базу взят Pentium-120, чей *индекс* принят равным 100. В таблице 3.3 представлены эти показатели для некоторых типов процессоров Intel.

При этом надо учитывать, что измерения проводятся в составе системы, настроенной на максимальное *быстродействие* именно данных процессоров, и только самой фирмой Intel.

Ценность этих показателей и всех им подобных не слишком велика. Для конкретного компьютера и разных процессоров величина показателя может предоставить вполне объективные данные, позволяющие оценить, например, целесообразность замены процессора на более мощный. Но усредненность показателей *iCOMP* не позволяет точно сказать, как будет себя вести *процессор* в различных задачах, которые ориентированы на преимущественное использование разных *типов команд*.

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| Таблица 5.2. Индексы производительности *iCOMP*. | | | | | |
| i486SX-25 | 100 | | i486DX4-100 | | 435 |
| i386DX-33 | 56 | | Pentium-60 | | 510 |
| i486SX-33 | 136 | | Pentium-100 | | 815 |
| i486DX2-66 | 297 | | Pentium-133 | | 1110 |
| Таблица 3.3. Индексы производительности *iCOMP* Index 2.0. | | | | | | | |
| Pentium-100 | | 90 | | Pentium *MMX*-166 | | | 160 |
| Pentium-120 | | 100 | | Pentium *MMX*-233 | | | 203 |
| Pentium-150 | | 114 | | *Pentium Pro*-200 | | | 220 |
| Pentium-200 | | 142 | | Pentium II-266 | | | 303 |

Точная оценка быстродействия процессора возможна только в составе конкретной системы при решении определенной задачи. Но все перечисленные здесь факторы можно и нужно учитывать при выборе процессора. А количественные показатели помогают сделать выбор.

**Лекция 6. Процессорное ядро и память микроконтроллеров**

Основной особенностью современного этапа развития МПС является завершение перехода от систем, выполненных на основе нескольких больших ИС, к однокристальным *МК*, которые объединяют в одном кристалле все основные элементы МПС: центральный *процессор* (ЦП), *постоянное запоминающее устройство* (*ПЗУ*),*оперативное запоминающее устройство* (*ОЗУ*), порты ввода/вывода, таймеры.

### 6.1. Классификация и структура микроконтроллеров

В настоящее время выпускается *целый* ряд типов *МК*. Все эти приборы можно условно разделить на три основных класса:

* 8-разрядные *МК* для встраиваемых приложений;
* 16- и 32-разрядные *МК* ;
* цифровые сигнальные процессоры (*DSP*).

Наиболее распространенным представителем семейства *МК* являются 8-разрядные приборы, широко используемые в промышленности, бытовой и компьютерной технике. Они прошли в своем развитии *путь* от простейших приборов с относительно слаборазвитой *периферией* до современных многофункциональных контроллеров, обеспечивающих реализацию сложных алгоритмов управления в реальном масштабе времени. Причиной жизнеспособности 8-разрядных *МК* является использование их для управления реальными объектами, где применяются, в основном, алгоритмы с преобладанием логических операций, скорость обработки которых практически не зависит от разрядности процессора.

Росту популярности 8-разрядных *МК* способствует постоянное расширение номенклатуры изделий, выпускаемых такими известными фирмами, как Motorola, *Microchip*, Intel, Zilog, Atmel и многими другими. Современные 8-разрядные *МК* обладают, как правило, рядом отличительных признаков. Перечислим основные из них:

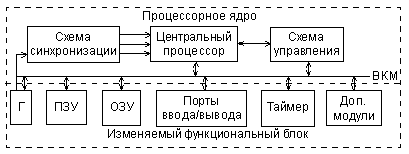
* модульная организация, при которой на базе одного *процессорного ядра* (центрального процессора) проектируется ряд (линейка) *МК*, различающихся объемом и типом *памяти программ*, объемом *памяти данных*, набором периферийных модулей, частотой синхронизации;
* использование закрытой архитектуры *МК*, которая характеризуется отсутствием линий *магистралей* адреса и данных на выводах корпуса *МК*. Таким образом, *МК* представляет собой законченную систему обработки данных, наращивание возможностей которой с использованием параллельных *магистралей* адреса и данных не предполагается;
* использование типовых функциональных периферийных модулей (таймеры, *процессоры событий*, контроллеры *последовательных интерфейсов*, *аналого-цифровые преобразователи* и др.), имеющих незначительные отличия в алгоритмах работы в *МК* различных производителей;
* расширение числа режимов работы периферийных модулей, которые задаются в процессе инициализации регистров специальных функций *МК*.

При модульном принципе построения все *МК* одного семейства содержат *процессорное ядро*, одинаковое для всех *МК* данного семейства, и изменяемый функциональный блок, который отличает *МК* разных моделей. Структура модульного *МК* приведена на рис. 6.1.

***Процессорное ядро*** включает в себя:

* центральный процессор;
* внутреннюю контроллерную магистраль (ВКМ) в составе *шин адреса*, данных и управления;
* схему синхронизации *МК* ;
* схему управления режимами работы *МК*, включая поддержку режимов пониженного *энергопотребления*, начального запуска (сброса) и т.д.

Изменяемый функциональный блок включает в себя модули памяти различного типа и объема, порты ввода/вывода, модули *тактовых генераторов* (Г), таймеры. В относительно простых *МК* *модуль* обработки прерываний входит в состав *процессорного ядра*. В более сложных *МК* он представляет собой отдельный *модуль* с развитыми возможностями. В состав изменяемого функционального блока могут входить и такие дополнительные модули как компараторы напряжения, *аналого-цифровые преобразователи* (*АЦП*) и другие. Каждый *модуль* проектируется для работы в составе *МК* с учетом протокола ВКМ. Данный подход позволяет создавать разнообразные по структуре *МК* в пределах одного семейства.



**Рис. 6.1.**Модульная организация МК.

### 6.2. Процессорное ядро микроконтроллера

#### 6.2.1. Структура процессорного ядра МК

Основными характеристиками, определяющими производительность *процессорного ядра*   *МК*, являются:

* набор регистров для хранения промежуточных данных;
* *система команд* процессора;
* *способы адресации* операндов в пространстве памяти;
* организация процессов выборки и исполнения *команды*.

С точки зрения *системы команд* и способов адресации операндов *процессорное ядро* современных 8-разрядных *МК* реализует один из двух принципов построения процессоров:

* процессоры с *CISC*-архитектурой, реализующие так называемую полную *систему команд* (Complicated *Instruction Set* Computer);
* процессоры с RISC-архитектурой, реализующие сокращенную *систему команд* (*Reduced* *Instruction Set* Computer).

*CISC*-процессоры выполняют большой набор *команд* с развитыми возможностями адресации, давая разработчику возможность выбрать наиболее подходящую *команду* для выполнения необходимой операции. В применении к 8-разрядным *МК* процессор с *CISC*-архитектурой может иметь однобайтовый, двухбайтовый и трехбайтовый (редко четырехбайтовый) формат *команд*. При этом *система команд*, как правило, неортогональна, то есть не все *команды* могут использовать любой из способов адресации применительно к любому из регистров процессора. Выборка *команды* на исполнение осуществляется побайтно в течение нескольких*циклов работы* *МК*. Время выполнения *команды* может составлять от 1 до 12 циклов. К *МК* с *CISC*-архитектурой относятся *МК* фирмы Intel с ядром *MCS*-51, которые поддерживаются в настоящее время целым рядом производителей, *МК* семейств НС05, НС08 и НС11 фирмы Motorola и ряд других.

В процессорах с RISC-архитектурой набор исполняемых *команд* сокращен до минимума. Для реализации более сложных операций приходится комбинировать *команды*. При этом все *команды* имеют формат фиксированной длины (например, 12, 14 или 16 бит), выборка *команды* из памяти и ее исполнение осуществляется за один цикл (такт) синхронизации. *Система команд* RISC-процессора предполагает возможность равноправного использования всех регистров процессора. Это обеспечивает дополнительную гибкость при выполнении ряда операций. К *МК* с RISC-процессором относятся *МК* *AVR* фирмы Atmel, *МК* PIC16 и PIC17 фирмы*Microchip* и другие.

На первый взгляд, *МК* с RISC-процессором должны иметь более высокую производительность по сравнению с *CISC* *МК* при одной и той же тактовой частоте внутренней магистрали. Однако на практике вопрос о производительности более сложен и неоднозначен.

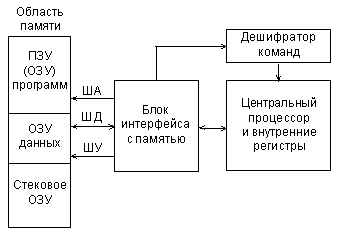
Во-первых, оценка производительности *МК* по времени выполнения *команд* различных систем (RISC и *CISC*) не совсем корректна. Обычно производительность МП и *МК* принято оценивать числом операций пересылки "регистр-регистр", которые могут быть выполнены в течение одной секунды. В *МК* с *CISC*-процессором время выполнения операции "регистр-регистр" составляет от 1 до 3 циклов, что, казалось бы, уступает производительности *МК* с RISC-процессором. Однако стремление к сокращению формата *команд* при сохранении ортогональности *системы команд* RISC-процессора приводит к вынужденному ограничению числа доступных в одной *команде* регистров. Так, например, *системой команд*   *МК* PIC16 предусмотрена возможность пересылки результата операции только в один из двух регистров — регистр-источник операнда f или рабочий регистр W. Таким образом, операция пересылки содержимого одного из доступных регистров в другой (не источник операнда и не рабочий) потребует использования двух *команд*. Такая необходимость часто возникает при пересылке содержимого одного из регистров общего назначения (РОН) в один из портов *МК*. В то же время, в *системе команд* большинства *CISC*-процессоров присутствуют *команды* пересылки содержимого РОН в один из портов ввода/вывода. То есть более сложная *система команд* иногда позволяет реализовать более эффективный способ выполнения операции.

Во-вторых, оценка производительности *МК* по скорости пересылки "регистр-регистр" не учитывает особенностей конкретного реализуемого алгоритма управления. Так, при разработке быстродействующих устройств автоматизированного управления основное внимание следует уделять времени выполнения операций умножения и деления при реализации уравнений различных *передаточных функций*. А при реализации пульта дистанционного управления бытовой техникой следует оценивать время выполнения логических функций, которые используются при опросе клавиатуры и генерации последовательной кодовой посылки управления. Поэтому в критических ситуациях, требующих высокого быстродействия, следует оценивать производительность на множестве тех операций, которые преимущественно используются в алгоритме управления и имеют ограничения по времени выполнения.

В-третьих, необходимо еще учитывать, что указанные в справочных данных на *МК* частоты синхронизации обычно соответствуют частоте подключаемого кварцевого резонатора, в то время как длительность цикла центрального процессора определяется частотой обмена по ВКМ. Соотношение этих частот индивидуально для каждого *МК* и должно быть принято в расчет при сравнении производительности различных моделей контроллеров.

С точки зрения организации процессов выборки и исполнения *команды* в современных 8-разрядных *МК* применяется одна из двух упоминавшихся архитектур МПС: фон-неймановская (принстонская) или гарвардская.

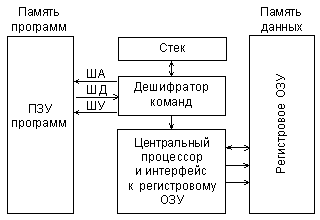
Основной особенностью фон-неймановской архитектуры является использование общей памяти для хранения программ и данных, как показано на рис. 6.2.



**Рис. 6.2.**Структура МПС с фон-неймановской архитектурой.

Основное преимущество архитектуры Фон-Неймана – упрощение устройства МПС, так как реализуется обращение только к одной общей памяти. Кроме того, использование единой области памяти позволяло оперативно перераспределять ресурсы между областями программ и данных, что существенно повышало гибкость МПС с точки зрения разработчика программного обеспечения. Размещение *стека* в общей памяти облегчало доступ к его содержимому. Неслучайно поэтому *фон-неймановская архитектура* стала основной архитектурой универсальных компьютеров, включая персональные компьютеры.

Основной особенностью гарвардской архитектуры является использование раздельных адресных пространств для хранения *команд* и данных, как показано на рис. 6.3.



**Рис. 6.3.**Структура МПС с гарвардской архитектурой.

Гарвардская архитектура почти не использовалась до конца 70-х годов, пока производители *МК* не поняли, что она дает определенные преимущества разработчикам автономных систем управления.

Дело в том, что, судя по опыту использования МПС для управления различными объектами, для реализации большинства алгоритмов управления такие преимущества фон-неймановской архитектуры как гибкость и универсальность не имеют большого значения. Анализ реальных программ управления показал, что необходимый объем *памяти данных*   *МК*, используемый для хранения промежуточных результатов, как правило, на порядок меньше требуемого объема *памяти программ*. В этих условиях использование *единого адресного пространства* приводило к увеличению формата *команд* за счет увеличения числа разрядов для адресации операндов. Применение отдельной небольшой по объему *памяти данных* способствовало сокращению длины *команд* и ускорению поиска информации в *памяти данных*.

Кроме того, гарвардская архитектура обеспечивает потенциально более высокую скорость выполнения программы по сравнению с фон-неймановской за счет возможности реализации параллельных операций. Выборка следующей *команды* может происходить одновременно с выполнением предыдущей, и нет необходимости останавливать процессор на время выборки *команды*. Этот метод реализации операций позволяет обеспечивать выполнение различных *команд* за одинаковое число тактов, что дает возможность более просто определить время выполнения циклов и критичных участков программы.

Большинство производителей современных 8-разрядных *МК* используют гарвардскую архитектуру. Однако гарвардская архитектура является недостаточно гибкой для реализации некоторых программных процедур. Поэтому сравнение *МК*, выполненных по разным архитектурам, следует проводить применительно к конкретному приложению.

#### 6.2.2. Система команд процессора МК

Так же, как и в любой *микропроцессорной системе*, набор *команд* процессора *МК* включает в себя четыре основные группы *команд*:

* *команды* пересылки данных;
* *арифметические команды* ;
* *логические команды* ;
* *команды переходов*.

Для реализации возможности независимого управления разрядами портов (регистров) в большинстве современных *МК* предусмотрена также группа *команд* битового управления (булевый или битовый процессор). Наличие *команд* битового процессора позволяет существенно сократить объем кода управляющих программ и время их выполнения.

В ряде *МК* выделяют также группу *команд* управления ресурсами контроллера, используемую для настройки режимов работы портов ввода/вывода, управления таймером и т.п. В большинстве современных *МК* внутренние ресурсы контроллера отображаются на *память данных*, поэтому для целей управления ресурсами используются *команды* пересылки данных.

*Система команд*   *МК* по сравнению с *системой команд* универсального МП имеет, как правило, менее развитые группы арифметических и логических *команд*, зато более мощные группы *команд* пересылки данных и управления. Эта особенность связана со сферой применения *МК*, требующей, прежде всего, контроля окружающей обстановки и формирования управляющих воздействий.

#### 6.2.3. Схема синхронизации МК

Схема синхронизации *МК* обеспечивает формирование сигналов синхронизации, необходимых для выполнения командных циклов центрального процессора, а также обмена информацией по внутренней магистрали. В зависимости от исполнения центрального процессора командный цикл может включать в себя от одного до нескольких (4 — 6) тактов синхронизации. Схема синхронизации формирует также метки времени, необходимые для работы таймеров *МК*. В состав схемы синхронизации входят делители частоты, которые формируют необходимые последовательности синхросигналов.

### 6.3. Память программ и данных МК

В *МК* используется три основных вида памяти. *Память программ* представляет собой постоянную *память* (*ПЗУ*), предназначенную для хранения программного кода ( *команд* ) и констант. Ее содержимое в ходе выполнения программы не изменяется. *Память данных* предназначена для хранения переменных в процессе выполнения программы и представляет собой *ОЗУ*. Регистры *МК* — этот вид памяти включает в себя внутренние регистры процессора и регистры, которые служат для управления периферийными устройствами (регистры специальных функций).

#### 6.3.1. Память программ

Основным свойством *памяти программ* является ее энергонезависимость, то есть возможность хранения программы при отсутствии питания. С точки зрения пользователей *МК* следует различать следующие типы энергонезависимой *памяти программ*:

* ПЗУ масочного типа — mask-ROM. Содержимое ячеек ПЗУ этого типа заносится при ее изготовлении с помощью масок и не может быть впоследствии заменено или допрограммировано. Поэтому *МК* с таким типом *памяти программ* следует использовать только после достаточно длительной опытной эксплуатации. Основным недостатком данной памяти является необходимость значительных затрат на создание нового комплекта фотошаблонов и их внедрение в производство. Обычно такой процесс занимает 2-3 месяца и является экономически выгодным только при выпуске десятков тысяч приборов. ПЗУ масочного типа обеспечивают высокую надежность хранения информации по причине программирования в заводских условиях с последующим контролем результата.
* ПЗУ, программируемые пользователем, с ультрафиолетовым стиранием — EPROM (Erasable Programmable ROM). ПЗУ данного типа программируются электрическими сигналами и стираются с помощью ультрафиолетового облучения. Ячейка памяти EPROM представляет собой МОП-транзистор с "плавающим" затвором, заряд на который переносится с управляющего затвора при подаче соответствующих электрических сигналов. Для стирания содержимого ячейки она облучается ультрафиолетовым светом, который сообщает заряду на плавающем затворе энергию, достаточную для преодоления потенциального барьера и стекания на подложку. Этот процесс может занимать от нескольких секунд до нескольких минут. *МК* с EPROM допускают многократное программирование и выпускаются в керамическом корпусе с кварцевым окошком для доступа ультрафиолетового света. Такой корпус стоит довольно дорого, что значительно увеличивает стоимость *МК*. Для уменьшения стоимости *МК* с EPROM его заключают в корпус без окошка (версия EPROM с однократным программированием).
* ПЗУ, однократно программируемые пользователем, — OTPROM (One-Time Programmable ROM). Представляют собой версию EPROM, выполненную в корпусе без окошка для уменьшения стоимости *МК* на его основе. Сокращение стоимости при использовании таких корпусов настолько значительно, что в последнее время эти версии EPROM часто используют вместо масочных ПЗУ.
* ПЗУ, программируемые пользователем, с электрическим стиранием — EEPROM (Electrically Erasable Programmable ROM). ПЗУ данного типа можно считать новым поколением EPROM, в которых стирание ячеек памяти производится также электрическими сигналами за счет использования туннельных механизмов. Применение EEPROM позволяет стирать и программировать *МК*, не снимая его с платы. Таким способом можно производить отладку и модернизацию программного обеспечения. Это дает огромный выигрыш на начальных стадиях разработки микроконтроллерных систем или в процессе их изучения, когда много времени уходит на поиск причин неработоспособности системы и выполнение циклов стирания-программирования *памяти программ*. По цене EEPROM занимают среднее положение между OTPROM и EPROM. Технология программирования памяти EEPROM допускает побайтовое стирание и программирование ячеек. Несмотря на очевидные преимущества EEPROM, только в редких моделях *МК* такая память используется для хранения программ. Связано это с тем, что, во-первых, EEPROM имеют ограниченный объем памяти. Во-вторых, почти одновременно с EEPROM появились Flash-ПЗУ, которые при сходных потребительских характеристиках имеют более низкую стоимость;
* ПЗУ с электрическим стиранием типа Flash — Flash-ROM. Функционально Flash-память мало отличается от EEPROM. Основное различие состоит в способе стирания записанной информации. В памяти EEPROM стирание производится отдельно для каждой ячейки, а во Flash-памяти стирать можно только целыми блоками. Если необходимо изменить содержимое одной ячейки Flash-памяти, потребуется перепрограммировать весь блок. Упрощение декодирующих схем по сравнению с EEPROM привело к тому, что *МК* с Flash-памятью становятся конкурентоспособными по отношению не только к *МК* с однократно программируемыми ПЗУ, но и с масочными ПЗУ также.

#### 6.3.2. Память данных

*Память данных*   *МК* выполняется, как правило, на основе статического ОЗУ. Термин "статическое" означает, что содержимое ячеек ОЗУ сохраняется при снижении тактовой частоты *МК* до сколь угодно малых значений (с целью снижения *энергопотребления*). Большинство *МК* имеют такой параметр, как "напряжение хранения информации" — USTANDBY. При снижении напряжения питания ниже минимально допустимого уровня UDDMIN, но выше уровня USTANDBY работа программы *МК* выполняться не будет, но информация в ОЗУ сохраняется. При восстановлении напряжения питания можно будет сбросить *МК* и продолжить выполнение программы без потери данных. Уровень напряжения хранения составляет обычно около 1 В, что позволяет в случае необходимости перевести МК на питание от автономного источника (батареи) и сохранить в этом режиме данные ОЗУ.

Объем *памяти данных*   *МК*, как правило, невелик и составляет обычно десятки и сотни байт. Это обстоятельство необходимо учитывать при разработке программ для *МК*. Так, при программировании *МК* константы, если возможно, не хранятся как переменные, а заносятся в ПЗУ программ. Максимально используются аппаратные возможности *МК*, в частности, таймеры. Прикладные программы должны ориентироваться на работу без использования больших массивов данных.

#### 6.3.3. Регистры МК

Как и все МПС, *МК* имеют набор регистров, которые используются для управления его ресурсами. В число этих регистров входят обычно регистры процессора (аккумулятор, регистры состояния, индексные регистры), регистры управления (регистры управления прерываниями, таймером), регистры, обеспечивающие ввод/вывод данных (регистры данных портов, регистры управления параллельным, последовательным или аналоговым вводом/выводом). Обращение к этим регистрам может производиться по-разному.

В *МК* с RISC-процессором все регистры (часто и аккумулятор) располагаются по явно задаваемым адресам. Это обеспечивает более высокую гибкость при работе процессора.

Одним из важных вопросов является размещение регистров в адресном пространстве *МК*. В некоторых *МК* все регистры и *память данных* располагаются в одном адресном пространстве. Это означает, что *память данных* совмещена с регистрами. Такой подход называется "отображением ресурсов *МК* на память".

В других *МК* адресное пространство устройств ввода/вывода отделено от общего пространства памяти. Отдельное пространство ввода/вывода дает некоторое преимущество процессорам с гарвардской архитектурой, обеспечивая возможность считывать *команду* во время обращения к регистру ввода/вывода.

#### 6.3.4. Стек МК

В *микроконтроллерах* ОЗУ данных используется также для организации вызова подпрограмм и обработки прерываний. При этих операциях содержимое программного счетчика и основных регистров (аккумулятор, регистр состояния и другие) сохраняется и затем восстанавливается при возврате к основной программе.

В фон-неймановской архитектуре единая область памяти используется, в том числе, и для реализации *стека*. При этом снижается производительность устройства, так как одновременный доступ к различным видам памяти невозможен. В частности, при выполнении *команды* вызова подпрограммы следующая *команда*выбирается после того, как в *стек* будет помещено содержимое программного счетчика.

В гарвардской архитектуре стековые операции производятся в специально выделенной для этой цели памяти. Это означает, что при выполнении программы вызова подпрограмм процессор с гарвардской архитектурой производит несколько действий одновременно.

Необходимо помнить, что *МК* обеих архитектур имеют ограниченную емкость памяти для хранения данных. Если в процессоре имеется отдельный *стек* и объем записанных в него данных превышает его емкость, то происходит циклическое изменение содержимого указателя *стека*, и он начинает ссылаться на ранее заполненную ячейку *стека*. Это означает, что после слишком большого количества вызовов подпрограмм в *стеке* окажется неправильный адрес возврата. Если *МК* использует общую область памяти для размещения данных и *стека*, то существует опасность, что при переполнении *стека* произойдет запись в область данных либо будет сделана попытка записи загружаемых в *стек* данных в область ПЗУ.

#### 6.3.5. Внешняя память

Несмотря на существующую тенденцию по переходу к закрытой архитектуре *МК*, в некоторых случаях возникает необходимость подключения дополнительной внешней памяти (как *памяти программ*, так и данных).

Если *МК* содержит специальные аппаратные средства для подключения внешней памяти, то эта операция производится штатным способом (как для МП).

Второй, более универсальный, способ заключается в том, чтобы использовать порты ввода/вывода для подключения внешней памяти и реализовать обращение к памяти программными средствами. Такой способ позволяет задействовать простые устройства ввода/вывода без реализации сложных шинных интерфейсов, однако приводит к снижению быстродействия системы при обращении к внешней памяти.