**ФЕДЕРАЛЬНОЕ АГЕНТСТВО СВЯЗИ**

**Федеральное государственное  
бюджетное образовательное учреждение высшего образования**

**«САНКТ-ПЕТЕРБУРГСКИЙ**

**ГОСУДАРСТВЕННЫЙ УНИВЕРСИТЕТ ТЕЛЕКОММУНИКАЦИЙ**

**им. проф. М. А. БОНЧ-БРУЕВИЧА»**

**(СПбГУТ)**

**О. Л. Неелова**

**«Архитектура вычислительных систем»**

**УЧЕБНОЕ ПОСОБИЕ**

Описание: C:\Users\XZ\Desktop\logo_1.wmf

**САНКТ-ПЕТЕРБУРГ**

**2020**

УДК 004.25 + 004.27 + 004.3 + 621.39 + 621.3.049.77

ББК 32.973.2

Рецензенты:

Заместитель начальника Центра Прикладных Исследований АО «НИИ «Вектор»

*А. А. Овчинников*

кандидат технических наук, доцент кафедры радиосвязи и вещания

*Б. Г. Шамсиев*

*Утверждено редакционно-издательским советом СПбГУТ*

*в качестве учебного пособия*

**Неелова О. Л.**

|  |  |
| --- | --- |
|  | Конспект лекций по курсу «Архитектура вычислительных систем» : учебное пособие / О. Л. Неелова; СПбГУТ. – СПб., 2020. – с.  Приводятся сведения о различных типах архитектуры микропроцессорных систем, принципах работы основных блоков, структуре процессорного ядра и путях повышения производительности процессоров.  Предназначено для подготовки бакалавров по на­правлению: 09.03.04 «Программная инженерия».  **УДК 004.25 + 004.27 + 004.3 + 621.39 + 621.3.049.77**  **ББК 32.973.2** |

Неелова О. Л., 2020

© Федеральное государственное бюджетное

образовательное учреждение высшего образования

«Санкт-Петербургский государственный университет

телекоммуникаций им. проф. М. А. Бонч-Бруевича», 2020

**ВВЕДЕНИЕ**

В настоящее время компьютеры стали неотъемлемой частью нашей жизни, но многим представляется, что они появились сравнительно недавно. На самом деле идеи создания вычислительных машин появились более 350 лет назад. Такие машины были созданы, они были, разумеется, механическими, и их можно отнести к нулевому поколению компьютеров.

Первую счетную машину с шестеренками и ручным приводом сконструировал Блез Паскаль в 1642 году. Она могла производить два действия: сложение и вычитание. Тридцать лет спустя Лейбниц построил более совершенную машину, которая могла выполнять уже и операции второй ступени – умножение и деление.

Прошло еще 150 лет и Чарльз Бэббидж, профессор математики из Кембриджа, сконструировал разностную машину для подсчета таблиц чисел для морской навигации. Она, как машина Паскаля, могла выполнять лишь операции сложения и вычитания и работала по алгоритму, основанному на методе конечных разностей. Вывод информации этой машины был предшественником перфорирования – результат выдавливался стальным штампом на медной дощечке. Бэббидж не остановился на достигнутом и, в 1834 году разработал аналитическую машину. Машина эта имела 4 компонента: память, вычислительное устройство, устройство ввода, считывающее информацию с перфокарт (металлических) и устройства вывода – перфоратор и печатающее устройство. Память этой машины состояла из 1000 слов по 50 десятичных разрядов. Для такой машины нужно уже создавать программное обеспечение. Первым человеком, создавшим программное обеспечение (первым программистом) была Ада Ловлейс, работавшая у Бэббиджа. Но, к сожалению, конструкция на шестеренках, выполняющая столь сложные функции, в 19 веке не могла быть полностью отлажена. Идеи же Бэббиджа были столь передовыми, что и в современных компьютерах есть сходства в конструкции с его аналитической машиной.

Но к идее создания вычислительной машины вернулись только спустя почти 100 лет, когда использовались электромагнитные реле. В основном, такие работы интенсивно проводились в годы Второй мировой войны для военных нужд. Из этих работ стоит упомянуть счетную машину Джона Атанасова (США). В отличие от остальных разработок того времени, в ней использовалась двоичная арифметика и память на конденсаторах, которые периодически обновлялись. Тот же принцип используется и в современных ОЗУ. К сожалению, эта машина также не стала действующей, она опередила свое время. Действующие машины тогда работали с десятичными числами и использовали реле. Одной из таких машин была *Mark I*, разработанная в Гарварде Говардом Эйкеном, который ознакомился с работой Бэббиджа. В этой конструкции была заложена модель использования отдельных областей памяти для инструкций и данных. Устройства ввода и вывода работали с перфолентой и инструкции хранились только на перфолентах, а операнды – регистрах, основанных на реле. Блок схема конструкции представлена на рис. В.1.

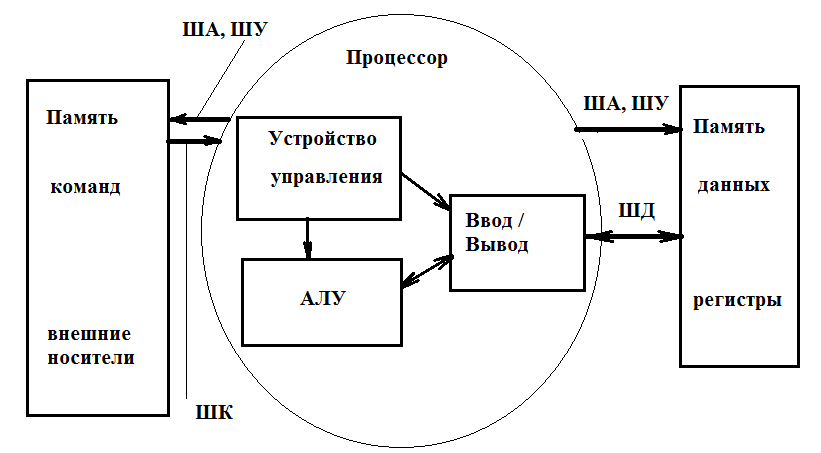


Рис. В.1. Блок схема конструкции машины Эйкена.

Машина была готова в 1944 году, и Эйкен начал работу над компьютером *Mark II*, которая устарела к моменту выпуска. После 1945 года, с созданием и применением электронных ламп, такие машины быстро устарели, и начался новый этап развития этой техники.

Первой машиной с использованием электронных ламп был *ENIAC* (*Electronic Numeral Integrator and Computer*). Однако в нем также использовалась десятичная арифметика и релейная память. Идея создания принадлежит Моушли и Экерту. Машина была создана в 1946 году, но тогда военным она не стала нужна. Один из участников проекта *ENIAC*, Джон фон Нейман, занялся собственной разработкой в Институте специальных исследований в Принстоне. Эта разработка называлась *IAS* (*Immediate Address Storage*). Принцип построения, описанный и реализованный в этом проекте, носит название Архитектуры фон Неймана, рис. В.2.

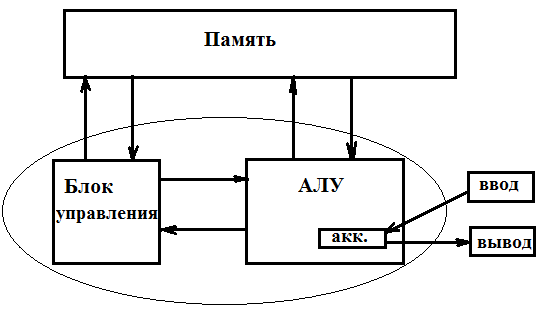


Рис. В.2. Блок схема конструкции машины фон Неймана.

В такой конструкции использовалась двоичная арифметика. Память состояла из 4096 слов длиной в 40 бит и хранила команды и данные. В каждом слове могло быть целое знаковое число длиной 40 бит, или 2 команды по 20 бит. Тип команды определяли 8 старших бит, а на 12-ти записывался адрес одного из 4096 слов в памяти. Машина исполняла только целочисленные операции.

Ламповые машины прослужили до конца 50-х годов. Изобретенные в середине 50-х транзисторы дали начало новому этапу развития компьютерной техники. Однако уже через 10 лет транзисторная техника начала вытесняться техникой на ИЛС (интегральных логических схемах) и в дальнейшем все развитие и модернизация компьютерной техники происходило в зависимости от роста степени интеграции таких схем.

Итак, на сегодняшний день мы имеем многообразие микропроцессорных систем (МПС), которые всегда имеют три обязательных блока: процессорный блок, блок памяти и блок устройств ввода-вывода, рис. В.3.

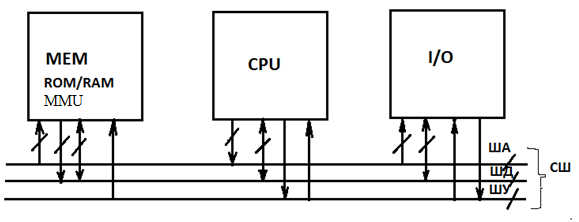


Рис. В.3. Блок схема структуры микропроцессорной системы.

1. **СИСТЕМНАЯ ШИНА**
   1. **Общие сведения о *PCI* и *PCI-Express***

Если рассматривать структуру микропроцессорной системы, как следует из рис. В.3, в виде стандартной схемы, где на общую системную шину подключаются все блоки: центральный процессор, память и адаптеры периферии, то этот вариант будет работать крайне медленно.

Когда мы рассматривали этот вариант, то полагали, что самым главным блоком является процессор, затем следует блок ввода-вывода, а уж память, это самый подчиненный блок, не вырабатывающий никаких управлений на шину. Это очевидно при рассмотрении данной схемы. Но ведь наш главный блок – процессор просто не будет выполнять никаких действий, если в памяти не будет инструкций и данных, которые необходимо обрабатывать. А блок памяти получит всю необходимую информацию только через блок устройств ввода-вывода. Вот и получается, что реально не процессор разрешает захватить шину блоку ввода-вывода, а некоторая схема-арбитр разрешает процессору работать на шине, если периферии она не нужна. Значит, на время пересылки кодов и данных с внешнего носителя в память, при работе по такой схеме, процессор не может иметь выхода на шину. Все персональные компьютеры первых моделей работали на небольших скоростях, и эта конструкция, называемая шина *ISA* (*Industry Standard Architecture* – стандартная промышленная архитектура), вполне удовлетворяла потребности пользователей. Однако, процессорный блок и периферийные адаптеры изменялись для использования на все более высоких частотах, а максимальная частота шины *ISA* составляла всего 8,33МГц при возможности передачи 2 байта/цикл.

Поэтому, в 1990 году компанией Intel была разработана новая модель структуры, разделяющая шину памяти и шину периферии. Эту шину называли *PCI* (*Peripheral Component* *Interconnect* –взаимодействие периферийных компонентов). Структура шины показана на рис. 1.1.

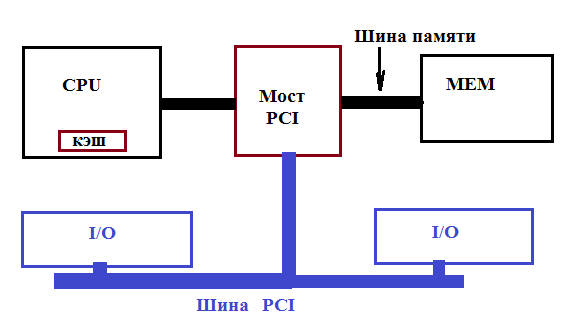


Рис. 1.1. Структура шины *PCI*.

Процессорный блок обменивается информацией с блоком памяти по низкочастотной шине. С помощью схемы моста к этой части подсоединяется шина периферии, работающая на более высоких частотах. Периферийные устройства прежних поколений, выходящие на ISA, также могли работать в системе, т.к. *ISA* подключалась к *PCI* с помощью своего моста. Все запросы и транзакции на шине *PCI* происходили только параллельным образом. Для экономии количества проводников адресная часть и данные совмещались и выводились на 64 проводника. Арбитраж осуществлялся с помощью специальной схемы, встроенной в мост. Минимальная транзакция занимает три цикла: в первом задающее устройство выставляет на шину адрес подчиненного и сигналы управления для начала транзакции; во втором цикле адрес удаляется, шина передается подчиненному устройству; во время третьего цикла подчиненное устройство выдает необходимую информацию на шину. Если этого не происходит, вводится режим ожидания.

Начало каждого цикла отсчитывается по спаду импульса синхронизации. Максимальная частота генерируемых тактовых импульсов для *PCI* – 66МГц.

Таким образом, при 64 разрядах шины, ее пропускная способность составляла:

**64** бита/цикл или **64**х**66**х**106**:**8** = **4224**х**106**:**8** = **528**Мбайт/сек.

Это довольно высокая производительность, но вскоре она уже перестала удовлетворять условиям обмена информацией для подключаемых высокочастотных устройств. Всевозможные дополнительные мосты только усложняли схему и увеличивали габариты платы. К тому же, при работе на высоких частотах параллельным способом, неизбежно возникают помехи и ошибки. Поэтому шина *PCI* была заменена на *PCI Express*, рис. 1.2.

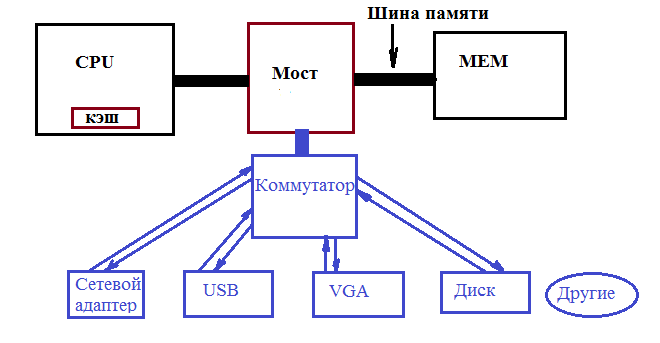


Рис. 1.2. Структура *PCI Express.*

Теперь в схему моста к шине памяти включается коммутатор, подключающий пары последовательных линий, сигнальной и заземляющей. Минимальное количество таких пар – одна, но может допускаться до 32 (кратно степени 2). Каналы, входящие в пары, называются полосами.

Параллельная передача слов, требующая дополнительных управляющих сигналов, заменяется передачей пакетов. Такая передача не требует дополнительных управляющих сигналов, так как любой пакет имеет заголовок, содержащий адресную и управляющую информацию. Перед заголовком может передаваться порядковый номер пакета. После заголовка следуют непосредственно передаваемые данные, называемые полезной нагрузкой. Для определения начала посылки информации и разделения информационных пространств, применяется кодирование 8/10. Каждый байт кодируется при помощи 10-разрядного символа. За полезной нагрузкой обязательно должен следовать код исправления ошибок – *CRC*(*Cycle Redundancy Check* – циклический код избыточности). Этот код генерируется путем вычисления полинома, составленного на основе заголовка и полезной нагрузки. Устройство, получившее пакет, также анализирует заголовок и полезную нагрузку. Если коды CRC совпадают, отправителю высылается пакет подтверждения. Если же коды не совпали, делается запрос на повторную высылку пакета.

Передача информации по двум проводам не требует тактового генератора. Любое подключенное устройство начинает передавать информацию с той частотой, на которой оно работает. Для исключения перегрузки получателя, имеющего меньшую скорость работы, чем отправитель, применяется механизм управления потоками. Он заключается в выдаче отправителю определенного количества разрешений на передачу пакетов. Таким образом, при возможной скорости передачи, значительно превышающей скорость приема, информация выдается порционно. Каждый раз, перед отправлением следующего допустимого объема информации, отправитель дожидается разрешения от получателя.

1. **АРХИТЕКТУРА БЛОКА ПАМЯТИ**
   1. **Преобразование памяти**

Микропроцессорные системы могут строиться на основе как *RISC* так и *CISC* процессоров, и использовать архитектуру фон Неймана или модифицированную Гарвардскую в зависимости от круга исполняемых задач. Основными требованиями при построении этих систем всегда является получение высокой производительности без усложнений программирования и унификация внешних разъемов. А так как производительность зависит от способа обмена процессор-память, то сначала обратимся к основным особенностям конструкций памяти.

В структуре микропроцессорной системы присутствует блок управления памятью – *MMU*. Что входит в функции этого блока?

Для ответа на этот вопрос необходимо определить понятия **виртуальной памяти** и **памяти физической**.

**Физическая память**, это тот реальный объем памяти, который есть в МПС. В компьютерах 50-х годов, объем памяти которых едва превышал 64Кбита с экзотическими параметрами, например, 4096 слов по 18 разрядов, программист имел возможность записать программу во внутреннюю память целиком. Это была, конечно, виртуозная работа, и часто приходилось использовать не самые быстрые алгоритмы для уменьшения объема кода. Затем, по мере усложнения решаемых задач, коды изначально записывали на внешнем диске, но разбивали программы на участки, **оверлеи**, и загружали их во внутреннюю память поочередно. Это было очень долго и тяжело, так как при написании программы все время приходилось помнить о реальном объеме памяти системы. Программисту приходилось самостоятельно управлять обменом между основной внутренней памятью и вспомогательной памятью. Поэтому, еще в 1961году группой исследователей из Манчестера был предложен метод, который предполагает автоматизацию загрузки участков программы во внутреннюю память. Метод основан на использовании **виртуальной памяти**.

Суть этого метода заключается в возможности дополнительного описания фрагмента, загружаемого из внешней памяти во внутреннюю. Такое описание может основываться на страничной организации внешней памяти или на ее сегментации.

* + 1. ***Страничная организация памяти***

Предположим, имеется маленькая МПС с 4К основной памяти. Тогда, если внешнюю память разбить на пропорциональные (по 4К) участки, то возможно копировать содержимое любого из таких участков во внутреннюю память, рис. 2.1.

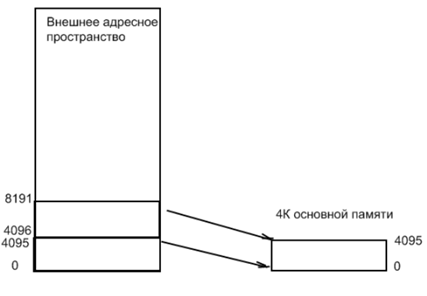


Рис. 2.1. Пример копирования участка внешней памяти в память внутреннюю.

Однако, основная память реально имеет гораздо больший объем, поэтому удобнее и основную память представить в виде страниц. Размеры страниц виртуальной и физической памяти всегда одинаковы. Таким образом, если представим страницу размером 4К, а основную (физическую) память, имеющую размер 32К, то такая память вмещает 8 страничных кадров, которые могут быть помещены с любых страниц виртуальной памяти.

Первая таблица, табл. 2.1, показывает распределение 64К младших адресов виртуального пространства, вторая, табл. 2.2 – распределение 32К внутренней памяти. Размер страницы – 4К.

Таблица. 2.1.

|  |  |
| --- | --- |
| страница | Виртуальный адрес |
| ------ | ------- |
| 15 | 61440-65535 |
| 14 | 57344-61439 |
| 13 | 53248-57343 |
| 12 |  |
| 11 |  |
| 10 |  |
| 9 |  |
| 8 | 32768-36863 |
| 7 | 28672-32767 |
| 6 |  |
| 5 |  |
| 4 |  |
| 3 |  |
| 2 | 8192-12287 |
| 1 | 4096-8191 |
| 0 | 0-4095 |

Таблица 2.2.

|  |  |
| --- | --- |
| Страничный кадр | Физические адреса |
| 7 | 28672-32767 |
| 6 |  |
| 5 |  |
| 4 |  |
| 3 |  |
| 2 | 8192-12287 |
| 1 | 4096-8191 |
| 0 | 0-4095 |

Теперь предположим, что виртуальная память для нашего примера составляет 4Gb. Тогда появляется вопрос, как нам отобразить 32-битный виртуальный адрес на 15-битный физический?

Все необходимые преобразования производятся в ***MMU*** (*Memory* *Management Unit*) или диспетчер памяти. В этом блоке, в частности, размещаются специальные таблицы – *Translation Look-aside Buffers*(***TLB*** – буфер быстрого преобразования адреса). Таблицы представляют собой блок ассоциативной памяти, структура которого, соответственно, имеет поле тега и поле данных. При загрузке страниц в память в поле тега загружается состояние старших разрядов виртуального адреса, отображающих адрес страницы, и выставляется бит присутствия. Таким образом, при каждом обращении по преобразованию адресов, поле тега позволяет отследить, имеется ли соответствующая страница виртуальной памяти в памяти физической. Это происходит путем сравнения тега на шине с сохраненными тегами. В поле данных записывается адрес страничного кадра. При нахождении соответствия тегов состояние поля данных заносится в определенные разряды физического адреса.

Схема, иллюстрирующая преобразование 32-битного виртуального адреса в 15-битный физический, представлена на рис. 2.2.

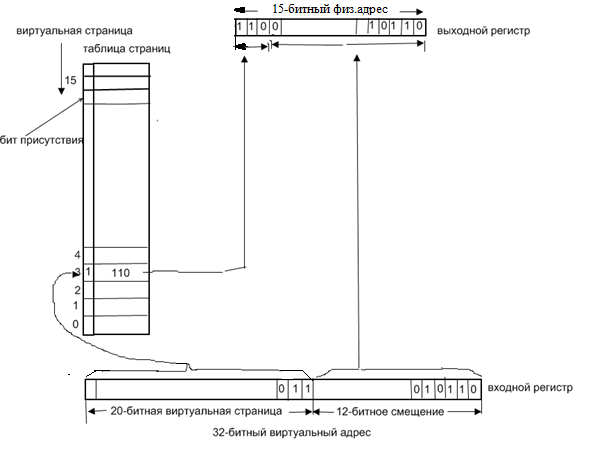


Рис. 2.2. Схема преобразования виртуального адреса в физический, при объеме внешней памяти 4*Gb* и объеме внутренней памяти 32*Kb*.

В структуру виртуального адреса входит адрес страницы и адрес внутри страницы. Так как мы считаем, что размер страницы 4К, то на адрес внутри страницы отводим 12 бит. Номер виртуальной страницы используется в качестве индекса для таблицы страниц. В данном случае это номер 3. Из таблицы выбирается 3-й элемент, и диспетчер памяти проверяет, находится ли текущая страница в данный момент в памяти (у нас 220 виртуальных страниц и 8 физических, поэтому такая проверка необходима). Контроллер проверяет бит присутствия. Если он «1», то такая страница в памяти есть, и теперь из таблицы выбирается номер страничного кадра. В нашем примере он равен 6-ти. Это число копируется в старшие 3 разряда выходного регистра, а в 12 младших разрядов параллельно копируется содержимое соответствующих 12 бит входного регистра.

Если же бит присутствия оказывается равным «0», что называется ошибкой отсутствия страницы, такую страницу нужно вызвать из внешней памяти. Изначально, очевидно, все биты присутствия нулевые. Рассмотрим сначала механизмы загрузки страниц в физическую память. Общеизвестны два метода: вызов страниц по требованию и набор рабочего множества. При вызове страниц по требованию страницы переносятся в основную память только в случае необходимости. Например, при вызове процессором первой команды программы сразу произойдет ошибка отсутствия, потому что в основной памяти еще ничего нет. Поэтому страница, содержащая первую команду, будет загружена в память и внесена в таблицу страниц. Далее, во фрагменте программы, загруженном в основную память, встречается адрес перехода, выходящий за пределы данной страницы. Опять возникает ошибка отсутствия, и страница, содержащая фрагмент перехода, также заносится в основную память и т.д. Такой механизм эффективен при запуске программы, далее же нужные страницы уже будут присутствовать в основной памяти. Если же программа состоит из нескольких процессов, разделенных во времени, этот метод не подойдет. В таком случае используют набор рабочего множества. Метод основан на статистике обращений программ к набору страниц, и именно эти страницы по умолчанию загружаются в память при каждом перезапуске.

Если ошибка отсутствия страницы произошла при заполненной основной памяти, то при этом для страницы необходимо освободить место, т.е. какую-то страницу придется удалить. Как определить, какую страницу убрать?

Для этого можно использовать или алгоритм ***LRU*** (*Least Recently Used* – алгоритм удаления дольше всего не использовавшихся страниц), или алгоритм ***FIFO***. Рассмотрим вкратце эти алгоритмы.

По первому алгоритму производится статистика обращений, и выявляются страницы, к которым программа обращается меньше всего. При возникновении ошибки отсутствия происходит удаление страницы, к которой дольше всех не обращались, и замена ее на запрашиваемую.

По второму алгоритму каждому страничному кадру соответствует свой счетчик. При вызове страницы ее счетчик обнуляется, а значение счетчиков всех присутствующих в памяти страничных кадров увеличивается на 1. Удаляется та страница, которой соответствует наибольшее состояние счетчика.

Оба алгоритма работают в случае, когда объем доступной памяти превышает размер рабочего множества.

Удаленная из основной памяти страница записывается обратно на диск только в том случае, если информация на ней менялась. Изменение информации на странице фиксируется с помощью специального бита в диспетчере страниц. Этот бит сбрасывается при загрузке страницы и устанавливается, когда информация на ней меняется.

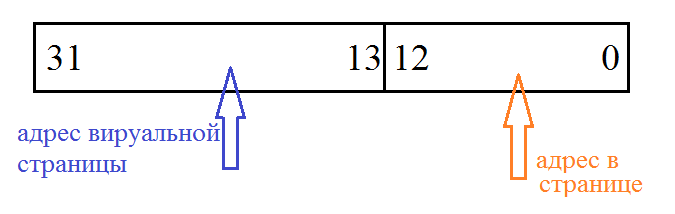
Выбор оптимального размера страниц производится, исходя из следующих соображений: при больших размерах страниц остается много неиспользуемого пространства на странице, кроме того, большой размер страницы не позволяет выделить в основной памяти большого количества страничных кадров, что вызывает пробуксовку при замене страниц; при страницах малого размера пробуксовка будет возникать реже и коэффициент использования пространства страницы будет выше, но для таблицы *TLB* потребуется большое количество регистров, что ведет к удорожанию аппаратуры и увеличению времени загрузки и сохранения содержимого этих регистров при запуске программы и ее остановке. Эффективность работы диска также выше при большом размере страниц. Поэтому 4К, представленные в нашем примере, это минимальный размер страницы.

Пример преобразования памяти с помощью *TLB* показан на рис.2.3(а, б).

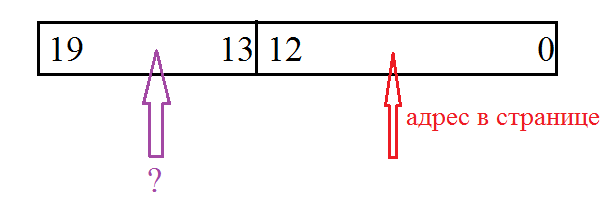
Пусть надо преобразовать 4Гбайт виртуального пространства со страницами размером 8 Кбайт в 1 Мбайт физического пространства со страничными кадрами размером 8 Кбайт.

4 Гбайт – 232 ; 1 Мбайт = 220 ; 8 Кбайт – 213.

Вычислим количество страничных кадров: 220/ 213 = 27 (128)



а)



б)

Рис. 2.3(а – вид виртуального адреса, б – вид физического адреса).

Предположим, виртуальный адрес – 0х000195С3.

Номер страницы выясняем по состоянию старших 19 разрядов. 15 старших разрядов в нуле. На остальных четырех 1100 (12). Предположим, что один из тегов TLB совпал. Данные под этим тегом – 25(11001). Это номер страничного кадра.

Получаем физический адрес 0х335С3.

***2.1.2.*** ***Сегментация памяти***

Второй способ получения виртуального адресного пространства – сегментация памяти. При этом образуется сразу несколько адресных пространств, каждое из которых может содержать свой тип информации.

Обратимся сначала к простому примеру сегментации, который применялся для реального режима процессоров линейки *Intel 8086/88.* Шина адреса там была 20-разрядная, что позволяло бы обращаться к 1Мбайту памяти. Все регистры были 16-разрядные. Таким образом, если взять для описания адресов сегментов 16-разрядный регистр, а потом определять смещения внутри сегмента по регистру такой же разрядности, то возможное количество адресов увеличится до 4 Гбайт

216 х 216 = 232

Для описания адресов сегментов использовалось 4 регистра. Они назывались: *CS, DS, ES и SS*. По смыслу это сегменты кода, данных и стека. Базовыми регистрами для определения внутрисегментного адреса являлись регистры *IP, BX, SP*. Пример проиллюстрирован на рис. 2.4.

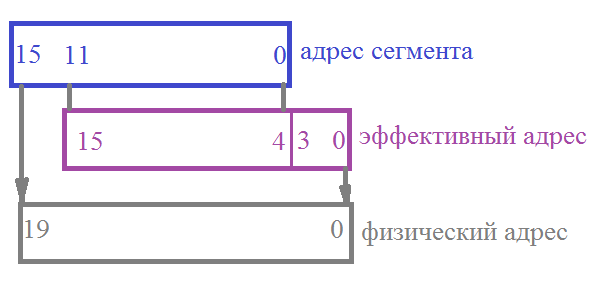


Рис. 2.4. Получение физического адреса для реального режима процессоров *Intel86/88*.

Для получения физического адреса состояние сегментного регистра сдвигалось влево на 4 разряда и складывалось со смещением внутри сегмента (эффективный адрес).

Например*: CS: 0x1A47, IP: 0x0092*

*0x1A470 + 0x0092 =0x1A512*

Или *DS: 0x1731, BX: 0x3202*

*0x17310 + 0x3202 = 0x1A512*

В современных процессорных системах, где адрес выводится на 32 разряда шины, применяется другая схема.

Виртуальная память поддерживается с помощью двух таблиц дескрипторов: *LDT (Local Descriptor Table)* и *GDT (Global Descriptor Table).* Локальная таблица индивидуальна для каждой программы и поддерживает в ней все типы сегментов, а глобальная едина для всех программ пользователей и для операционной системы.

В сегментные регистры (*CS, DS, ES, FS, GS и SS*) загружается селектор сегмента – индекс (номер элемента в таблице дескрипторов), принадлежность к одной из таблиц (*LDT/GDT*) и уровень привилегий программы (0-3), рис. 2.5.

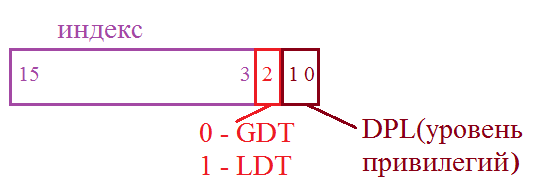


Рис. 2.5. Формат селектора.

После загрузки селектора в регистр соответствующий дескриптор вызывается из таблицы (локальной или глобальной) и его содержимое сохраняется во внутренних регистрах *MMU* для облегчения доступа и ускорения обмена. Для вызова доступен любой дескриптор, кроме нулевого.

Получение указателя на дескриптор следующее:

1. Определяется тип таблицы дескрипторов.
2. Адрес дескриптора из соответствующего типа таблиц сохраняется во внутреннем регистре *MMU*. На рис. 2.6 определим его *Rad*.
3. Индекс, содержащийся в селекторе используемого сегмента, копируется во внутренний регистр *MMU* на соответствующие разряды (15 – 3), а младшие разряды в регистре обнуляются. На рис. 2.6 определим этот регистр *Ras*.
4. Состояния регистров складывается. Сумма представляет собой адрес ячейки, в которой записан номер элемента выбранной таблицы дескрипторов.

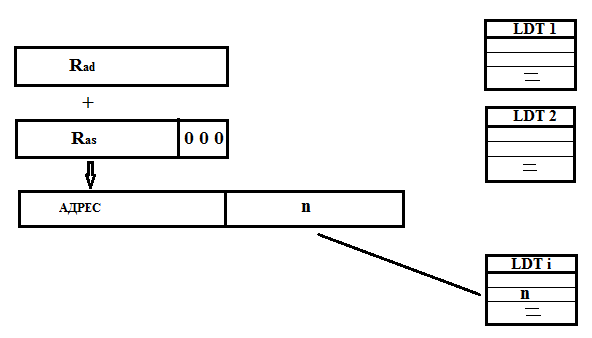


Рис. 2.6. Получение указателя на дескриптор.

Формат дескриптора – 64 бита, рис. 2.7.

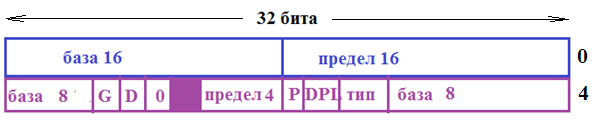


Рис. 2.7. Формат дескриптора.

База –базовый адрес сегмента;

Предел – длина сегмента;

Тип – тип сегмента;

*G(granularity)* –степень дробления поля предел (0- в байтах, 1 – в страницах);

*D* – разрядность сегмента (0- 16, 1 -32);

*P* – бит присутствия сегмента в памяти;

*DPL* – уровень привилегий (0 -3). Уровень привилегий отражает степень защиты работающей программы. Самый низкий уровень привилегий у обычных пользовательских программ. Далее – общие библиотечные процедуры, системные вызовы и ядро операционной системы.

В локальной таблице каждому сегментному регистру соответствует свой тип дескриптора. Вызов дескриптора производится, если этот сегмент присутствует в памяти, при условии, что индекс дескриптора отличен от 0.

Затем производится проверка соответствия смещения размеру сегмента. При 32-разрядном сегменте необходимо представлять предел в страницах, так как размер страницы не меньше 4Кбайт, а поле лимита 20 разрядов.

Если смещение не превышает размер сегмента, то формируется линейный адрес. К базовому адресу из дескриптора прибавляется смещение. Линейный адрес выставляется как физический в том случае, когда нет разбиения сегментов на страницы.

В случае разбиения сегментов на страницы линейный адрес будет еще не физическим, а виртуальным. Он будет содержать адрес каталога страниц, адрес страницы в таблице и адрес символа на странице, рис. 2.8.

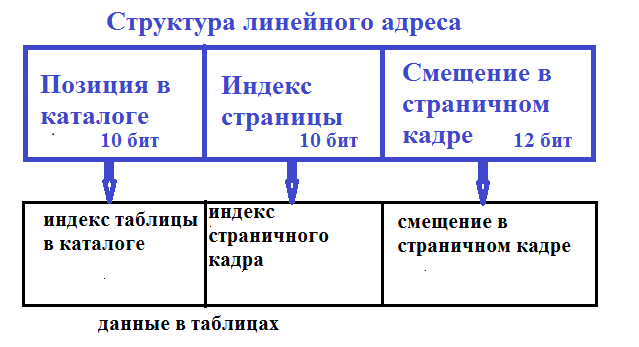


Рис. 2.8. Структура линейного адреса.

Рассмотрим пример, проиллюстрированный на рис. 2.9.

Предположим, что мы получили линейный адрес *0х01C1B035.*

Таким образом, из каталога страниц на 7 позиции мы находим указатель на, предположим 10-ю таблицу и находим в ней 27 страницу. На 27 странице записан 20 номер страничного кадра. Размер страничного кадра 4 Кбайта, или 4096 элементов адреса. Начало нашего страничного кадра – 81920.

Смещение внутри – 53. Физический адрес – 81973. На шине *0х00014035.*

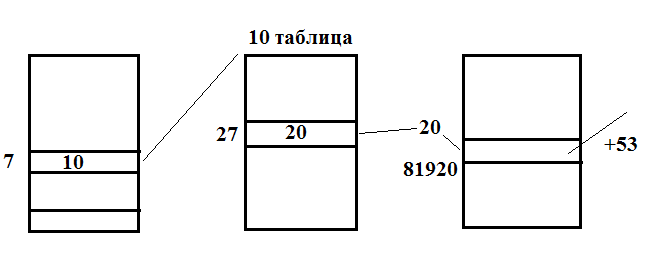


Рис. 2.9. Получение физического адреса из линейного.

* 1. **Фрагментация**

При преобразовании памяти, как в страничной организации, так и в сегментированной, неизбежно появляются неиспользованные участки. Эта проблема называется фрагментацией. При страничной организации памяти возникает **внутренняя фрагментация**, так как размер страницы всегда фиксирован. Неиспользованными остаются участки страниц после выполнения переходов, или если это последняя страница программы. Для сокращения внутренней фрагментации удобнее использовать страницы небольшого объема. Но, с точки зрения структуры диспетчера памяти, использование маленьких страниц очень невыгодно, так как увеличивается количество регистров в *TLB*.

В отличие от страниц, сегменты не имеют фиксированного размера. Размер сегмента оговаривается в структуре программы. Поэтому, при замене сегментов, в памяти возникают пустоты. Это **внешняя фрагментация**, рис. 2.10.

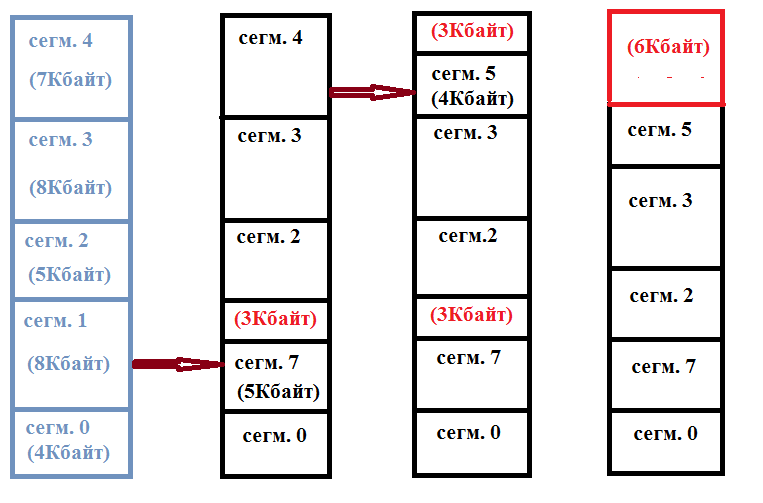


Рис. 2.10. Внешняя фрагментация.

Очевидно, что появление таких участков понижает эффективность работы системы. Способы предотвращения этой ситуации могут быть различны.

1. Уплотнение сегментов – «выдавливание» пустых пространств. Такое уплотнение можно производить или сразу по появлению пустого пространства, или после некоторого накопления таких пространств, когда доля незаполненных участков достигнет определенного допустимого процента от общего объема памяти. Этот способ имеет один существенный недостаток – дополнительные затраты времени на процесс.
2. Алгоритм оптимальной подгонки. В этом случае должен присутствовать список всех адресов и размеров пустот. При копировании сегмента в память выбирается самая маленькая из пустот, в которую он может поместиться.
3. В этом случае также имеется список всех пустот, но оптимальной подгонки не производится, а сегмент копируется в первое подходящее для него пространство. Этот способ позволяет получить меньшее количество маленьких пустых пространств, чем алгоритм оптимальной подгонки.

Но два последних способа все равно порождают накопление пустых пространств, которые невозможно ничем заполнить. Поэтому объединение таких пространств, дефрагментация все равно необходима.

То есть если основываться на простой подкачке сегментов, не разделяя их на страницы, мы все время будем сталкиваться с проблемой внешней фрагментации. Поэтому сегменты разбивают на страницы фиксированного размера. Это снижает размеры внешней фрагментации, хотя в этом случае появляется фрагментация внутренняя. Но надо помнить, что при этом линейный адрес, полученный путем сложения базы дескриптора со смещением, уже не будет представлять собой физический адрес, а станет указателем для следующей системы таблиц. Как рассмотрено в примере, приведенном в предыдущей лекции, в случае разбиения сегментов на страницы линейный адрес будет еще не физическим, а виртуальным. Он будет содержать адрес каталога страниц, адрес страницы в таблице и адрес символа на странице.

Различные процессорные системы поддерживают свои способы преобразования памяти. В персональных компьютерах (*Core i7*)поддерживаются все перечисленные способы преобразования. Системы, работающие с задачами меньших объемов, как, например, системы, основанные на *ARM*7, поддерживают только страничное преобразование.

Но, в отличие от старых систем, преобразование памяти в современных системах служит в большей степени для оптимизации ее использования.

Рассмотрим организацию памяти в системах на основе процессоров *ARM*7.

Здесь поддерживается 4 варианта размера страниц: 4Кбайт, 64Кбайт, 1Мбайт и 16Мбайт.

Соответственно, смещение будет определяться в 1 варианте – на 12 битах, во втором – на 16, в третьем – 20, а в четвертом – 24.

Так как количество страниц при 32 разрядной организации будет соответствовать количеству страничных кадров. В *TLB* хранятся номера только последних использованных 128 страниц, причем страницы команд и данных рассматриваются отдельно. Информация об остальных страничных кадрах присутствует в страничных таблицах. Эти таблицы имеют два уровня: для страниц большого объема первый уровень, допускающий 12-разрядный индекс, а для маленьких страниц на первом уровне записывается указатель на 8-ми разрядный второй. К таблицам идет обращение только в случае *TLB*-промаха.

**2.3. Способы повышения производительности блока памяти**

При рассмотрении структуры микропроцессора, прежде всего вспоминают о его производительности, и способах ее повышения. Но, обычно, основная память в микропроцессорной системе базируется на динамических ячейках. Поэтому в системе возникают проблемы задержек доставки операнда (время ожидания) и пропускной способности шины (объема данных, передаваемого в единицу времени).

Если рассматривать структуру динамической памяти, *DRAM*, то очевидно, что в ней на обращение за единицей информации тратилось 5 тактов. Это показано на рис. 2.11, где начало обращения происходит по отрицательному краю импульса стробирования адреса строки, *RAS*, а окончание обращения по его положительному краю.

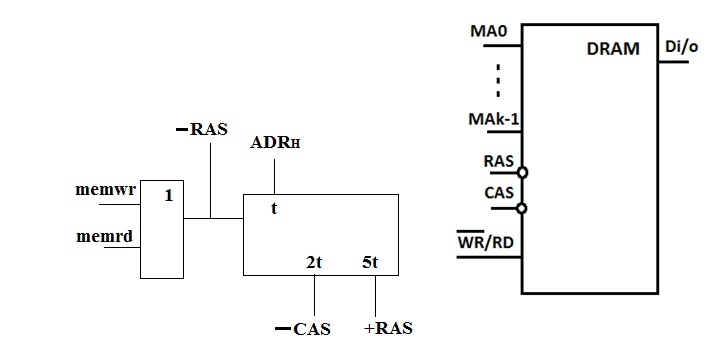


Рис. 2.11. Формирование обращения к *DRAM.*

При разделении памяти на банки, рис.2.12, время обращения сокращалось, но пропускная способность шины все равно была недопустимо низкая. Применение схем технологий *SDRAM* и *DDR SDRAM* значительно улучшило положение.

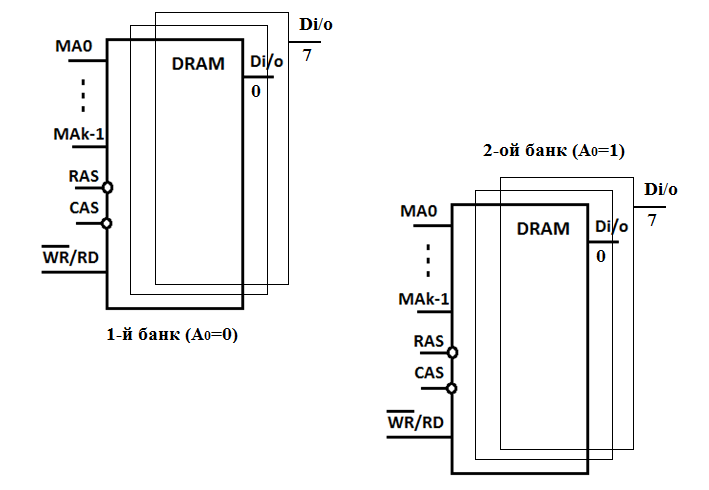


Рис. 2.12. Разделение памяти на банки по младшему адресному разряду.

Рассмотрим вкратце технологию *SDRAM*. *SDRAM* (*Synchronous Dynamic Random Access Memory*) – это синхронизированная динамическая оперативная память. Появилась она в 1997 году. *SDRAM* состоит из физических ячеек, которые собраны в страницы. Размер страницы может быть от 512 байт до нескольких килобайт. Каждая страница разбита на два банка: в одном банке ячейки с нечетными адресами строк, а в другом – с четными (для ускорения работы памяти). Обращение к ячейке *SDRAM* аналогично обращению к ячейке *DRAM*, но в страничном режиме, передав номер строки, можно получить доступ ко всем ячейкам с разными номерами столбцов, то есть не надо для каждой из них передавать номер строки, достаточно только номера столбца – экономится цикл. Это возможно потому, что сигналы обращения к строке *DRAM* синхронизированы с тактовым генератором системной платы. Такой режим называется *Fast Page Mode*. Во время обращение к одному банку в другом производится выборка адреса - такой режим иногда называют расслоением. Для того чтобы увеличить скорость доступа к памяти, используется пакетный режим (*burst*) доступа: после установки строки и столбца ячейки происходит обращение к следующим трем смежным адресам без дополнительных состояний ожидания. Схема пакетного режима будет выглядеть так: *x-y-y-y*, где *х* – время выполнения первой операции доступа состоящей из продолжительности цикла и времени ожидания, а *y* – это число циклов, необходимое для выполнения каждой последующей операции. Для *SDRAM* схема будет выглядеть так: 5-1-1-1. Диаграмма работы *SDRAM* представлена на рис. 2.13.

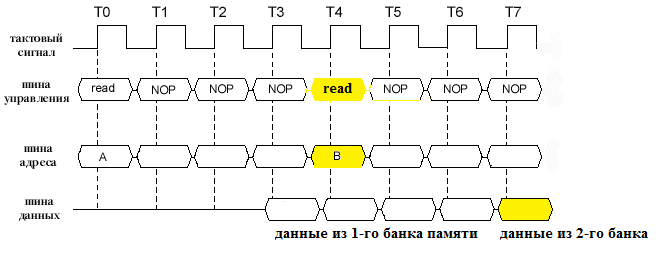


Рис. 2.13. Диаграмма работы *SDRAM*.  
Для повышения производительности обмена процессора с памятью применяется *DDR – DDR SDRAM* (*Double Data Rate Synchronous Dynamic Random Access Memory* – динамическая синхронизированная память с произвольным порядком выборки и удвоенной передачей данных). Основное отличие *DDR SDRAM* от обычной *SDRAM* заключается в том, что за один цикл происходит два обращения к данным: по положительному и отрицательному краям импульса тактового сигнала системной шины - чтение/запись. В схеме *DDR SDRAM*  формируется дополнительный строб. Это достигается установкой инверторов в цепях опроса и вывода данных на шину, позволяющих отрицательный край импульса синхронизации воспринимать как положительный. Таким образом, скорость передачи данных процессору повышается в два раза.  
Диаграмма работы *DDR SDRAM* представлена на рис. 2.14.

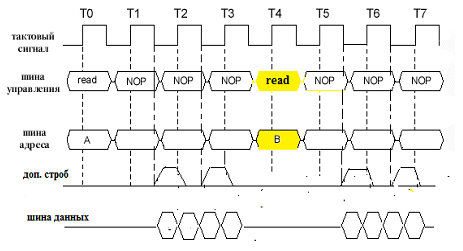


Рис. 2.14. Диаграмма работы *DDR SDRAM.*

Однако, даже при применении *DDR SDRAM*, основная память остается динамической. Это не способствует повышению производительности системы.

Производительность системы значительно повышается при применении статической памяти, *SRAM*, построенной на основе триггерных ячеек. Использование только такого типа памяти значительно поднимает стоимость производства процессорной системы, поэтому как основная память *SRAM* используется в маленьких системах, а в крупных ставится в дополнение к основной *DDR SDRAM* . Этот метод известен как кэширование.

Одним из способов решения проблем повышения производительности является многоуровневое кэширование. Кэш первого уровня располагается в самой схеме процессора и всегда разделенная (команды и данные отдельно). Кэш второго уровня вынесена за пределы схемы процессора, но может находиться в одном корпусе с ней. Это более простая по конструкции – объединенная кэш-память (данные и команды совместно). Далее, в зависимости от сложности процессора, располагается кэш третьего уровня, она уже находится в отдельном корпусе. В *Core I7*, где коды команд могут генерировать микрокоманды, в блоке дешифрации есть еще кэш нулевого уровня, содержащая эти микрокоды. Для линейки *ARM-7* иерархия памяти выглядит следующим образом, рис. 2.15.

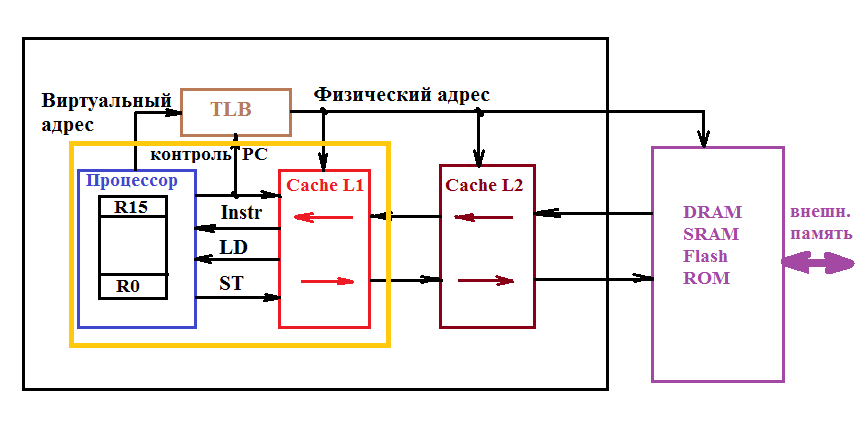


Рис. 2.15. Иерархия памяти для процессоров линейки *ARM*-7.

Теперь кратко обратимся к структуре кэш-памяти, общее представление о которой мы уже имеем.

Здесь мы будем обращаться к структуре *CortexA9* (линейка *ARM* -7), и для примера возьмем кэш объема 64Кбайта.

Так как общий доступный объем памяти по 32-разрядной шине составляет 4Гбайта, то мы имеем

232 / 216 = 216

страниц такой памяти. Если взять длину адреса строки 32 байта и кэш прямого доступа, то адрес, выставляемый программным счетчиком, будет иметь следующую структуру, представленную на рис. 2.16.

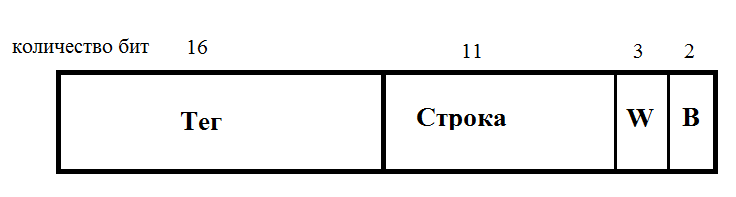


Рис. 2.16. Структура физического адреса при обращении к кэш.

Каждая страница содержит 2048 строк по 8 слов в каждой. Размер слова – 4 байта. Структура такой кэш представлена на рис. 2.17.

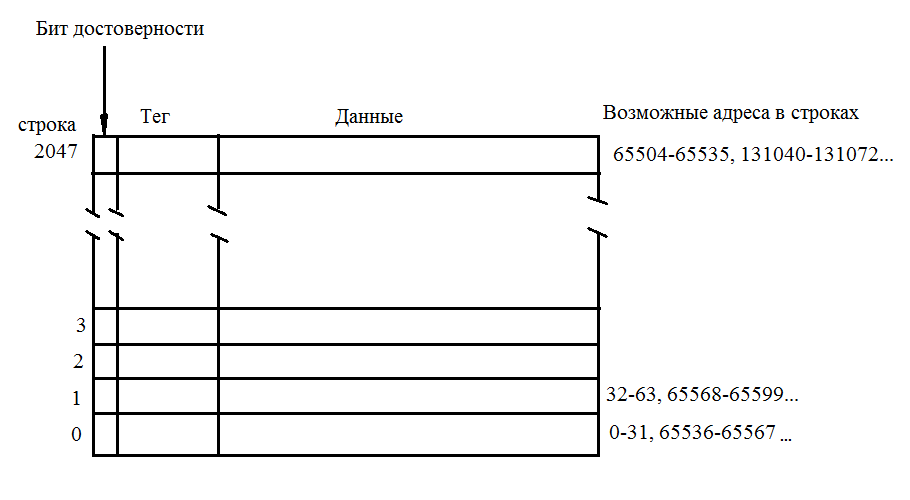


Рис. 2.17. Структура кэш прямого отображения.

При копировании страниц в кэш присутствует пространственная локализация адресов и временная локализация адресов. Первый случай основан на вероятности последовательного выполнения команд программы (последовательной обработки массива). Второй случай характерен для работы в циклах или со стеком.

Кэш прямого отображения не просто копирует определенную страницу из основной памяти, но набирает страницу из строк с любой из возможных страниц, не изменяя при этом их последовательности на основных страницах. Понятно, что, при таком построении кэш, в ней не могут храниться две одноименные строки с разных страниц. Поэтому, при частых последовательных обращениях к одноименным строкам, находящимся на разных уровнях, могут возникать проблемы конкуренции строк и, вследствие этого задержки, связанные с перезагрузкой строки в кэш. Чтобы этого не происходило, в каждом элементе кэш помещают несколько одноименных строк. Самым оптимальным вариантом является ассоциативная кэш-память, содержащая по 4 такие строки (4-входовая ассоциативная кэш-память). Теперь каждая доступная строка при том же объеме кэш будет содержать всего 2 слова. Структура такой памяти представлена на рис. 2.18.

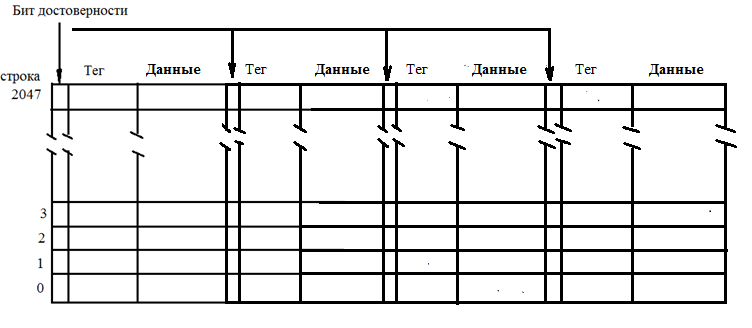


Рис. 2.18. Структура 4-входовой ассоциативной кэш.

В процессоре *CortexA*9 кэш первого уровня – ассоциативная 4-входовая. Длина строкового элемента (адреса строки) кэш – 8 слов (32 байта). Возможны конфигурации кэш по объемам 16 Кбайт, 32 Кбайт, 64 Кбайт. Кэш имеет два 32-разрядных входных буфера и один 32 –разрядный выходной буфер для замещаемых строк. Кроме этого, в структуре кэш есть 64-разрядный 4-входовой буфер хранения данных.

В случае кэш-промаха, при нулевом бите достоверности, должно происходить замещение строк.

Для кэш инструкций замещение строк может производиться как по принципу *FIFO* (наибольшая отработка строки), так и по принципу дольше всего не использовавшейся - *LRU*(*least recently used*). Для кэш данных при замещении используется только алгоритм *LRU*.

1. **АРХИТЕКТУРА ПРОЦЕССОРНОГО ЯДРА**
   1. **Общая структура микропроцессора**

Общая структура микропроцессора показана на рис. 3.1. В качестве примера рассматривается микропроцессор *CortexA9.*

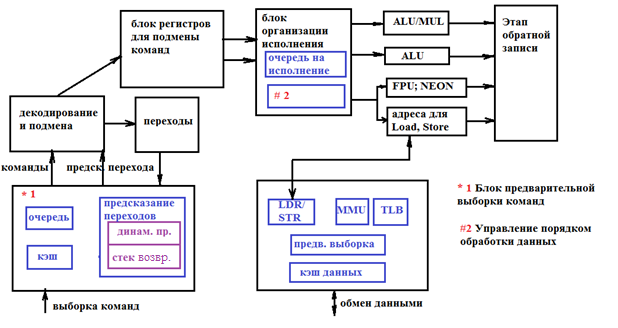


Рис. 3.1. Общая структура микропроцессора.

В структуру процессора входят:

- блок предварительной выборки команд;

- блок декодирования и подмены;

- блок организации исполнения;

- исполнительный блок;

- блок осуществления обратной записи.

Блок предварительной выборки команд включает в себя:

- первичную очередь команд, поступающих из кэш инструкций;

- блок организации прогнозирования переходов.

Блок декодирования и подмены включает в себя:

- блок декодирования;

- блок подтверждения перехода;

- блок подмены команд;

- блок подмены регистров.

Блок организации исполнения включает в себя:

- вторичную, реальную очередь команд;

- блок управления порядком обработки данных.

Исполнительный блок включает в себя:

- блок переключений вычислительных функций: блок целочисленных вычислений или вычислений с плавающей точкой;

- АЛУ, АЛУ + умножитель, умножитель и прочие блоки исполнения команд;

- блок регистров с подключением кэш данных.

Блок осуществления обратной записи включает в себя:

- интерфейсы для кэш L2, GIC (контроллер прерываний) и т.п.

Первые три блока нужно рассматривать в комплексе, также как и последующие два.

При обращении к кэш инструкций программный счетчик выставляет на адресной шине последовательность адресов и команды, идущие непрерывным потоком, поступают на блок предварительной очереди. Блок очереди всегда представляет собой регистр сдвига, построенный на ячейках стандартного для используемого процессора формата, кратного 16-ти битам. По сути, это FIFO небольшого объема. Предварительной эта очередь является потому, что выполнение команд в таком порядке следования неэффективно, как будет показано далее.

В частности, большие потери процессорного времени влекут за собой команды условных переходов. Поэтому в современных процессорах всегда есть блоки прогнозирования переходов, позволяющие удержать определенную ветвь программы с первого этапа получения потока команд. Структура и функционирование таких блоков будет рассмотрено в данной главе далее.

С предварительной очереди команды поступают на блок дешифрации, который связан с блоками подмены команд и блоками резервных регистров для подмены основных.

В результате произведенных на этом этапе операций первоначальный порядок следования команд изменяется, и образуется окончательная очередь команд в порядке их реального исполнения. С этой очереди команды поступают в исполнительный блок.

В зависимости от кода операции команда, сходящая с очереди, поступает в назначенный раздел исполнительного блока: команды арифметики первой ступени и логических действий – в АЛУ, умножения – в умножитель, прямой или обратной загрузки – на блок регистров и т.д.

Блок переключений вычислительных функций позволяет определить, какой тип обработки данных должно поддерживать ядро процессора. Это может быть ядро, позволяющее векторные вычисления над целочисленными величинами, или числами с плавающей точкой *– NEON MPE(Media Processing Engine)*. Оно поддерживает медиа-приложения, такие, как 3*D*.

Второй тип позволяет работать со скалярными величинами, производит вычисления над переменными с плавающей точкой. Для *CortexA9* блок называется *FPU (Floating Point Unit*). Медиа-приложений он не поддерживает.

Результат произведенной операции необходимо передать получателю. Иногда получатель имеет адрес памяти, не принадлежащей первому уровню кэш. Для поиска адреса в этом случае необходим интерфейс подключения второго уровня кэш. Для организации прерываний необходим интерфейс контроллера – *GIC*. Все эти интерфейсы относятся к последнему блоку, хотя их подключение может потребоваться и на других этапах.

* 1. **Методы повышения производительности.**

Все современные микропроцессоры построены на основе конвейерной обработки и выполнения команд. Классический конвейер имеет пять ступеней (этапов), известных нам как Выборка кода из памяти, Декодирование, Исполнение (или выборка операндов), Результат (выполнение команд) и Обратная запись.

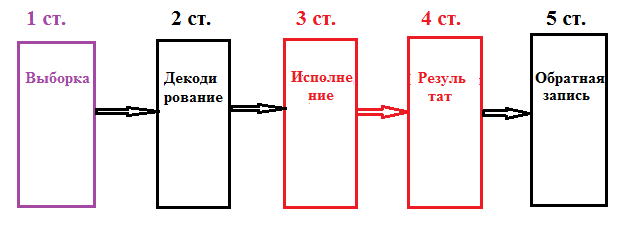


Рис. 3.2. Этапы конвейера.

В представленной на рис. 3.2 схеме имеются участки, которые могут замедлять, или даже приостанавливать работу конвейера. Это ступень выборки при разветвленных алгоритмах, а также ступени 3 и 4. Действительно, если для получения результата команде *N* требуются операнды, полученные в результате выполнения команды *N*-1, то, с большой долей вероятности, ожидается временная задержка на получение этих операндов. На примере структуры *Cortex A9* посмотрим, как удается минимизировать временные потери, и как в результате преобразуется схема конвейера.

Структура процессора *Cortex A9* (без интерфейсов для прерываний) приведена на рис. 3.1.

Нам известно, что команды, необходимые для выполнения поставленной программы, записываются в кэш команд. Кэш команд размещается в блоке предварительной выборки; выбираемые из этой памяти команды программы поступают в предварительную очередь команд. Если бы команды, выходящие из этой очереди, сразу поступали на дешифрацию, а затем исполнение, то неизбежны простои конвейера. В частности, это возможно при командах переходов.

Предположим, имеется участок программы:

\*\*\*\*\*\*\*\*

*CMP R*3*, #*0

*BGE MET*1

*MOV R*5*, R*3

*B MET*2

*MET*1*: ADD R*2*, R*1*, R*3

\*\*\*\*\*\*\*\*\*\*

Мы знаем, что на момент исполнения команды условного перехода по *MET*1, следующая команда уже находится на этапе дешифрации, поэтому обязательно должна исполниться. Но в программе, если содержимое *R*3 меньше «0», исполнение этой команды не нужно. Первые конвейеризированные машины в таких случаях просто простаивали, пока не начиналась выборка из области, в которую совершен переход. Но, при таком подходе, чем более разветвлена программа, тем больше время простоя. С безусловными переходами дело обстоит несколько проще. Некоторые машины поддерживают **слот отсрочки**. Это позиция, куда помещается команда, следующая за безусловным переходом. Для условных переходов тоже есть слот отсрочки, но при этом надо еще принимать решение по условию и передать блоку выборки. А на это уходит время. Поэтому были разработаны технологии прогнозирования переходов. Эти технологии могут опираться как на использование программных средств (статическое прогнозирование переходов), так и на использование аппаратных средств (динамическое прогнозирование переходов).

При **статическом прогнозировании** переходов решение о совершении перехода принимает компилятор. Он сообщает аппаратуре о ее дальнейших действиях. Известные способы статического предсказания для команд условного перехода можно классифицировать следующим образом:

1. Переход происходит всегда (ПВ).

Этот способ предполагает, что каждая команда условного перехода в программе обязательно завершится переходом. Таким образом, после команды условного перехода, дальнейшая выборка команд всегда производится, начиная с адреса перехода. При проведении исследования с помощью тестовых программ точность такого предсказания оказывается в пределах от 55% до 76%. Этот способ предсказания применялся в 486 процессорах фирмы Intel.

1. Переход никогда не происходит (ПН).

Следующий способ предполагает, что ни одна из команд условного перехода в программе никогда не завершится переходом. Поэтому дальнейшая выборка команд, следующих за командой условного перехода, продолжается в естественном порядке. Такой способ предсказания давал точность не более 50%, тем не менее, он реально применялся в некоторых микропроцессорах фирмы *DEC*.

1. Предсказание определяется по результатам профилирования.

При этом изначально выполнение программы производится при некотором эталонном наборе исходных данных, который позволяет произвести сбор статистики об исходах каждой команды условного перехода. После сбора статистики тем командам, которые чаще завершались переходом, назначается стратегия ПВ, а всем остальным – ПН. Выбор стратегии фиксируется в специальном бите кода операции. Некоторые компиляторы самостоятельно проводят профилирование и по результатам профилирования устанавливают специальный бит в коде команд условного перехода. При выполнении программы в зависимости от состояния этого бита делается предсказание исхода команды условного перехода и определяется адрес той команды, которая должна выполняться после команды условного перехода. Основным недостатком этого способа является тот факт, что изменение набора исходных данных для профилирования может существенно менять поведение одних и тех же команд условного перехода. Поэтому вероятность правильного предсказания невысока и примерно равна 75%.

1. Предсказание определяется кодом операции команды перехода.

Все команды условного перехода делят на две группы: в командах одной группы переход определяется, как совершающийся всегда, а в командах другой – никогда. Распределение команд на группы можно проводить либо по результатам профилирования, либо по кодам команд. Например, стратегию ПВ можно назначить командам перехода по условиям, определяемым флагом нулевого результата, а всем остальным командам – стратегию ПН. Этот способ имеет примерно такую же точность предсказания, что и третий, т.е. примерно 75%.

1. Предсказание зависит от направления перехода.

Для команд, в которых адрес перехода меньше содержимого счетчика команд, назначается стратегия ПВ, поскольку по статистике именно такие команды используются для организации циклов. Согласно статистике переход назад выполняется в 85% случаев. Для команд с переходом по адресу, превышающему адрес команды перехода, т.е. переходом вперед, назначается стратегия ПН. Этот способ еще называют: переход назад происходит всегда, а его точность также равна 75%.

6. При первом выполнении команды переход имеет место всегда.

Этот способ предсказания статическим является только частично, поскольку предсказания на последующее выполнение команды зависят от правильности начального предсказания, т.е. стратегия однозначно определяет исход команды только при первом ее выполнении, а все последующие переходы определяются уже при выполнении программы. Точность прогноза данного способа несколько выше, чем у всех предшествующих, но этот способ трудно реализовать на практике для больших программ из-за того, что нужно отслеживать слишком много команд условного перехода.

Таким образом, можно заметить, что статическое предсказание переходов достаточно просто реализуется, но имеет относительно невысокую точность предсказания. С целью повышения точности предсказания на практике применяют динамическое предсказание переходов.

**Динамическое прогнозирование** переходов выполняется во время работы программы, приспосабливаясь к текущему режиму.

Технология динамического прогнозирования предполагает наличие в структуре ядра специальной таблицы, в которую процессор записывает все встречающиеся условные переходы. При повторном появлении условного перехода его можно найти в таблице. Таблицы динамического прогнозирования переходов имеют структуру кэш-памяти (аналогично *TLB*). Существующие технологии динамического прогнозирования переходов различаются способом получения тега.

Для прогнозирования многократных переходов, в случае циклической обработки, удобнее использовать способ получения тега по состоянию программного счетчика.

Если учитывать, что формат команды перехода 32 разряда, т.е. 4 байта, то в формировании адреса 2 младших разряда не участвуют. Адрес перехода, исключая два младших разряда, заносится в таблицу с установкой бита достоверности. Предположим, что таблица имеет *k* строк. Тогда тег выделяется из состояния программного счетчика на битах с *k*+1 по 2-ой, как указано на рис. 3.3.

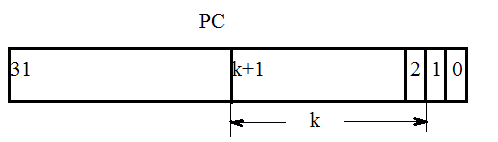


Рис. 3.3. Принцип выделения тега при прогнозировании многократных переходов.

Если тег совпадает с адресом, указанным в таблице, и бит достоверности установлен, то из таблицы извлекаются биты прогнозирования. Если при обращении в таблицу совершается промах (несовпадение тега или бит достоверности 0), то можно применять довольно простое правило.

Считается, что в циклах чаще всего осуществляются переходы назад, поэтому предположить, что выполняться будут все условные переходы назад. В этом случае встретившийся переход вперед выполнен не будет. Этот метод прост, но, при неправильном прогнозировании, довольно сложно отменять выполненные зря команды. Для этого надо использовать скрытые регистры, куда будут записываться результаты команд до выяснения правильности произведенного прогноза.

Биты прогнозирования, записанные в таблице, показывают, совершался ли переход в прошлый раз (младший бит) и прогноз перехода (старший бит).

Иногда используют только один бит прогнозирования, но это влечет за собой больше ошибок при выполнении программы. В этом случае единственный несовершенный переход из множества совершенных приводит к последующему сбою. Поэтому удобнее отмечать дальнейший прогноз по двум одинаково свершившимся событиям. Такой алгоритм можно представить работой конечного автомата на 4 состояния, как показано на рис. 3.4. Если переходы не совершались 2 и более раз, автомат находится в состоянии **00**. После первого перехода прогноз на отсутствие сохраняется, но выставляется бит совершения перехода – **01**. Затем, если переход совершится вновь, автомат перейдет в состояние **11**, т.е. даст прогноз дальнейшего перехода при втором свершившемся. Если после этого переход опять не произойдет, автомат не сбросит прогноз, но перейдет в **10**, из которого может или вернуться в **11**, или уйти в **00**.

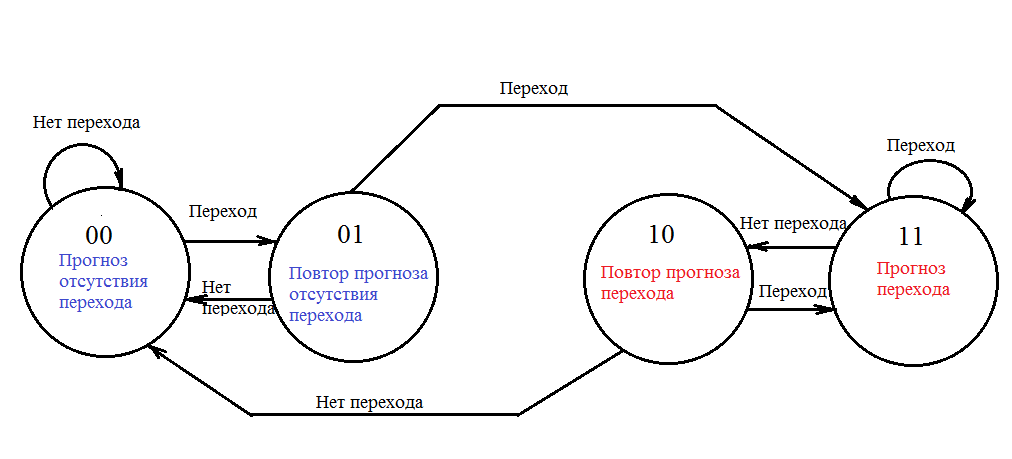


Рис. 3.4. Алгоритм работы конечного автомата для установки бита совершенного перехода и бита прогнозирования.

Иногда таблицы строятся таким образом, что указывают не только непосредственно адреса команд переходов , но и целевой адрес. Структуры таблиц приведены на рис. 3.5.

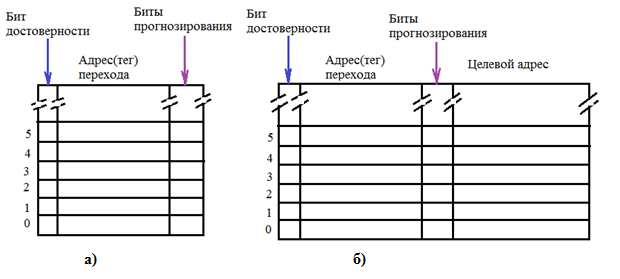


Рис. 3.5. Структуры таблиц: а) – таблица с указанием адреса перехода; б) – таблица с указанием адреса команды перехода и целевого адреса.

Для исключения конфликта строк таблицы, как и в случае кэш прямого отображения, используются 2-входовые или 4- входовые ассоциативные структуры.

Для прогнозирования одиночных переходов, возникающих при создавшемся условии, для получения тега применяется «регистр глобальной истории» (*Global History Buffer – GHB*), или буфер общей статистики выполненных переходов, рис. 3.6. Такой метод основан на использовании *N*-разрядного регистра сдвига, собирающего статистику о *N* переходах программы. После выполнения каждой текущей команды перехода содержимое регистра сдвигается, и в освободившийся разряд записывается 1, если переход был выполнен или 0, если переход не был выполнен. Для таблицы, содержащей *k* строк, в таком регистре выделяется *k* младших разрядов под теговое слово (*k* может быть равно *N*). В таблице каждому тегу будет соответствовать адрес перехода и запись комбинации битов прогнозирования.

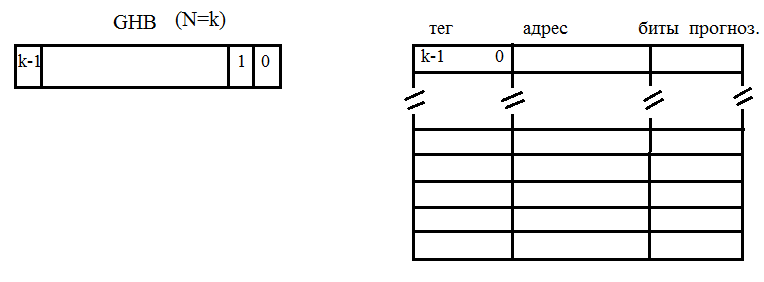


Рис. 3.6. Структура «регистра глобальной истории».

Для возможности прогнозирования как циклических, так и одиночных переходов, эти способы формирования тегов применяются комбинированно.

На рис. 3.7 показана структура блока предварительной выборки команд процессора *CortexA*9, в котором применяется комбинированный способ формирования тегов. Рассмотрим блок *BP*.

Блок прогнозирования переходов обеспечивает 2-уровневый механизм прогнозирования, включающий в себя:

2-входовую кэш адресов переходов (*Branch Target Address Cache – BTAC*) с различными возможностями *RAM*-конфигурации:

2x256 элементов для 512-элементной *BTAC*;

2x512 элементов для 1024-элементной *BTAC*;

2x1024 элементов для 2048-элементной *BTAC*;

1x20948 элементов для 4096-элементной *BTAC*;

Буфер общей статистики выполненных переходов (*Global History Buffer – GHB*), содержащий от 1024 до 16384 строк с 2-битными указателями прогноза.

Стек возврата с 8-ю 32-битными элементами сохраняет и возвращает адреса и результаты выполнения операций при прогнозировании переходов с функциями вызова. Стек связан с состоянием *LR* (*R*14).

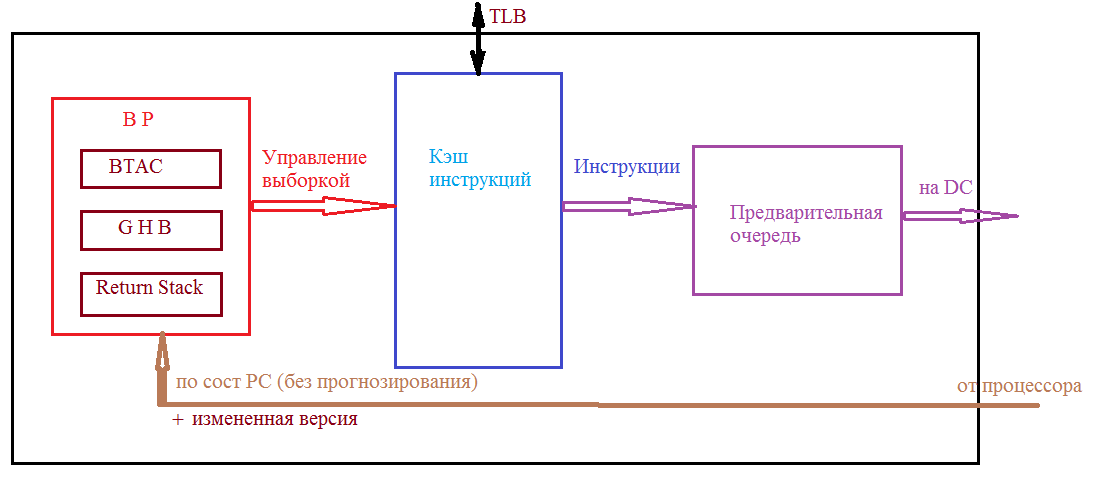


Рис. 3.7. Структура блока предварительной выборки команд *CortexA*9.

Блок предварительной выборки передает команды блоку декодирования и подмены. Рассматривая структуру конвейера, мы заметили, что нестыковки во временных соотношениях для некоторых случаев выполнения команд (3ступень – 4ступень) могут привести к замедлению работы процессора. Для уверенной работы должны быть соблюдены следующие зависимости:

***RAW*** - взаимосвязь. (***Read After Write***). Считывать состояние регистра команда может лишь после того, как в него записала данные предыдущая команда.

***WAR*** – взаимосвязь. (***Write After Read***). Переписать состояние регистра команда может только тогда, когда его считала предыдущая команда.

***WAW*** – запись после записи. (***Write After Write***). Записывать операнд-источник в регистр можно только после записи результата.

Таким образом, запускать команду на 3 стадию нельзя в трех случаях:

1. Если какой-либо операнд записывается.
2. Если считывается состояние регистра результатов.
3. Если записывается регистр результатов.

Если предположить, что команды должны исполняться строго в том порядке, в котором они поступали в блок декодирования, конвейер будет простаивать.

Поэтому используют методы подмены команд и подмены регистров.

Например, во фрагменте

*MUL R*4*, R*1*, R*0

*ADD R*2*, R*4*, R*3

*EOR R*5*, R*5

команда сложения не может быть запущена на исполнение, так как состояние *R*4 еще не записано (*RAW*). В этом случае может быть вместо нее исполнена команда логического сравнения. Но рассмотрим случай, когда третьей командой в подобном фрагменте окажется, например:

*MUL R*4*, R*1*, R*0

*ADD R*2*, R*4*, R*3

*MOV R*3, #10

В таком примере подмена второй команды третьей невозможна из-за *WAR*-взаимосвязи.

Для рассмотрения примера порядка исполнения и подмены команд и регистров еще раз обратимся к структуре конвейера. Мы рассматривали только обычный конвейер, но, для большей эффективности его работы в компьютерах, основанных на *RISC*-процессорах, начали применять сдвоенный конвейер, рис. 3.8. При этом блок выборки команд выставлял на конвейер сразу две команды.

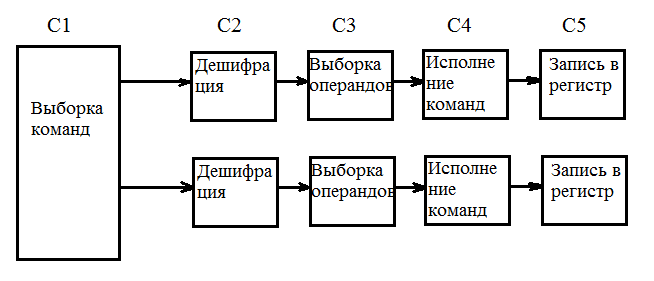


Рис. 3.8. Сдвоенный конвейер.

Команды должны быть независимы друг от друга. У них должны быть разные источники и приемники, и результат одной команды не должен влиять на выполнение другой. У компании *INTEL* подобная схема начала применяться в Pentium, но там был один основной конвейер, в который могли поступать любые команды, а второй вспомогательный, для простых команд. Операндами простых команд могли быть только целые числа. Если к сложной команде не находилось пары (несовместимая, или также сложная команда), то работал только основной конвейер, а ко второй команде, в свою очередь, подбиралась пара.

Очевидно, что такой путь наращивания числа параллельных конвейеров не слишком эффективен из-за жестких требований к выбираемым командам. Тем более, что в каждом из конвейеров самым длительным будет четвертый этап. Поэтому распараллеливание начали делать на этапе *С*4 – исполнение команд. Такой подход к построению конвейера носит название **суперскалярная архитектура**, рис. 3.9.

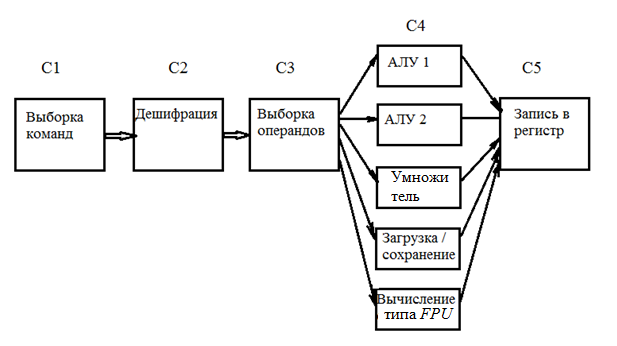


Рис. 3.9. Суперскалярная архитектура.

Казалось бы, скорость выполнения операций в таком случае вырастает пропорционально числу параллельных блоков исполнения, но, в случае возникающих известных нам конфликтов операндов, выдача команд временно прекращается.

Теперь мы можем рассмотреть пример выполнения фрагмента программы на суперскалярной архитектуре, считая, что блок декодирования выдает по две команды за цикл.

1. ***MUL R*3*, R*0*, R*1**
2. ***ADD R*4*, R*0*, R*2**
3. *ADD R*5*, R*0*, R*1
4. *ADD R*6*, R*1*, R*4
5. ***MUL R*7*, R*1*, R*2**
6. ***SUB R*1*, R*0*, R*2**

Изначально предположим, что команды должны исполняться строго в том порядке, как они поступают из блока выборки. А затем посмотрим, что изменится, если будет разрешена подмена команд и регистров.

В любом случае блок подмены задействует счетчики, обсчитывающие количество обращений к регистрам источникам (с них производится считывание), к регистрам приемникам (в них производится запись), и к различным функциональным блокам этапа исполнения. Это необходимо для устранения возможных конфликтов. Для регистров-источников счетчики подсчитывают количество использований каждого регистра в выполняющихся командах. Как только команда, содержавшая поименованный регистр выполнится, состояние счетчика уменьшается на «1».

Для регистров-приемников в счетчике содержится по одному биту на регистр. Если этот бит установлен в «1», это означает, что команда еще не выполнена, и обращение других команд к данному регистру невозможно.

Запишем порядок работы в виде таблиц

Выполнение в порядке следования из блока предварительной выборки представлено в табл. 3.1.

Таблица 3.1.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| № цикла | команды | выдача | Результат  в регистре | комментарии |
| 1 | *MUL R*3, *R*0, *R*1  *ADD R*4*, R*0*, R*2 | 1  2 | -  - | *R*0*,R*1*-rd; R*3*-wr*  *R*0*,R*2*-rd; R*4*-wr* конфликтов нет |
| 2 | *ADD R*5*, R*0*, R*1  *ADD R*6*, R*1*, R*4 | 3  - | -  - | *R*0*,R*1*-rd; R*5*-wr*  *R*1*-rd, R*4 *(RAW)*  Выдача приостановлена |
| 3 |  |  |  | Не готов р-т *MUL* |
| 4 |  |  | 1  2  3 | Свободен для чтения *R*4 |
| 5 | *MUL R*7*, R*1*, R*2 | 4  5 |  | Свободен для чтения *R*4  *R*1*,R*2*-rd; R*7*-wr* |
| 6 | *SUB R*1*, R*0*, R*2 |  |  | *R*1*(WAR)* |
| 7 |  |  | 4 |  |
| 8 |  |  | 5 | Свободен для записи *R*1 |
| 9 |  | 6 |  | Свободен для записи *R*1 |
| 10 |  |  |  |  |
| 11 |  |  | 6 |  |

Счетчики источников в первом цикле определяют использование *R*1, *R*2 и дважды *R*0. Конфликта нет, команда сложения также следует на исполнение. Счетчик приемников отметил *R*3 и *R*4.

Первая команда второго цикла также не встречает конфликтов, счетчики источников *R*0 и *R*1увеличиваются на «1». Счетчик приемников отметил *R*5. Вторая команда во втором цикле не сможет выйти на исполнение, т.к. она должна использовать содержимое приемника *R*4 незавершенной команды умножения.

Из таблицы видим, что команда сложения, использующая *R*4, сможет быть запущена на исполнение лишь на пятом цикле. С ней в паре запустится команда умножения, использующая операнд-источник *R*1.

В шестом цикле команда вычитания должна переписать *R*1, используя его как приемник. Возникает *WAR*-конфликт, исполнение команды задерживается. Оно становится возможным при завершении на восьмом цикле команды умножения. *SUB* уходит на исполнение только на девятом цикле.

Теперь рассмотрим возможность подмены команд и регистров, табл. 3.2.

Таблица 3.2.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| № цикла | команды | выдача | Результат  в регистре | комментарии |
| 1 | *MUL R*3*, R*0*, R*1  *ADD R*4*, R*0*, R*2 | 1  2 | -  - | *R*0*,R*1*-rd; R*3*-wr*  *R*0*,R*2*-rd; R*4*-wr* конфликтов нет |
| 2 | *ADD R*5*, R*0*, R*1  *ADD R*6*, R*1*, R*4 | 3  - | -  - | *R*0*,R*1*-rd; R*5*-wr*  *R*1*-rd, R*4 *(RAW)*  Замена команды |
| 3 | *MUL R*7*, R*1*, R*2  *SUB R*1*, R*0*, R*2 | 5  6 | 2 | *R*1*(WAR)* в 6 команде заменяется на *S*1. Свободен для чтения *R*4 |
| 4 |  | 4 | 1  3 | Свободен для чтения *R*4 |
| 5 |  |  | 6 |  |
| 6 |  |  | 4  5 |  |

Теперь порядок выдачи команд на исполнение не зависит от порядка выборки команд из памяти. Для команд первого цикла и первой команды второго цикла все остается подобно первому случаю. Вторая команда во втором цикле не может быть выдана на исполнение, но она заменяется первой командой третьего цикла. При такой подмене необходима предварительная проверка возможных конфликтов операндов выдаваемой и пропускаемой команд. Для этого включаются счетчики использования регистров на пропускаемую команду.

Во второй команде третьего цикла, для устранения *WAR*-конфликта, регистр *R*1 подменяется на скрытый регистр *S*1. В реальных компьютерах для подготовки очереди на исполнение используются только скрытые регистры, а их связь с заявленными регистрами устанавливается с помощью табличной памяти. Это позволяет избежать *WAR* и *WAW*-конфликтов.

Следует заметить, что представленные выше методы повышения производительности допустимы лишь в пределах базовых блоков программы. Базовым блоком считается линейная часть программы, ограниченная командами переходов. Таким образом, чем больше переходов в программе, тем короче код базового блока и «тяжелее» конструкция блока прогнозирования переходов.

Для сокращения количества переходов применяется предикация, то есть условное выполнение команд. Например, для *ARM*-кодирования с 32-разрядным форматом команды, все команды являются предикатными, т.е. имеют поле условия. Это показано на формате команды, рис. 3.10.

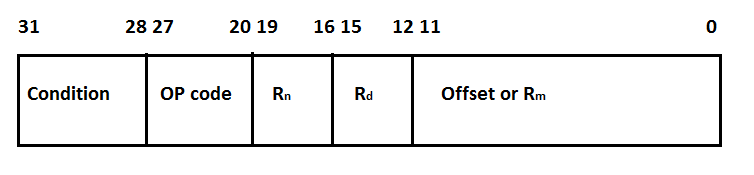


Рис. 3.10. Формат команды для *ARM* – кодирования.

У большинства команд код этого поля указывает безусловное выполнение, но есть команды, которые будут выполнены только при наличии условия. Такими условиями обычно служат флаги нулевого результата или знака. Предикация применяется во многих современных процессорных системах.

Иногда структура процессора позволяет взять на подмену первую команду следующего базового блока. Это называется спекулятивным исполнением. Но такой метод имеет ряд ограничений, как, например, обязательную возможность отмены результата спекулятивной команды, если она окажется, что она была не нужна. Это накладывает дополнительную нагрузку на аппаратную часть и компилятор. Поэтому, в отличие от предикации, такой метод используется реже.

Итак, мы видим, что в современных процессорах каждая ступень конвейера состоит, в свою очередь, из нескольких этапов. Если вернуться к прогнозированию переходов, то следует заметить, что каждая команда условного перехода все равно дойдет до этапа исполнения. Если же при этом окажется, что прогноз был неверен, то процессор возвращается к началу предварительной выборки, и конвейер отменяется.

Кроме того, очевидно, что конвейер, даже для суперскалярной архитектуры, не может полностью уберечь от простоев, если в обработку идет один программный поток. Простои вызываются конфликтами, а те, в свою очередь, или сложностью команды, или отсутствием операнда в кэш 1-го уровня. В последнем случае необходимо обращение ко второму уровню кэш, уже по внешней шине, как показано на рис. 3.11.

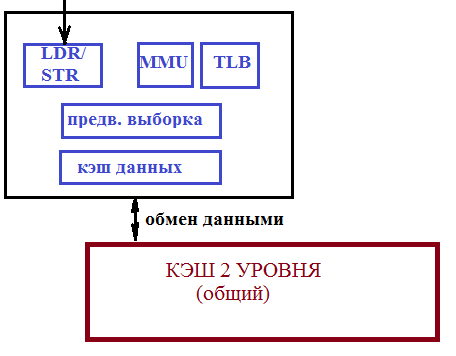


Рис. 3. 11. Подключение кэш второго уровня.

Наконец, простои могут быть вызваны при неверном прогнозировании условного перехода и при исполнении безусловного.

Сократить количество простоев может **внутрипроцессорная многопоточность,** т.е. возможность вызывать команды из разных потоков (задач). Реализация многопоточности может быть различна. Для простого конвейера, когда на обработку вызывается одна команда, возможны реализации мелкомодульной или крупномодульной многопоточности.

В **мелкомодульной многопоточности**, рис. 3.12, выборка команд из разных потоков идет «по кругу», не дожидаясь простоя.

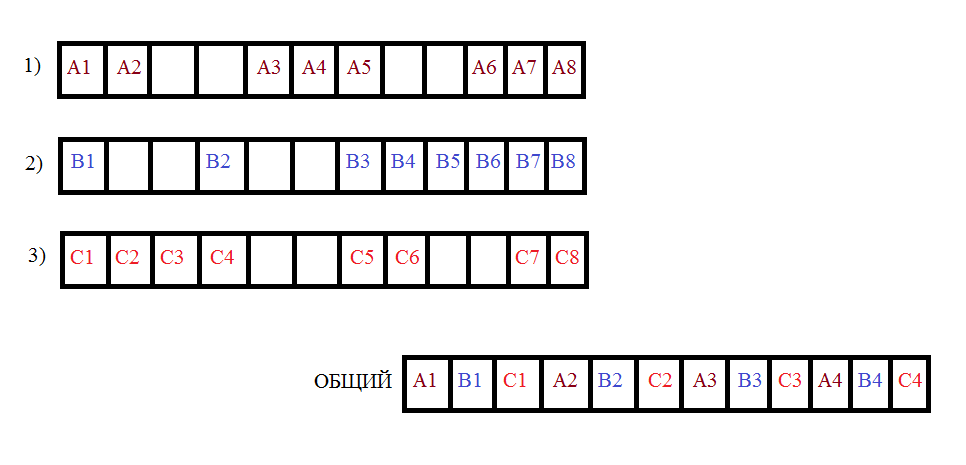


Рис. 3.12. Мелкомодульная многопоточность.

Каждой команде, поступающей на обработку, присваивается индекс потока, поэтому задачи не смешиваются, и путаницы не возникает. Если количество потоков соответствует количеству ступеней конвейера, то простои практически исключаются, но возникает иная трудность. Командам различных потоков нужны свои наборы регистров, поэтому под большое количество потоков нужен мощный внутренний ресурс. А это невыгодно.

В случае **крупномодульной многопоточности**, рис. 3.13, обращение к следующему потоку происходит только в случае возникновения простоя. Перед запуском каждого следующего потока конвейер очищается. Решение о запуске другого потока производится сразу после декодирования команды.

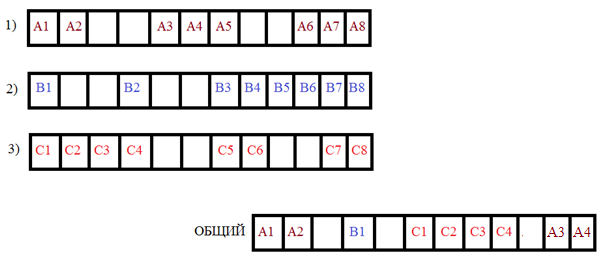
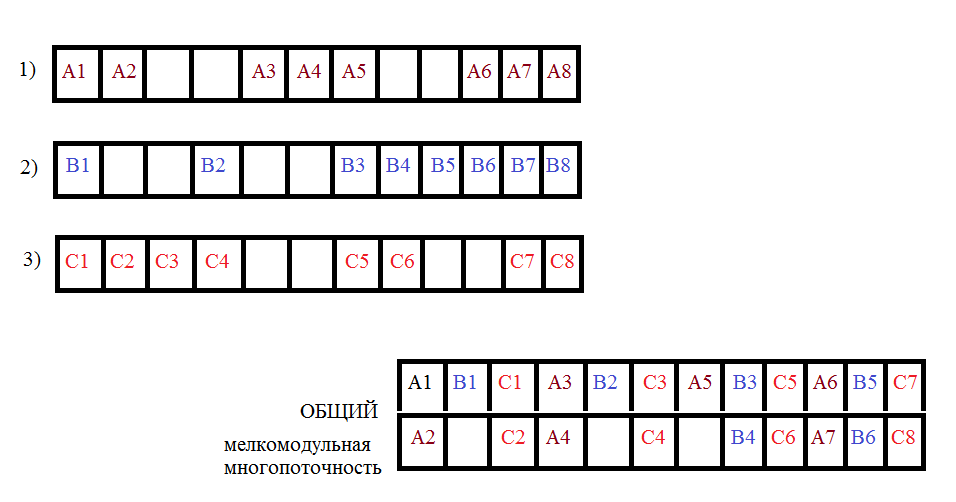
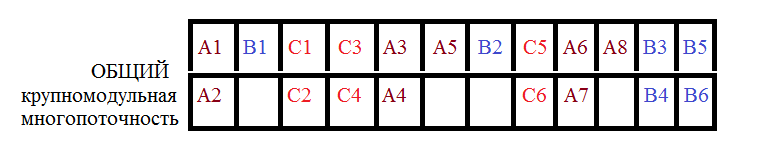


Рис. 3.13. Крупномодульная многопоточность.

Все вышеизложенное относилось к случаю выдачи одной команды за цикл. Если же рассмотреть минимальную суперскалярную архитектуру конвейера, то оба метода многопоточности не спасают от возможности простоев. В суперскалярных процессорах используется метод **синхронной многопоточности**. Это усовершенствованная крупномодульная многопоточность, позволяющая быстро переключать потоки и наиболее полно загружать функциональные блоки. Сравнение результатов всех перечисленных видов многопоточности приводится на рис. 3.14.





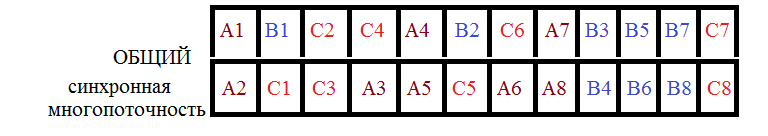


Рис. 3.14. Сравнение результатов различных видов многопоточности.

1. **ОБРАБОТКА ПРЕРЫВАНИЙ**

Мы определяли прерывание, как процесс, позволяющий приостановить выполнение основной программы на время выполнения подпрограммы. Обычно прерывания разделяют на аппаратные, программные и исключительные ситуации. Исключительные ситуации: ошибки, ловушки, аварийные завершения. Их также можно отнести к программным прерываниям, но они происходят в основном на этапе компиляции или отладки.

В зависимости от типа прерывания процессор должен установиться в определенный режим: *Supervisor* – программные прерывания или *reset* – выход на начальное состояние; *Abort* и *Undefined* – прерывания компилятора (ошибки);  *IRQ* и *FIQ* – аппаратные прерывания. Прерывания отладчика (ловушки) обычно организуются в программе отладчика с помощью установки флажка трассировки.

* 1. **Контроллер прерываний**

Аппаратные прерывания в процессорных системах линейки ARM (начиная с *ARM*7) обрабатываются с помощью контроллера прерываний *GIC – Generic Interrupt Controller*. Общая структура подключения *GIC* для двухъядерного процессора показана на рис. 4.1.

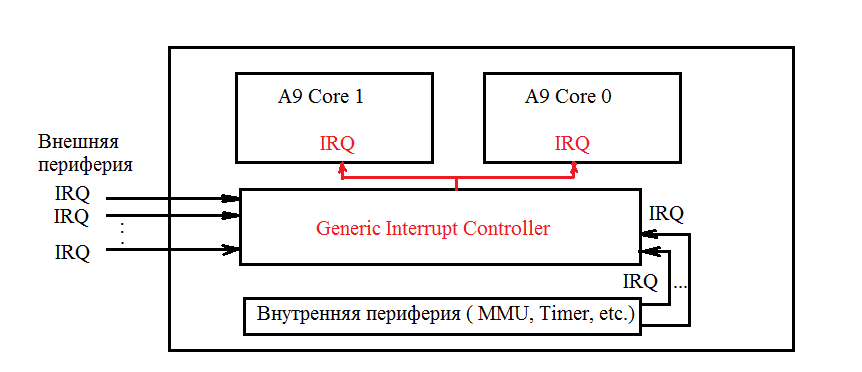


Рис. 4.1. Общая структура подключения *GIC* для *CortexA*9 с двухъядерной структурой.

Контроллер входит в структуру объединенного ядра процессора (например, *CortexA*9), принимая запросы как от внешних, так и от внутренних источников, способных генерировать запросы на аппаратные прерывания.

Для *CortexA*9 *GIC* способен обработать запросы от 255 источников. Все источники прерываний имеют свои идентификационные номера (*ID*). Интерфейс *CPU* способен распознать 1020 источников. При этом старшие идентификационные номера, с 1020 по 1023 зарезервированы. *ID* 1023 воспринимается, как ложный запрос.

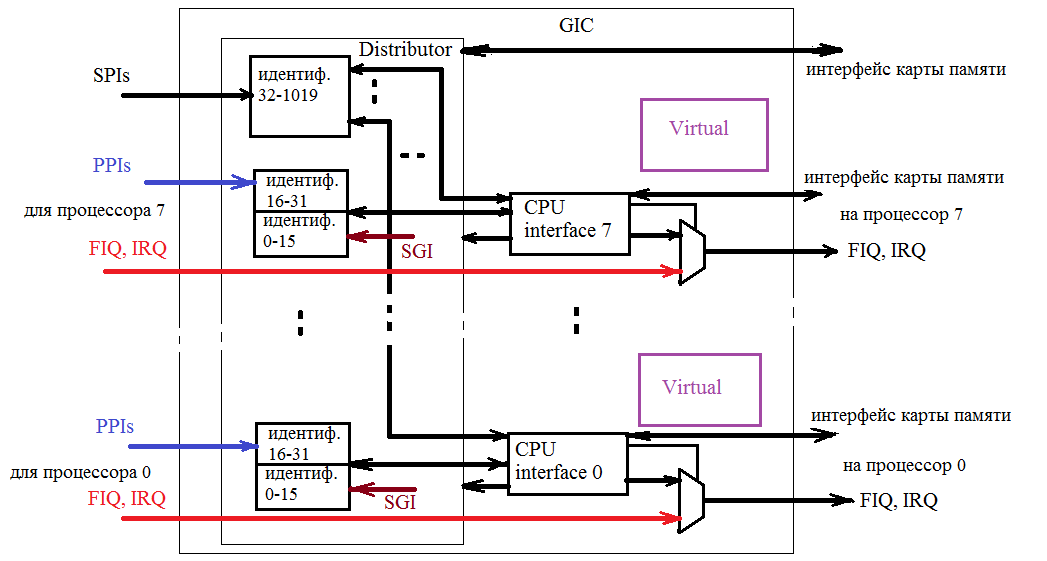


Рис. 4.2. Общая структура контроллера прерываний.

Структура контроллера включает два блока: распределитель и *CPU*-интерфейс.

Все запросы поступают на распределитель. Запросы делятся на три категории относительно их формирования и направления на ядро многоядерного процессора.

Внешние запросы, которые могут быть направлены на любое из ядер процессора, свободное на данный момент, общие - *Shared Peripheral Interrupts (SPI)*. Их *ID* с 32 по 1019.

Внешние запросы, следующие индивидуально на одно из ядер процессора, индивидуальные - *Private Peripheral Interrupts (PPI)*. Их *ID* с 16 по 31.

Все внешние запросы могут определяться как по уровню поступающего сигнала ***(Level-sensitive)****,* так и по его изменению, с помощью регистра краевого захвата ***(Edge-triggered)*.**

Внутренние запросы, генерируемые устройствами, входящими в МПС, *Software Generated Interrupt (SGI)*. Их *ID* с 0 по15. Запрос на прерывание этого типа всегда определяется с помощью краевого захвата. Информация о таких запросах изначально пишется в регистр *GICD\_SGIR*.

Для прерываний по запросам *PPI* и *SGI* необходимо точно определить свое направление. При этом такие запросы, имеющие одинаковые *ID*, могут направляться на разные ядра процессора. Для разрешения этих ситуаций в структуре *GIC* имеются табличные блоки памяти прерываний, которые идентифицируют единственный вариант комбинации направления для *ID*. Кроме таких блоков есть блоки регистровой памяти для создания копий информации для каждого ядра.

Все прерывания также разделяются на две группы по уровню защиты.

Группа 0 – защищенные прерывания. *FIQ* – входят в группу всегда, *IRQ* - только в случае защищенного программного обеспечения (ПО) процессора.

Группа 1 – незащищенные прерывания. Все *IRQ*.

Определение группы производится в распределителе. Контроль за распределением запросов по группам устанавливается согласно записи в регистре *GICD\_IGROUPR*. При этом учитывается тип процессора и степень защиты ПО. Если, например, процессорное ПО не имеет защиты, то в группу 0 попадают только *FIQ* запросы, а все *IRQ* попадают в группу 1. Для защищенного ПО запросы выделенных групп распределяются в блоке интерфейса.

Для обработки каждой группы запросов с целью выявления их приоритетов и дальнейшего продвижения подключается отдельный блок регистров. Приоритет запросов группы 0 (безопасные прерывания) всегда выше, чем приоритет группы 1 (небезопасные прерывания). Наивысшим считается приоритет 0, самый низкий приоритет – 255. Даже если незащищенное прерывание имеет высокий приоритет, он будет преобразован путем сдвига в специальных регистрах таким образом, что в старшем разряде запишется 1. Реализация такого преобразования иллюстрируется на рис. 4.3.

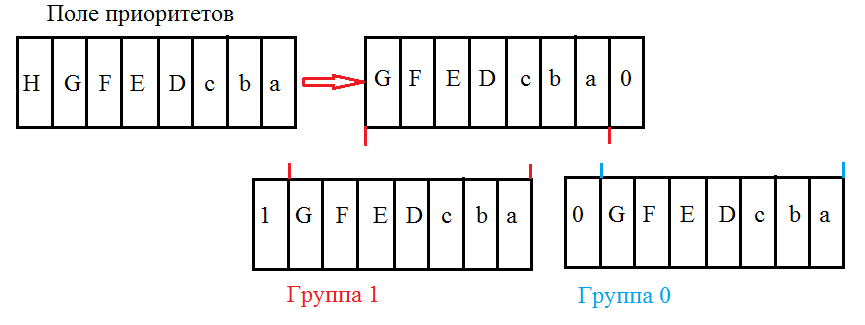


Рис. 4.3. Преобразование приоритетов для запросов по группам защищенности.

Разделенные по группам запросы обрабатываются в зависимости от информации, которую необходимо записать в регистрах распределителя, изображенных на рис. 4.4. Общая часть имени регистров распределителя – *ICD(Interrupt Controller Distributer).*

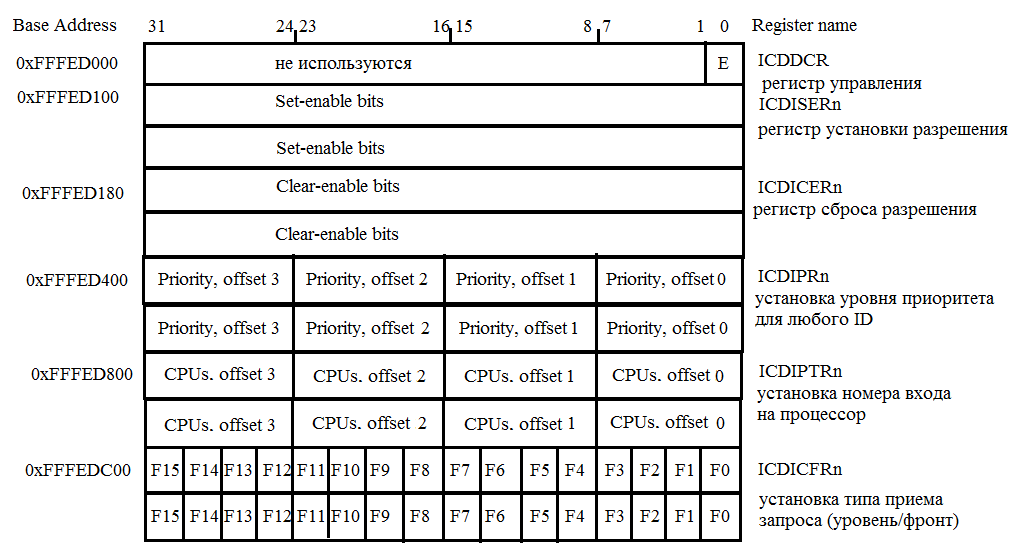
****

Рис. 4.4. Регистры распределителя.

***Distributor Control Register (ICDDCR)*** – регистр управления распределителя. Из его 32 разрядов активен только 0-й. В этом разряде устанавливается принципиальная возможность работы распределителя.

***Interrupt Set Enable Registers (ICDISERn)*** – регистр установки разрешения принять запрос с данным *ID*. В формат этого регистра входят 32-а 32-разрядных регистра, что занимает 256 адресных ячеек.

Аналогичный формат имеет регистр сброса разрешения на прием запроса с данным *ID -* ***Interrupt Clear Enable Registers (ICDICERn).***

***Interrupt Priority Registers (ICDIPRn)* –** регистр установки уровня приоритета для любого *ID*, предоставляет возможность расставить приоритеты для каждого из 256 обрабатываемых запросов. Приоритеты определяются в каждой группе в зависимости от *ID*. В зависимости от приоритета запросы ставятся в очередь на ожидание или получают статус активного. Формат регистра – 256 32-разрядных регистров, или 1024 адресных ячейки.

Такой же формат и у регистра ***Interrupt Processor Targets Registers* *(ICDIPTRn)*** – регистра установки номера входа на процессор.

***Interrupt Configuration Registers (ICDICFRn)*** – регистр установки типа приема запроса: по уровню, или по фронту, используя регистр краевого захвата. Для каждого *ID* выделяется 2 бита. Младший не используется, а в старшем бите устанавливается 0 для приема запроса по уровню или 1- для считывания с регистра краевого захвата. Только запросы, принимаемые по уровню, могут опрашиваться повторно и сниматься с очереди, если уровень не обнаружен.

**Итог.**

Распределитель принимает запросы, определяет их приоритеты и направляет в интерфейсный блок.

Функции распределителя:

- программирование **возможности** пересылки **любого** прерывания на C*PU*-интерфейс *(ICDDCR);*

- программирование возможности или невозможности приема **данного** прерывания *(ICDISERn, ICDICERn)*;

- установка уровня приоритета данного прерывания *(ICDIPRn)*;

**-** установка номера входа на процессор для данного прерывания *(ICDIPTRn)*;

- определение типа приема запроса: по уровню или по фронту, с помощью краевого захвата *(ICDICFRn).*

Кроме того, распределитель:

- определяет принадлежность прерывания к группе уровня защищенности. (Группа 0 – защищенные прерывания, на вход процессора допускаются как *IRQ*, так и *FIQ* запросы. Группа 1 – незащищенные прерывания – только *IRQ*);

- осуществляет пересылку программно-генерируемых запросов, *SGI*, на один или несколько процессорных входов;

- определяет состояние данного прерывания;

- запускает механизм программной установки или сброса подвешенного запроса.

С выхода распределителя утвержденные к исполнению запросы поступают в регистры CPU-интерфейса, рис. 4.5. Общая часть имени этих регистров *ICC (Interrupt Controller CPU-interface).*

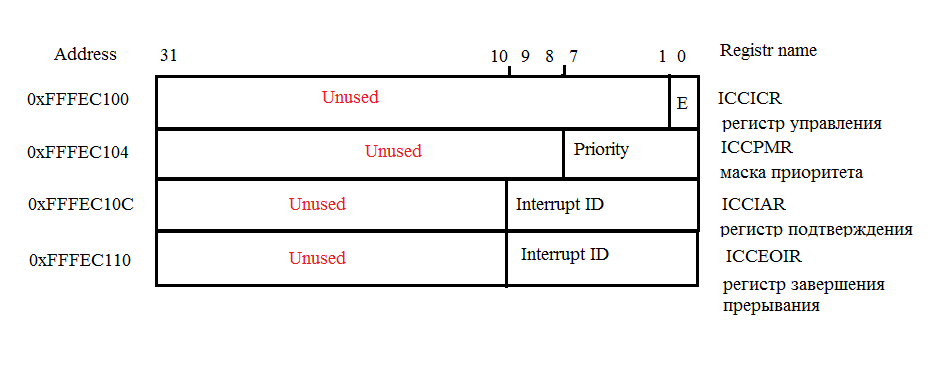


Рис. 4.5. Регистры *CPU* интерфейса.

***CPU Interface Control Register (ICCICR)*** – регистр управления. Подобен регистру управления распределителя.

***Interrupt Priority Mask Register (ICCPMR)*** – регистр маски приоритета. На младшем байте этого регистра записывается приоритет запроса, получившего статус активного. Если следом за ним поступает запрос от того же устройства ( одинаковый *ID*), то приоритет вновь поступившего запроса сравнивается с имеющейся маской. Запрос, имеющий приоритет выше, остается в статусе активного, а второй ставится на ожидание (подвешивается) в структуре блока *CPU*-интерфейса.

***Interrupt Acknowledge Register (ICCIAR)*** – регистр подтверждения отсылки запроса на процессор. В 10 младших разрядов записывается *ID* прошедшего на процессор запроса. В зависимости от этого номера формируется код прерывания.

По завершению подпрограммы прерывания в регистр завершения прерывания - ***End of Interrupt Register (ICCEOIR)*** должен быть записан *ID* его запроса.

**Итог.**

*CPU*-интерфейс формирует:

- возможность посылки запроса;

- определение порядка вытеснения прерывания для процессора;

- распределение степени приоритета подвешенных прерываний для процессора;

- подтверждение прерывания;

- индикацию завершения обработки прерывания процессором.

Запрос, поступающий на контроллер, проходя через проверки в его блоках, может находиться в одном из 4-х различных состояний:

- неактивный - запрос только что поступил, он не активен, или поставлен в очередь на ожидание;

- ожидающий в очереди – запрос опознан и принят, но поставлен на ожидание в блоке распределителя к выходу на *CPU*-интерфейс;

- активный – запрос, поступивший на контроллер, передан процессору и опознан, но обслуживание его не завершено;

- активный и поставленный на ожидание – в случае, когда контроллер получает подряд идущие запросы от одного и того же источника, процессор обслуживает запрос, имеющий более высокий приоритет, а контроллер на это время ставит на ожидание в *CPU*-интерфейсе следующий запрос от этого же источника.

Переходы из одного режима в другой управляются с помощью конечного автомата, рис. 4.6.

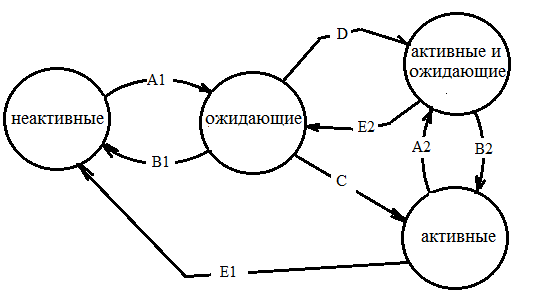


Рис. 4.6. Граф переходов конечного автомата состояний запросов на прерывания.

При рассмотрении переходов будем обращаться только к запросам от внешних источников.

Случаи переходов *A*. Состояние ожидания. На этапе нахождения в распределителе запрос ставится в очередь, если периферия подтверждает запрос (*A*1). На этапе нахождения в *CPU*-интерфейсе запрос, оказывающийся ниже приоритетом поступившего следом запроса от того же источника, ставится в очередь и отмечается при этом в регистре очереди на ожидание (*A*2).

Случаи переходов *B*. Удаление состояния, ожидающего в очереди. Запрос может быть снят с очереди в случае снятия подтверждения периферией и перехода в неактивное состояние (*B*1), или в случае подтверждения необходимого приоритета, сохранении «1» в регистре краевого захвата и установки бита сброса с очереди. В этом случае запрос переходит на обслуживание, т.е. становится активным (*B*2).

Случай перехода *С*. При наличии необходимого приоритета в регистре *CPU*-интерфейса.

Случай перехода *D*. Повторный запрос от того же источника. Прерывание разрешено, происходит проверка приоритета. На время анализа приоритетов запрос ожидает. Если за время анализа запрос будет снят, произойдет вариант *B*1. Если приоритет этого запроса окажется выше обрабатываемого от того же источника, вариант *B*2.

Случаи перехода *E*. Обработка закончена, прочтение регистра окончания обработки дает *E*1. Запрос от другого источника с более высоким приоритетом выталкивает запрос – *E*2.

1. **УСТРОЙСТВА ВВОДА-ВЫВОДА**

В предыдущих лекциях мы рассматривали функционирование части микропроцессорной системы, касающейся взаимодействия процессорного блока с блоком памяти. При этом устройства ввода-вывода упоминались, но подробно не рассматривались. Обратимся снова к структуре микропроцессорной системы, рис. 5.1.

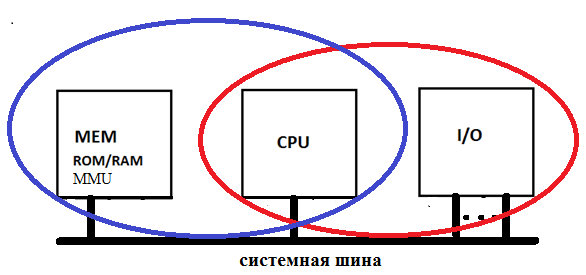


Рис. 5.1. Общая структура микропроцессорной системы.

Любой обмен данными между процессорной системой и внешним устройством происходит через устройства ввода-вывода. Внутри процессорной системы обмен производится между регистрами процессора и порта ввода-вывода.

Основным моментом этого взаимодействия является готовность внешнего устройства к приему или к передаче данных. Готовность фиксируется в регистре состояния порта, этот регистр процессор только читает. Регламент обмена записывается процессором в регистре управления порта. Обмен данными между процессором и портом производится параллельным способом. Обмен, между портом и внешним устройством может быть как последовательным, так и параллельным.

На большие расстояния данные всегда передаются по последовательной схеме, в линии. Примерами получения такого обмена являются *UART, USART, USB, EtherNet* и т.п. На короткие расстояния, зачастую ограниченные объемом процессорной системы, может также осуществляться последовательный обмен *(SPI, I2C).* Но любой последовательный обмен требует дополнительных преобразований, для которых необходим аппаратный и временной ресурс. Поэтому часто во встроенных системах используются параллельные порты. Мы рассматривали на практике параллельный порт *ARM*-процессоров. Он крайне прост. Для встраиваемых систем под *Intel* существует интерфейс 8255*A*. Схема эта была разработана более 40 лет назад, но, модернизированная, применяется и поныне.

Теперь рассмотрим более детально примеры перечисленных выше интерфейсов.

* 1. **. Устройства, обеспечивающие передачу информации на большие расстояния**
     1. ***Последовательный асинхронный приемопередатчик UART***

Это один из немногих устройств, использующихся с середины прошлого века. Разумеется, устройство постоянно модернизируется, но принцип его функционирования неизменен. В *UART* используется побитовая передача информации. Скорость передачи измеряется в битах в секунду – **бодах**. Минимальная скорость – 300 бод, максимальная чуть меньше миллиона – 921600 бод. Набор стандартных скоростей включает 13 позиций. Для перехода на выбранную скорость обмена, до его осуществления в специальные регистры-делители записывается коэффициент деления тактовой частоты системы. Только после этого в регистр управления записывается регламент обмена.

Информация, записанная в регистре управления, позволяет:

- определить количество разрядов для передаваемого символа (от 5-ти до 8-ми стандартно, в последних модернизациях - 9);

- фиксировать наличие (или отсутствие) контроля передаваемых символов, а также указывать тип контроля (паритет или непаритет). В современных *UART* контроль производится путем сложения по модулю 2 всех передаваемых бит символа;

- количество стоповых бит в посылке. Посылка всегда обрамляется специальными метками начала и окончания. Начальный бит всегда «0», он опознается в точке, расположенной по времени середины его удержания после рассоединения – «1». Бит окончания, стоповый, всегда «1». Он также определяется по середине его удержания, но иногда, для разделения посылок добавляют второй стоповый бит. Таким образом, стоповых бит может быть 1 или 2.

Кроме указанной информации в регистре управления определяется наличие соединения в принципе, а также тип действия, производимого в данный момент – обращение к регистрам данных или обращение к регистрам-делителям.

Регистры данных смешанного типа. Для передатчика – параллельно-последовательный, для приемника – последовательно-параллельный.

Кроме перечисленных регистров, в структуру *UART* входят регистры состояния линии и модема, регистр разрешения прерываний и регистр-идентификатор прерываний.

В настоящее время *UART*, также как и его синхронный вариант, *USART*, в большинстве случаев использования в аппаратуре, не имеют собственный разъемов, а включаются через *USB*.

**5.1.2. *Интерфейс USB (Universal Serial Bus - Универсальный Последовательный Интерфейс)***

Предназначен для подключения периферийных устройств к персональному компьютеру. Позволяет производить обмен информацией с периферийными устройствами на трех скоростях (спецификация *USB 2.0*):

Низкая скорость (*Low Speed* - LS) - 1,5 Мбит/с;

Полная скорость (*Full Speed* - FS) - 12 Мбит/с;

Высокая скорость (*High Speed* - HS) - 480 Мбит/с.

    Для подключения периферийных устройств используется 4-жильный кабель: питание +5 В, сигнальные провода *D+* и *D-*, общий провод.

*Структура USB*

Структура *USB* показана на рис.5.2. В нее входят:

- хост –основное ведущее устройство, как пример - персональный компьютер;

- хаб – концентратор. Содержит контроллер и повторитель. Регистры хаба-контроллера связываются с хостом и обеспечивают опознавание функции, и ее связь с хостом. Хаб-повторитель обеспечивает энергетический режим работы шины;

- функция – оконечное устройство, подключаемое к хосту. Всегда ведомое.

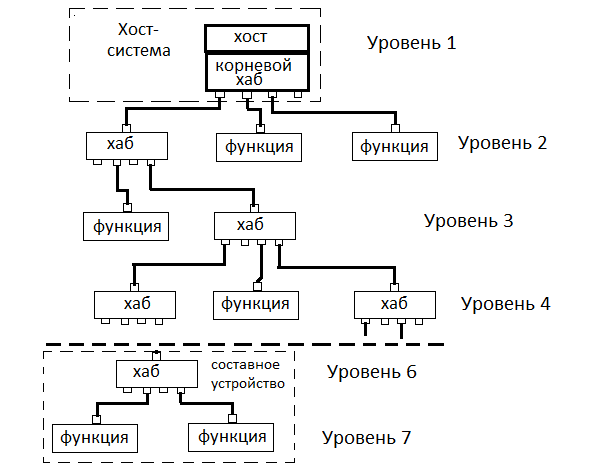


Рис. 5.2. Структура *USB.*

Порт хаба или функции, подключаемый к хабу более высокого уровня, называется восходящим портом (*upstream port*), а порт хаба, подключаемый к хабу более низкого уровня, или к функции, называется нисходящим портом(*downstream port*).

Все передачи данных по интерфейсу инициируются хостом, однако приемником или передатчиком может быть как хост, так и функция.

Функция не может сама инициировать обмен, весь обмен управляется хостом. Контроллер хоста периодически опрашивает подключенные и резервные устройства на предмет обмена, рассылая пакет начала кадра и составляет списки заявок. Если устройство готово к обмену, между ним и хостом организуется канал, т.е. программно поддерживаемое соединение. Так как функции представляют собой сложные схемы, в которых выдачей и приемом информации могут заниматься различные узлы, то внутри функции выделяются конечные точки. Таким образом, канал организуется между хостом и конечной точкой. Обмен всегда производится между хостом и конечной точкой при поддержке контроллера хаба.

Данные по *USB* шине всегда передаются младшими битами вперед.

Передача пакетная.

В интерфейсе *USB* испольуется несколько разновидностей пакетов:

**-пакет-признак** (*token paket*) описывает тип и направление передачи данных, адрес устройства и порядковый номер конечной точки (КТ - адресуемая часть USB-устройства).

**-пакет с данными** (*data packet*) содержит передаваемые данные;

**-пакет согласования** (*handshake packet*) предназначен для сообщения о результатах пересылки данных.

**- пакет начала кадра**(*start of frame packets*) содержит номер передаваемого кадра.

*USB* пакет состоит из нескольких полей.

Любой тип пакета должен содержать поле синхронизации, поле идентификатора пакета и поле конца пакета.

Пакет-признак кроме этих полей должен содержать поле адреса устройства, поле адреса конечной точки и поле контроля (циклический контроль по избыточности).

Пакет с данными должен содержать поле данных и поле контроля.

Пакет согласования содержит только три основных поля.

Пакет начала кадра должен содержать поле номера кадра и поле контроля.

Описание полей пакетов.

*·   Поле синхронизации (Sync Field)*

Все пакеты должны начинаться с поля синхронизации. Поле синхронизации имеет размер 8 бит для низкоскоростных и полноскоростных устройств или 32 бита для высокоскоростных устройств и используется для подсинхронизации тактового генератора, встроенного в *USB* контроллер. Последние два бита поля синхронизации являются маркером, который используется для идентификации конца области синхронизации и начала *PID* поля.

·   *Поле идентификатора пакета (PID Field)*

Идентификатор пакета следует непосредственно после поля синхронизации в каждом передаваемом *USB* пакете. *PID* состоит из 4-битного поля типа пакета, следующего за 4-битным собственным проверочным полем .

·         *Поле адреса устройства (Addr Field)*

Поле адреса используется для идентификации направления к устройству, в которое направлен текущий пакет. Размерность поля - 7 бит, что позволяет адресовать 127 уникальных *USB* устройств. После сброса или включения питания, адрес устройства устанавливается по умолчанию в 0 и должен быть запрограммирован хостом в ходе процесса присвоения номера. Адрес 0 (заданный по умолчанию) зарезервирован для вновь подключаемых устройств и не может быть назначен для нормальной работы.

·         *Поле адреса конечной точки (Endpoint Field)*

Поле адреса конечной точки имеет размерность 4 бита и позволяет размещать в устройстве до 16 конечных точек. Конечная точка это логический адрес, образованный внутри устройства. Связь хоста и конечной точки производится по каналу, представляющему собой логическое образование.

·         *Поле номера кадра (Frame Number Field)*

Поле номера кадра представляет собой 11-битное поле, которое инкрементируется хостом при инициировании нового кадра. Поле номера кадра начинается заново с нуля при достижении максимального значения 0*х*7*FF*, и используется только для *SOF* маркеров в каждом начале кадра.

·         *Поле циклического контроля по избыточности (CRC)*

Циклический контроль по избыточности (*CRC*) используются для защиты всех полей кроме *PID* в маркерах и пакетах данных. Защита Маркера и пакета данных обеспечивает 100% нахождение всех одиночных и двойных битовых ошибок. Для маркеров предусмотрено 5-битное поле *CRC*, которое используется для защиты полей *ADDR* и *ENDP* пакетов *IN*, *SETUP,OUT* или поле отметки времени маркера *SOF*. Для пакета данных используется 16-битный полином, кодирующий всё поле данных пакета.

*·         Поле конца пакета (EOP)*

Поле конца пакета представляет собой сигнал окончания пакета и устанавливается на шине путем выставления сигнала уровня 0 "*Single* *Ended Zero* " *SE*0 в течение двух битовых интервалов следующего за 1-м состоянием, установленным на время одного битового интервала.

Описание типов USB пакетов

*Пакеты-признаки (Token Packets)*

Существуют три типа таких пакетов:

1.      ***In*** - информируют *USB* устройство, что хост хочет читать данные из устройства

2.      ***Out*** - информирует *USB* устройство, что хост хочет передавать данные в устройство

3.     ***Setup*** - используются для обозначения начала управляющего (*Control Transfer*) типа передачи данных

Все пакеты-признаки имеют следующий формат:

*SYNC PID ADDR ENDP CRC5 EOP*

*·         Пакеты данных (Token Packets)*

Существуют два типа пакетов данных, каждый из которых способен содержать до 1024 байтов данных.

*DATA*0 и *DATA*1

У высокоскоростных устройств пакеты данных имеют два других идентификатора: *DATA*2 и *MDATA*. Пакет данных имеет следующий формат:

*SYNC PID DATA CRC16 EOP*

Максимальная полезная загрузка:

- для низкоскоростных устройств составляет 8 байт.

- для полноскоростных устройств составляет 1023 байта.

- для высокоскоростных устройств составляет 1024 байт.

Данные всегда посылаются целым числом байт.

*Пакеты согласования (Handshake Packets)*

Существуют три типа пакетов согласования, структура которых имеет только PID поле:

- ***ACK*** - подтверждение того, что пакет был успешно принят

- ***NAK*** - информирует, что устройство в данный момент не может принимать либо отправлять данные. Также используется в *Interrupt* транзакциях для информирования хоста, что устройство не имеет новых данных для передачи. Хост никогда не может выдавать ответ *NAK. NAK* используется в целях управления потоком данных.

- ***STALL*** - указывает, что устройство неспособно передавать или получать данные, и требуется вмешательство хоста для снятий состояния останова. Как только конечная точка устройства остановлена, устройство должно продолжить возвращать *STALL*, пока условие, вызвавшее останов не будет удалено с помощью вмешательства хоста. Хосту запрещено возвращать *STALL*.

Пакет подтверждения имеет следующий формат:

*SYNC PID EOP*

*Пакеты начала кадра (Start of Frame Packets)*

*SOF* пакет состоит из 11-ти битного номера кадра и генерируется хостом каждую 1*ms* ± 500*ns* для полноскоростной шины и каждые 125 µ*s* ± 0.0625 µ*s* для высокоскоростной шины.

Пакет начала кадра имеет следующий формат:

*SYNC PID Frame Number CRC*16  *EOP*

В зависимости от типа передаваемых данных, предъявляемых требований к скорости обработки, задержки доставки и т.п. определены следующие типы передач.

**Управляющие передачи**. Используются для конфигурирования устройств во время подключения и выполнения других специфических функций над устройством, включая организацию новых каналов.

**Прерывания**. Используются для спонтанных, но гарантированных передач с гарантированными скоростями и задержками. Используются обычно для передачи введенных данных от клавиатуры или сведений об изменении положения указателя мыши, в устройствах обратной связи, и т.д.

**Групповая пересылка**. Используется для гарантированной передачи данных больших объемов без предъявленных требований к скоростям и задержкам. Занимает под себя всю свободную пропускную способность шины. В любой момент доступная полоса может быть урезана при необходимости осуществления передач других видов с более высоким приоритетом, или добавлена, при освобождении другими устройствами. Обычно такие передачи используется между принтерами, сканерами, накопителями и др.

**Изохронные передачи**. Используются для потоковой передачи данных в реальном времени. Резервируют определенную полосу пропускания шины, гарантируют определенные величины задержек доставки, но не гарантируют доставку (в случае обнаружения ошибки повторной передачи не происходит). Передачи этого вида используются для передачи аудио и видео трафика.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
|  |  |  |  |  |

*Типы каналов***.**

      Каналы сообщений. Являются двунаправленными каналами. Возникает канал при отсылке хостом запроса в устройства, и управляет передачей только хост. Каналы сообщений используется только для передач управляющего типа.

       Потоковые каналы. Такие каналы являются однонаправленными. В потоковых каналах передачи могут контролироваться не только хостом, но и устройством. Потоковые каналы используются для передач данных следующих типов: прерывание, групповая пересылка, изохронная.

*Приоритеты передач по USB-шине.*

Все операции по передаче данных инициируются хост-системой независимо от того, принимает ли она данные или пересылает в периферийное устройство. Все операции, которые необходимо выполнить, хранятся в виде четырех списков по типам передач:

- изохронные передачи;

- передачи прерываний;

- передачи управляющих команд;

- передачи данных больших объемов.

Списки постоянно обновляются новыми запросами. Планирование операций по передаче информации в соответствии с упорядоченными в виде списков запросами выполняется  хостом с 1-миллисекундным интервалом. В начале каждого такого интервала хост посылает по шине пакет *SOF* (*StartOf Frame* -  начало кадра), после чего начинается обслуживание запросов из списка изохронных передач (т.к. они имеют наивысший приоритет).

После того, как все запросы из этого списка будут обслужены, хост-система переходит к списку операций передачи прерываний, затем к списку запросов на передачу данных большого объема.

По истечении 90% указанного 1-миллисекундного интервала хост автоматически переходит к обслуживанию запросов на передачу управляющих команд независимо от того, успел ли он полностью обслужить другие три списка или нет.

Тем самым гарантируется, что управляющим передачам всегда будет выделено не менее 10%  пропускной способности шины. Если передача всех управляющих пакетов будет завершена до истечения выделенной для них доли интервала планирования, то оставшееся время будет использовано хостом для передачи данных большого объема ( до конца указанного 1-миллисекундного интервала).

Таким образом:

- изохронные передачи гарантированно получают 90% пропускной способности шины;

- передачи прерываний занимают оставшуюся часть изохронных операций часть этой 90-процентной доли;

- под передачу данных большого объема выделяется все время, оставшееся после изохронных передач и передач прерываний (по-прежнему в рамках 90% доли пропускной способности);

- управляющим передачам гарантируется 10% пропускной способности;

- если передача всех управляющих пакетов будет завершена до истечения выделенного для них 10% временного интервала, то оставшееся время будет использовано для передачи данных большого объема.

**5.2. Устройства, обеспечивающие передачу информации на короткие расстояния**

Мы рассмотрели шину USB, используемую для связи различных цифровых устройств.

Определяя понятие «канал», мы дали определение конечной точки как блока функции, который будет участвовать в обмене информацией. Таким образом, мы объявили функцию сложным устройством, внутри которого необходимо также регулировать обмен информацией. Действительно, любая современная цифровая система нуждается в устройстве обмена между ее составляющими.

К устройствам, поддерживающим такой обмен, относятся шины SPI и *I*2*C*, а также различные внутренние параллельные интерфейсы.

**5.2.1. *Интерфейс (шина) SPI*.**

## Последовательный периферийный интерфейс (шина) служит для обмена информацией между микросхемами. Построение шины основано на строгой иерархии «ведущий – подчиненные», рис. 5.3.

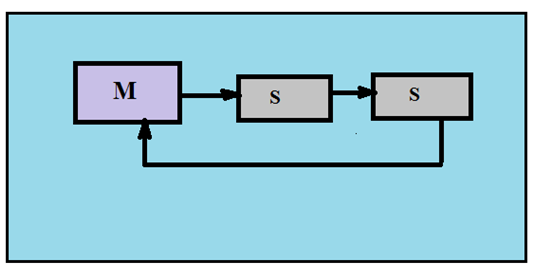


Рис. 5.3. Принцип построения шины *SPI*.

В качестве ведущего обычно выступает микроконтроллер, а подчиненными могут быть различные схемы памяти, таймеры, АЦП и ЦАП и т.п. Организация связи представляет собой цепь из последовательных регистров, выделяемых в пространстве объединяемых микросхем, рис. 5.4.

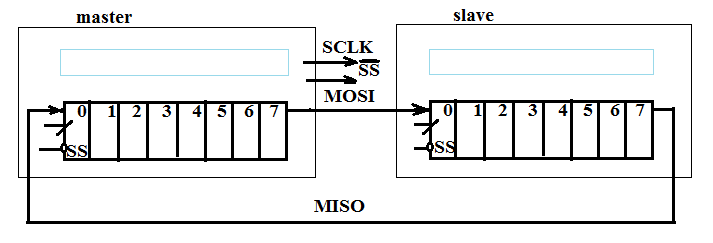
****

Рис. 5.4. Организация связи ведущий-ведомый в *SPI*/

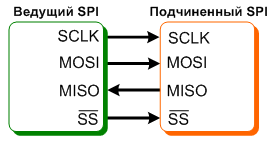
Для выбора ведомого, ведущее устройство посылает в его направлении низкий уровень по линии *slave select* (*SS*).

## Передача и прием ведутся одновременно, пакетами. Чаще всего длина пакета составляет 8 бит, но это не является обязательным условием. Сдвиг производится по тактовой частоте, генерируемой ведущим устройством.

## Ведомые устройства используют синхросигнал для определения моментов изменения битов на линии данных, при этом ведомые устройства никак не могут влиять на частоту следования битовых интервалов. Как в ведущем устройстве, так и в ведомом устройстве имеется счетчик импульсов синхронизации (битов). Счетчик в ведомом устройстве позволяет последнему определить момент окончания передачи пакета. Счетчик сбрасывается при выключении подсистемы SPI, такая возможность всегда имеется в ведущем устройстве. В ведомом устройстве счетчик обычно сбрасывается деактивацией интерфейсного сигнала SS.

Так как действия ведущего и ведомого устройства тактируются одним и тем же сигналом, то к стабильности этого сигнала не предъявляется никаких требований, за исключением ограничения на длительность полупериодов, которая определяется максимальной рабочей частотой более медленного устройства. Это позволяет использовать *SPI* в системах с низкостабильной тактовой частотой, а также облегчает программную эмуляцию ведущего устройства.

Самым простым будет подключение только одного ведомого устройства, рис. 5.5.

  
Рис. 5.5. Подключение одного ведомого устройства.

Если в структуре обмена присутствует несколько ведомых устройств, то их подключения к ведущему возможны одним из 2-х способов: независимое подключение, рис. 5.6, или каскадное подключение, рис. 5.7.

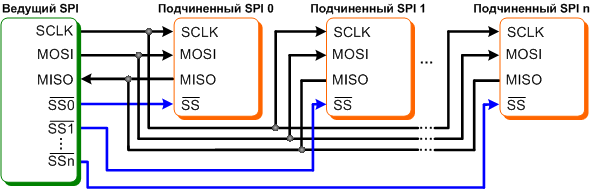


Рис. 5.6. Независимое подключение к шине SPI.

При таком подключении ведущему необходимо иметь несколько выходов для формирования сигналов выбора подключаемого ведомого. Если это нежелательно, то используется каскадное подключение.

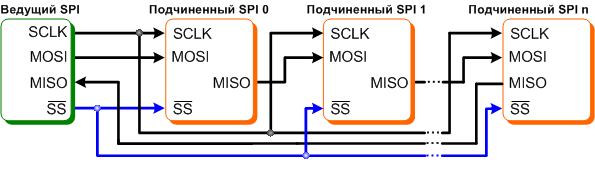


Рис. 5. 7.Каскадное подключение к шине SPI (возможно при совместимости ведомых устройств по режимам работы).

*Режимы работы SPI*

Функционирование любого регистра сдвига предполагает установку информации в каждом разряде с некоторой задержкой относительно ее выборки. Любое изменение уровня импульса синхронизации (фронт или спад) происходит за какое-то конечное время, которое не должно превышать время задержки триггера. Поэтому, для предотвращения сбоев при сдвиге информации, задержку триггера увеличивают (преимущественно за счет увеличения выходной емкости).

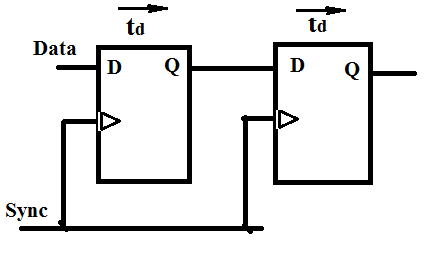


Рис. 5.8. Принцип организации этапов выборки и установки в *SPI*.

В регистрах шины SPI выборка и установка производятся по противоположным фронтам импульса синхронизации, рис 5.8. Исходя из этого, SPI имеет 4 режима работы, рис. 5.8.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Режим 0 | | Режим 1 | Режим 2 | Режим 3 |
| CPOL=0 | | CPOL=1 | CPOL=0 | CPOL=1 |
| CPHA=0 | | CPHA=0 | CPHA=1 | CPHA=1 |
| tabl2_pic1 | tabl2_pic2 | | tabl2_pic3 | tabl2_pic4 |

Рис. 5.8. Временная диаграмма первого цикла синхронизации.

*CPOL* – исходный уровень полярности сигнала синхронизации;

*CPHA* – исходная фаза цикла обмена.

*Преимущества интерфейса SPI.*

Полнодуплексная передача данных по умолчанию.

Возможность произвольного выбора длины пакета.

Возможность использования в системах с низкостабильной тактовой частотой.

Адрес ведомого устройства не передается в структуре пакета.

В отличие от параллельных интерфейсов имеет только 4 вывода.

*Недостатки.*

Ведомое устройство не может управлять потоком данных.

Нет подтверждения приема данных со стороны ведомого устройства (ведущее устройство может передавать данные «в никуда»).

Нет определенного стандартом протокола обнаружения ошибок.

Наличие множества вариантов реализации интерфейса.

Отсутствие поддержки горячего подключения устройств.

Необходимо больше выводов, чем для интерфейса *I*2*C*.

***5.2.2. Интерфейс (шина I2C - Inter-integrated circuit bus)***

Сетевой последовательный интерфейс, осуществляющий связь ведущих и ведомых по 2-проводной схеме. Два провода – сигнальные, существует еще и провод подключения земли, но он считается вспомогательным, не участвующим в передаче сообщений. К шине может быть подключено до 128 устройств, каждое из которых имеет свой адрес, если это не превысит предельную емкость (400 пф).

По скорости передачи различают три режима работы шины: стандартный (*S*) - 100Кбит/сек, быстрый (*F*)– 400Кбит/сек и высокоскоростной режим (*Hs*) – до 3,4Мбит/сек. Первые два режима функционируют практически одинаково. Передача пакетная, объем пакета 8 информационных бит, передача в пакете ведется старшим битом вперед. Устройство, инициирующее обмен – *master* (ведущее), устройство, назначаемое мастером для обмена – *slave* (ведомое). Как *master,* так и *slave* могут выступать в роли приемника или передатчика. Для режимов *S* и F в каждый момент времени информация передается только одним устройством и только в одну сторону.

Электрическое подключение к проводам изображено на рис. 5.9.

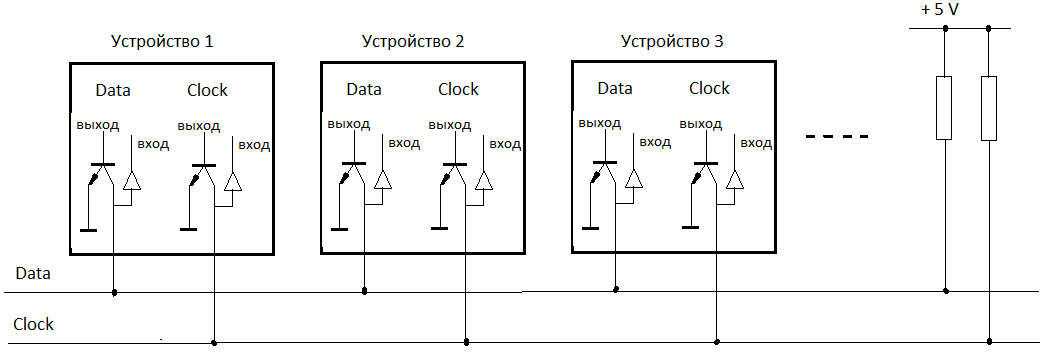


Рис. 5.9. Электрическое подключение устройств к проводам на шине *I*2*C.*

Устройства, подключаемые на шину *I*2*C*, имеют выход с открытым коллектором на линии данных и синхронизации. Изначально транзисторы заперты, и провода данных и синхронизации подтянуты к «1» через резисторы. Открытый транзистор подтягивает линию к земле, на линии устанавливается низкий уровень. Сопротивление подтягивающих резисторов будет зависеть от скорости работы устройств на линии. Для низких частот – несколько десятков кОм, для высоких – несколько кОм. Входы реагируют на состояние подключенных линий, но не вносят изменение в их состояние.

Ведущее устройство, master, первым выставившее на линию **данных** «0» при неактивной линии **синхронизации** может начинать обмен. **Синхронизацию всегда генерирует ведущее устройство.** При низком уровне *SCL* происходит выбор данных и их передача, при высоком – считывание приемником. Каждый бит информации передается за один такт. Уровень бита данных формируется при низком уровне на линии синхронизации, а считывается при высоком уровне синхроимпульса. Это иллюстрируется на рис. 5.10. Изменение уровня на линии данных при высоком уровне синхронизации недопустимо, потому что это будет воспринято как прекращение сеанса обмена. Перепад в «1» будет воспринят как прекращение всех обменов (*Stop*-условие мастера), перепад в «0» будет воспринят как начало нового обращения ведущего к одному из ведомых без закрытия предыдущего сеанса. Запрещенная комбинация отмечена на рис. 5.10 коричневой линией. Формирование *Start* и *Stop* условий показано на рис. 5.11.

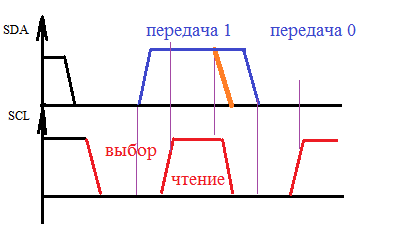


Рис. 5.10. Интервалы выбора и чтения информации.

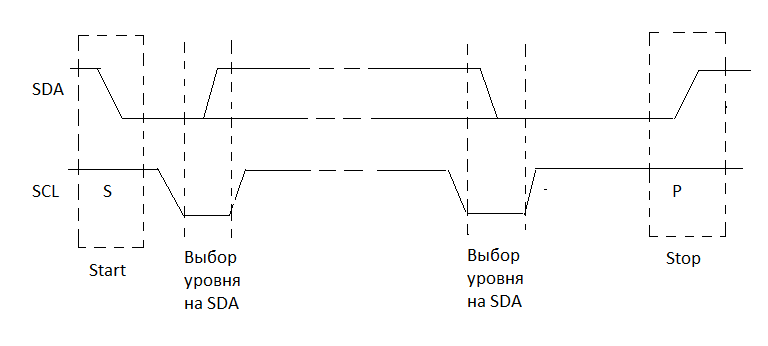


Рис. 5.11. Формирование *Start –*условия и *Stop-*условия.

Первая посылка, пакет-признак, всегда идет от ведущих, она содержит адреса ведомых и направление передачи. Ведущее устройство выставляет адрес ведомого и ждет подтверждения. Адрес ведомого устройства занимает 7 бит, 8-ой бит указывает дальнейшее направление передачи информации, рис. 5.12.

Далее следует подтверждение. Все устройства системы сравнивают 7 бит после старта со своими адресами, устройство, адрес которого был набран, становится ведомым и выдает сигнал подтверждения низким уровнем, *A (acknowledge).* После получения сигнала подтверждения ведущее устройство должно придержать линию синхронизации на низком уровне, пока ведомое ее не отпустит. Иначе синхронизация оборвется. Если выбранное устройство не готово к обмену, оно посылает высокий уровень в позиции бита подтверждения, *NA (not acknowledge)*. Обе эти ситуации показаны на рис. 5.13(а, б).

Далее следует пересылка пакетов информации. По окончании передачи ведомый должен передать в позиции подтверждения «1», после чего ведущий отпускает линию.

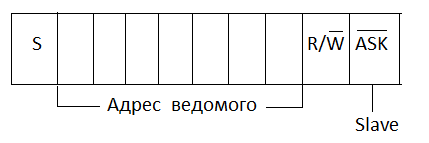


Рис. 5.12. Формат пакета-признака.

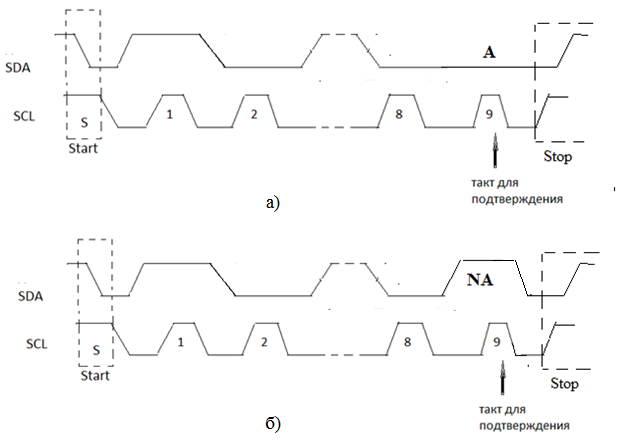


Рис. 5.13. Формат передаваемого пакета а) с подтверждением от ведомого, б) без подтверждения от ведомого.

Арбитраж.

Ведущим становится устройство, первым выставившее низкий уровень на провод данных перед появлением низкого уровня на проводе синхронизации.

Синхронизацию задает ведущее устройство, но ведомое, если оно не имеет достаточного быстродействия, может замедлять обмен данными.

Таким образом, низкий уровень на проводе синхронизации появляется от устройства с самой высокой частотой, а высокий уровень появляется, когда его сформирует устройство с самой низкой частотой обмена.

**5.2.3. *Интерфейс 8255A***

Схема этого параллельного интерфейса включает три 8-разрядных порта, разделяемых на две группы. При этом каналы порта *C* могут работать как в группе, так и автономно, если это необходимо. Структура …….

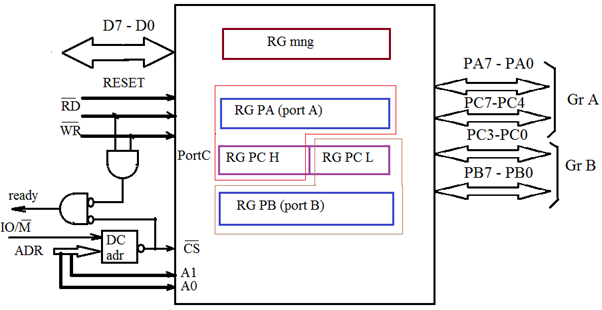


Рис. 5.14. Структура интерфейса 8255*A.*

Адресация регистров интерфейса производится по двум младшим адресным линиям. Режимы работы, в зависимости от адресации, состояний линий записи и чтения, а также входа выбора кристалла показаны в табл. 5.1.

Таблица 5.1.

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| *A*1 | *A*0 | *RD* | *WR* | *CS* | Описание обмена |
| 0 | 0 | 0 | 1 | 0 | Порт *A* выводит на ШД |
| 0 | 1 | 0 | 1 | 0 | Порт *B* выводит на ШД |
| 1 | 0 | 0 | 1 | 0 | Порт *C* выводит на ШД |
| 0 | 0 | 1 | 0 | 0 | С ШД ввод в порт *A* |
| 0 | 1 | 1 | 0 | 0 | С ШД ввод в порт *B* |
| 1 | 0 | 1 | 0 | 0 | С ШД ввод в порт *C* |
| 1 | 1 | 1 | 0 | 0 | C ШД ввод в регистр управления при *D*7=1, при *D*7=0 по ШД идут уровни установки или сброса |
| Х | Х | Х | Х | 1 | Линии ШД рассоединены со схемой |
| 1 | 1 | 0 | 1 | 0 | Запрещенная комбинация |
| Х | Х | 1 | 1 | 0 | Линии ШД рассоединены со схемой |

Схема может работать в двух режимах по каналам *A* и *B*, 0-м и 1-м. Дополнительно канал *A* может работать и в дуплексном режиме – 2-м.

Формат управляющего слова в регистре состояния показан на рис. 5.15.

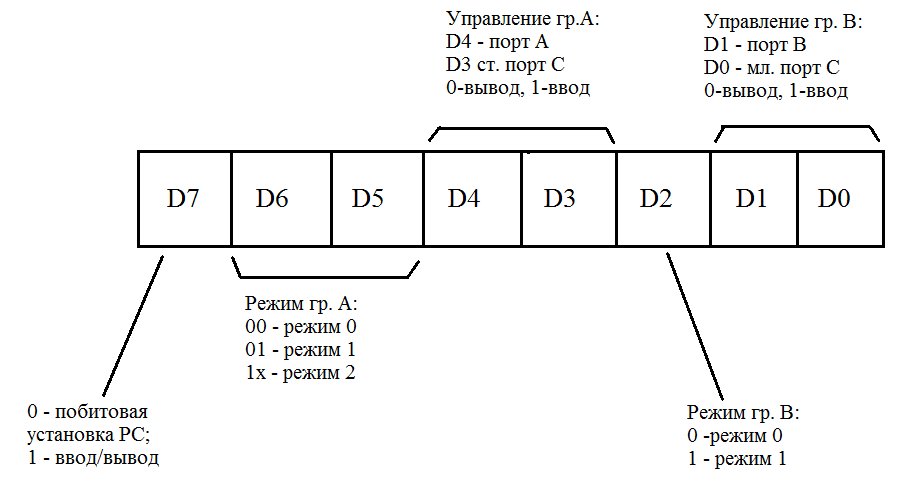


Рис. 5.15. Формат управляющего слова в регистре состояния 8255*A.*

Режим 0.

Нетактируемый ввод-вывод.

В этом режиме все каналы могут быть использованы для ввода или для вывода данных. Управляющие биты по каналу *C* не передаются. Порт *C* работает как две 4-битовые группы. Направление обмена для каналов задается управляющим словом.

Режим 1.

В данном режиме при вводе с порта A для ввода или вывода в канале C доступны только два бита: *PC*7 и *PC*6. Остальные биты поддерживают обмен по каналам *A* или *B*.

Для канала *A*:

*PC*5 при уровне 1 указывает на заполненность входного буфера. Прием следующего байта будет возможен лишь при *PC*5=0. (*IBFA*)

*PC*4 уровнем 0 передает сигнал синхронизации порта с шиной на ввод данных. (*STBA*)

*PC*3 в режиме 1 включен в поддержку канала *A*. Этот вывод используется как линия запроса на прерывание. Высокий уровень на этом выходе может быть использован для прерывания выполнения основной программы процессора, чтобы обслужить устройство ввода (*INTRA*). Сигнал *INTR* устанавливается в лог. 1 по переднему фронту сигнала *¬STB* (если установлен сигнал *INTE* разрешения прерывания) и сбрасывается по заднему фронту сигнала на входе.

Аналогично назначение *PC*2-*PC*0 для ввода по каналу *B*:

*PC*2 - *STBB*¬

*PC*1 - *IBFB*

*PC*0 - *INTRB*

При выводе по каналу *A* для ввода или вывода по каналу *C* могут использоваться только *PC*5 *PC*4.

*PC*7 указывает на заполненность выходного буфера (*OBFA*).

*PC*6 синхронизация приема данных с линии (*ACKA*).

*PC*3 – *INTRA*. Уровень 1 на этом выводе указывает на то, что информация с порта воспринята устройством. При 0 идет загрузка новых данных с процессора.

Для вывода по каналу *B*:

*PC*2 *- ACKB*

*PC*1 *- OBFB*

*PC*0 *- INTRB*

Режим 2.

В режиме 2 может работать только канал *A*. Канал становится двунаправленным, бит *PC3* так же используется для запросов на прерывание, свободных битов для ввода-вывода в старшей группе *C* нет.

*PC*4 *– STBA, PC*6 *- ACKA;*

*PC*5 *- IBFA, PC*7 *- OBFA*.

**ЛИТЕРАТУРА**

1.*Таненбаум Э., Остин Т. Архитектура компьютера.*[Электронный ресурс] / *Таненбаум Э., Остин Т. Архитектура компьютера.*– 6-е изд.– СПб.: Питер, 2018. – 816с.: ил.

2. Документация к ПО *Quartus* 15.0. *Introduction to the ARM Processor. Using Altera Toolchain. Altera Corporations – University Program, May 2015.*

3*. ARM Generic Interrupt Controller. Architecture Specification. Release Information*

*Copyright* © 2008, 2011, 2013 *ARM.*

**Неелова Ольга Леонидовна**

**«АРХИТЕКТУРА ВЫЧИСЛИТЕЛЬНЫХ СИСТЕМ»**

**Учебное пособие**

Редактор *И.И. Щенсняк*

План издания 2020 г.

Подписано к печати .2020

Объем 4 усл.-печ. л. Тираж экз. Заказ

Редакционно-издательский отдел СПбГУТ

193232 СПб., пр. Большевиков, 22

Отпечатано в СПбГУТ