Лекция 1.

Введение в АВС. Историческая справка.

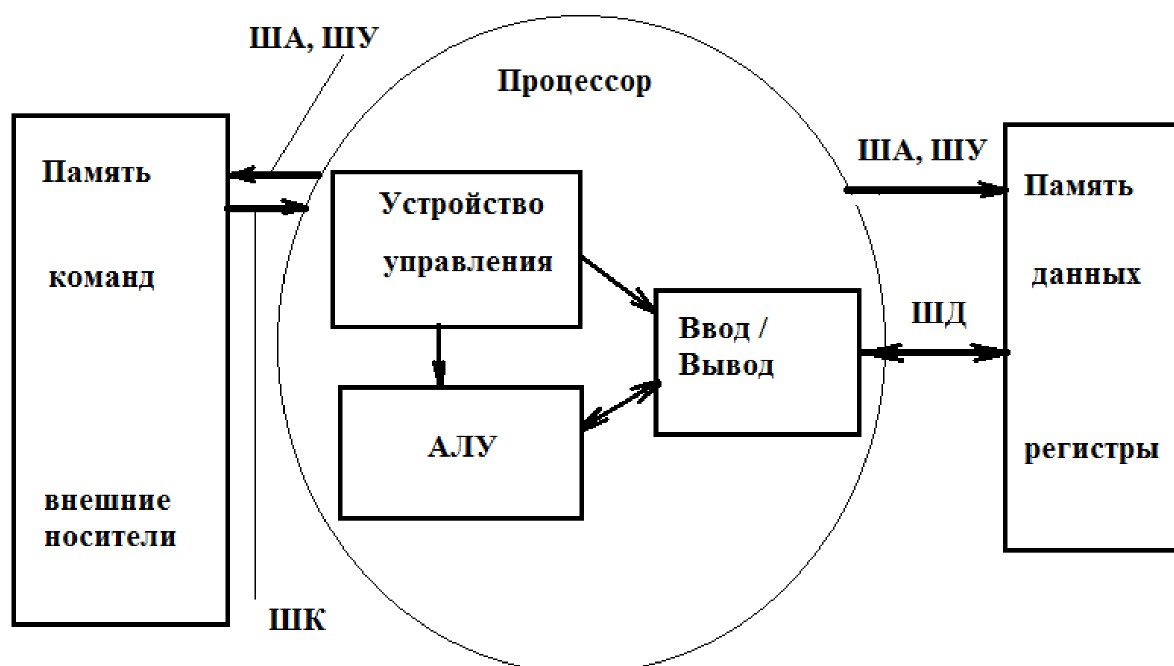
В настоящее время компьютеры стали неотъемлемой частью нашей жизни, но многим представляется, что они появились сравнительно недавно. На самом деле идеи создания вычислительных машин появились более 350 лет назад. Такие машины были созданы, они были, разумеется, механическими, и их можно отнести к нулевому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

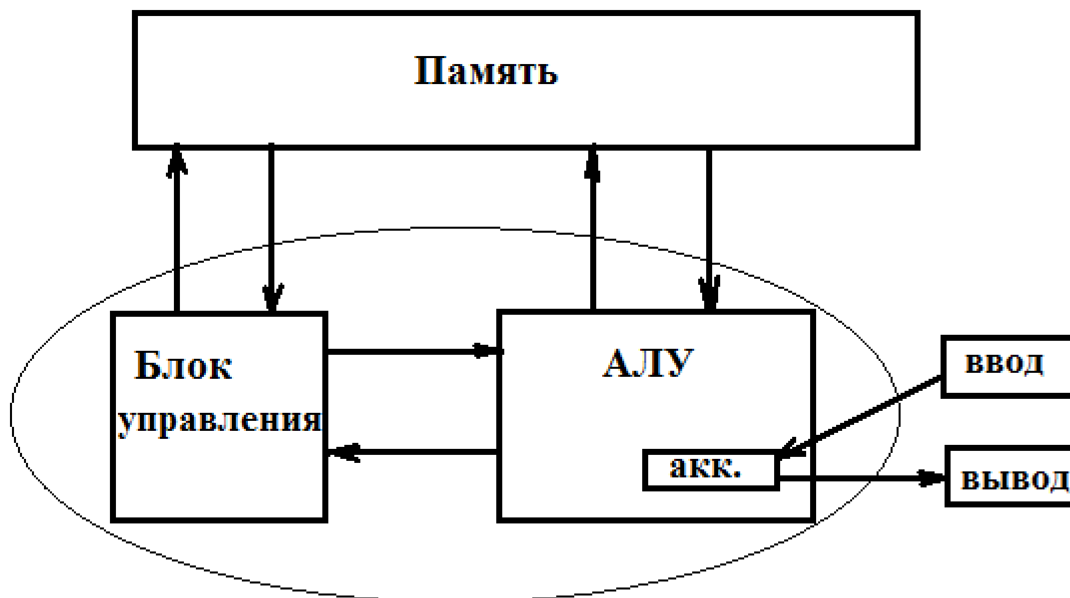
Но к идее создания вычислительной машины вернулись только спустя почти 100 лет, когда использовались электромагнитные реле. В основном, такие работы интенсивно проводились в годы Второй мировой войны для военных нужд. Из этих работ стоит упомянуть счетную машину Джона Атанасова (США). В отличие от остальных 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.



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.

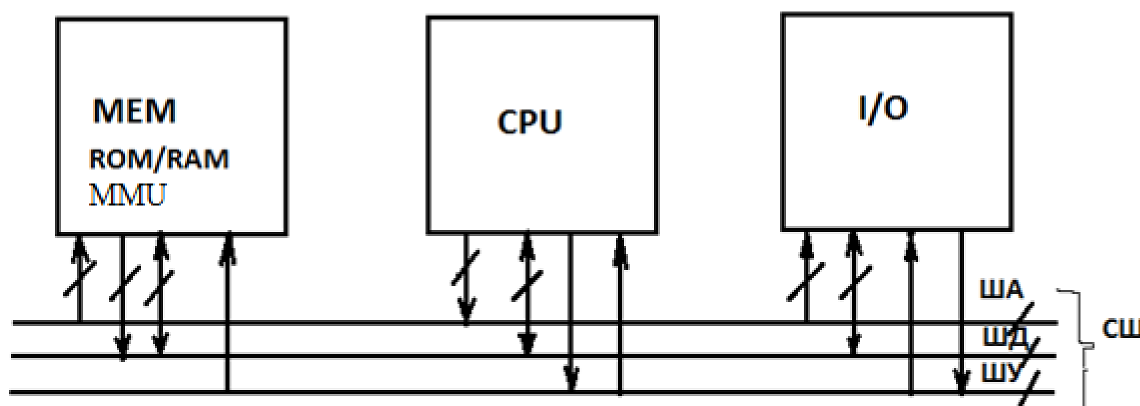
Первой машиной с использованием электронных ламп был ENIAC (Electronic Numeral Integrator and Computer). Однако в нем также использовалась десятичная арифметика и релейная память. Идея создания принадлежит Моушли и Экерту. Машина была создана в 1946 году, но тогда военным она не стала нужна. Один из 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.



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, которые всегда имеют три обязательных блока: процессорный блок, блок памяти и блок устройств ввода-вывода.



Такие 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**.

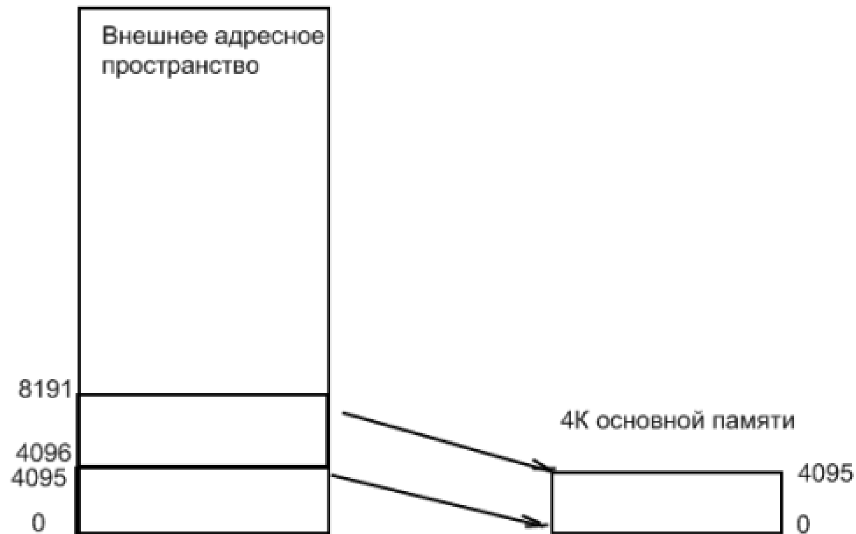
Физическая память, это тот реальный объем памяти, который есть в МПС. В компьютерах 50-х годов, объем памяти которых едва превышал 64Кбита с экзотическими параметрами, например, 4096 слов по 18 разрядов, программист имел возможность записать программу во внутреннюю память целиком. Это была, конечно, 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.

Рассмотрим кратко суть страничной организации 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.



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

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

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

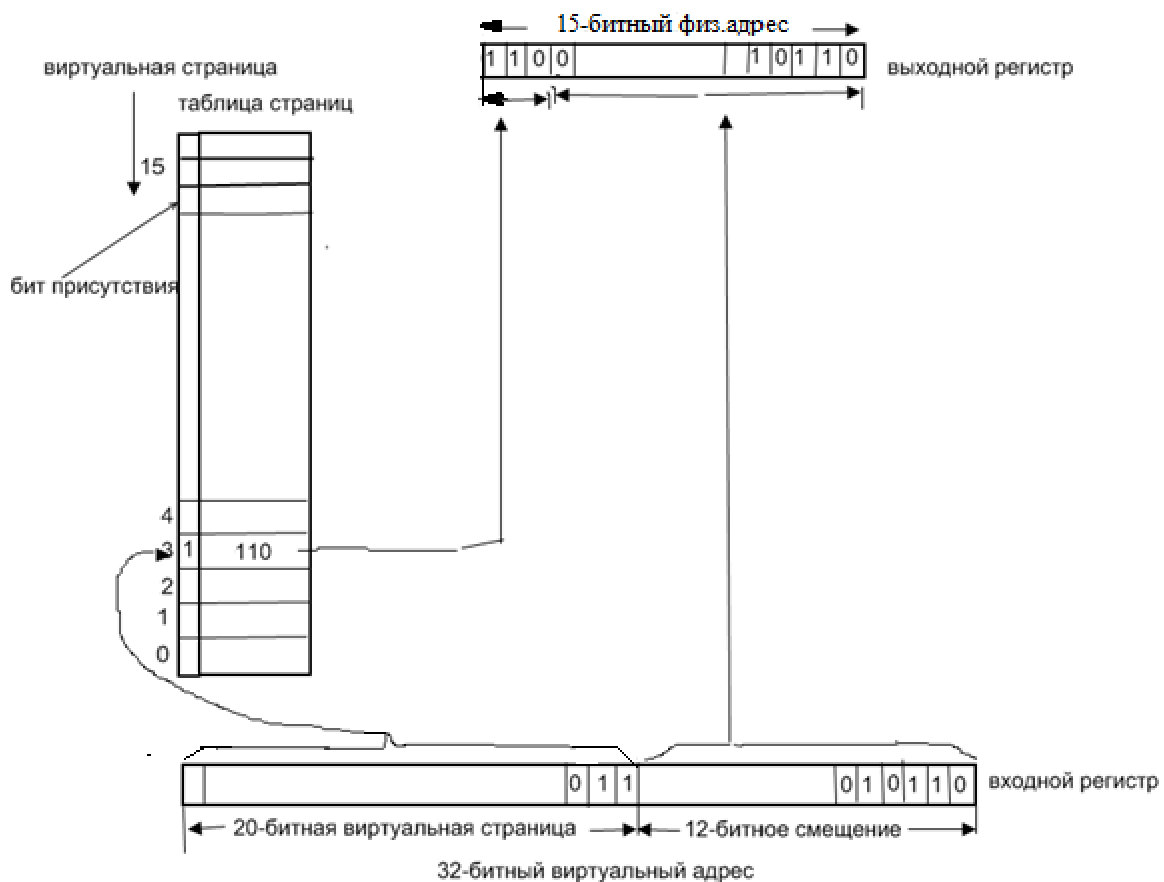
Страничный кадр	Физические адреса
7	28672-32767
6	
5	
4	
3	
2	8192-12287
1	4096-8191
0	0-4095

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).

Лекция 2.

Рассмотрим структуру преобразования виртуального адреса в физический.



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 20^{20} 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.

Пояснения к теме «Страничная организация памяти».

TLB (Translation Look aside Buffers) – буфер быстрого преобразования адреса. Представляет собой блок ассоциативной памяти. 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.

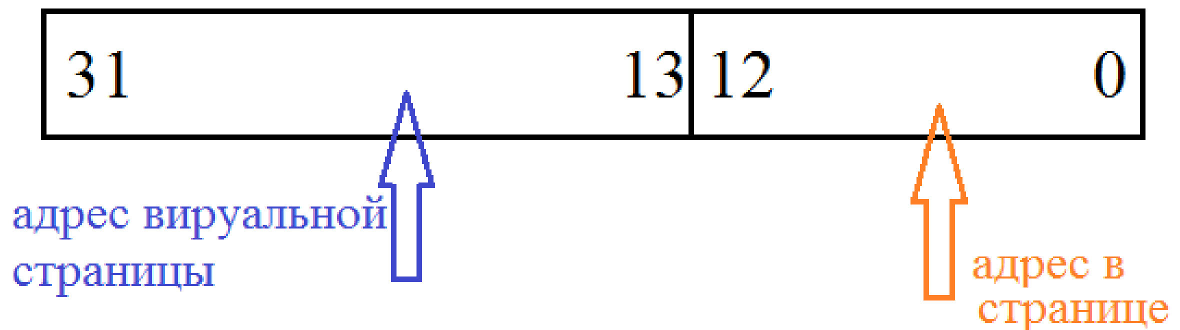
Пример.

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

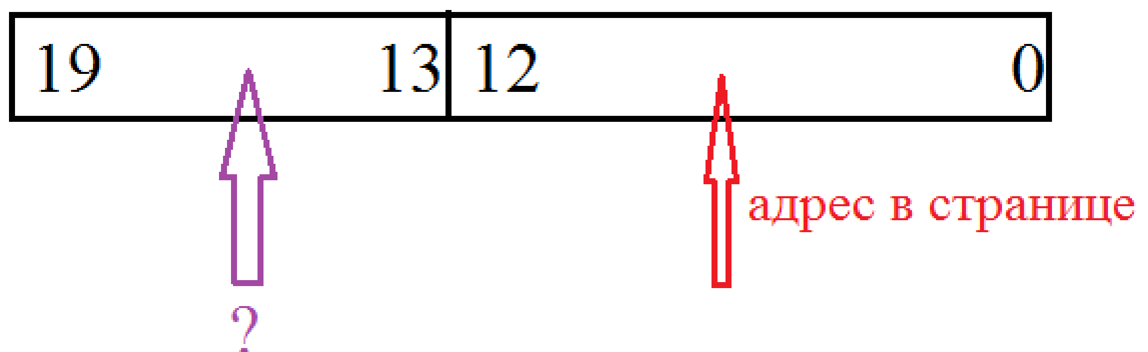
4 Гбайт – 2^{32}; 1 Мбайт = 2^{20}; 8 Кбайт – 2^{13}.

Вычислим количество страничных кадров: $2^{20}/ 2^{13} = 2^7$ (128)

Вид виртуального адреса



Вид физического адреса



Предположим, виртуальный адрес – 0x000195C3.

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.

Физический адрес 0x335C3.

Лекция 3.

Сегментация памяти.

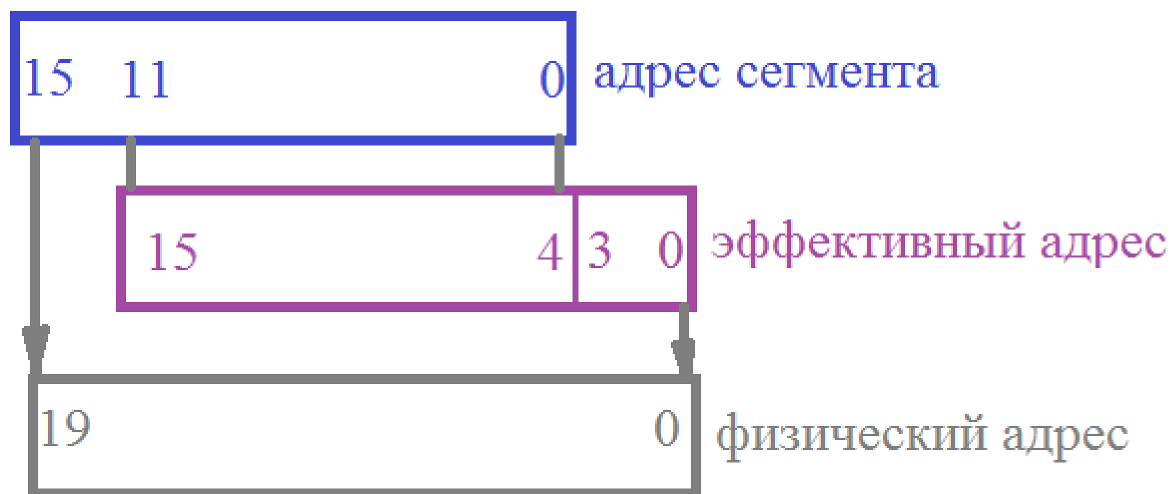
Второй способ получения виртуального адресного пространства – сегментация памяти. При этом образуется сразу несколько адресных пространств, каждое из которых может содержать свой тип информации.

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

$$2^{16} \text{ x } 2^{16} = 2^{32}$$

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.



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

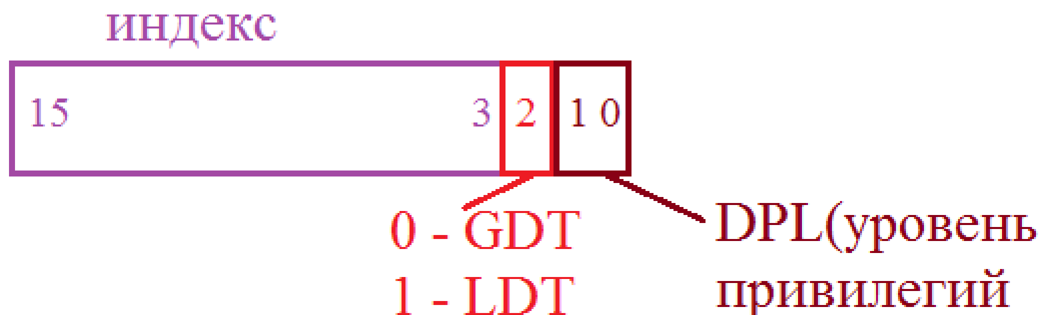
Или DS: 0x1731, BX: 0x3202

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: LTD (Local Descriptor Table) и GTD (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.

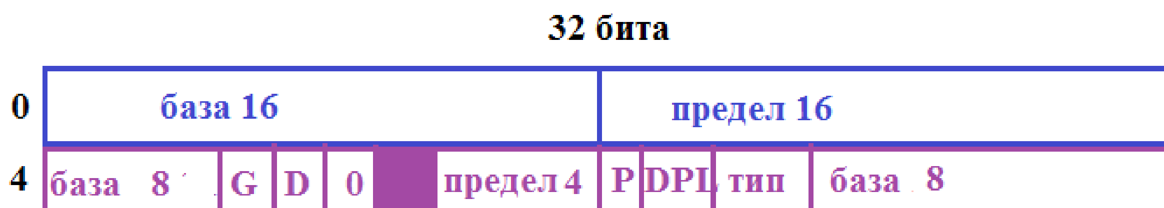
В сегментные регистры (CS, DS, ES, FS, GS и SS) загружается селектор сегмента – индекс (номер элемента в таблице дескрипторов), принадлежность к одной из таблиц (LDT/GDT) и уровень привилегий программы (0-3).



После загрузки селектора 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.

Формат дескриптора – 64 бита.



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 о.с.

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

Затем производится проверка соответствия смещения размеру сегмента. При 32-разрядном сегменте удобнее 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.

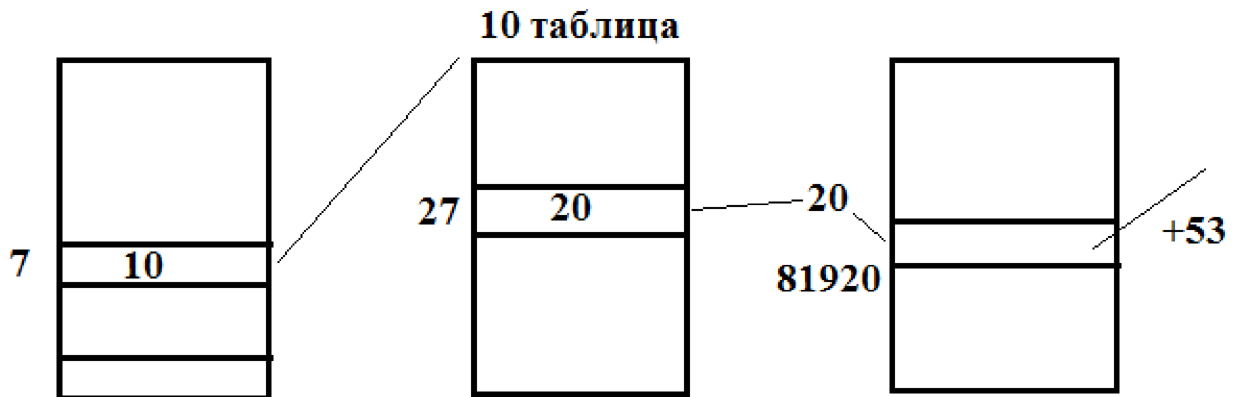
В случае разбиения сегментов на страницы линейный адрес будет еще не физическим, а виртуальным. Он будет содержать адрес каталога страниц,

адрес страницы в таблице и адрес символа на странице.

Предположим, что мы получили линейный адрес 0x01C1B035.

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 0x00014035.

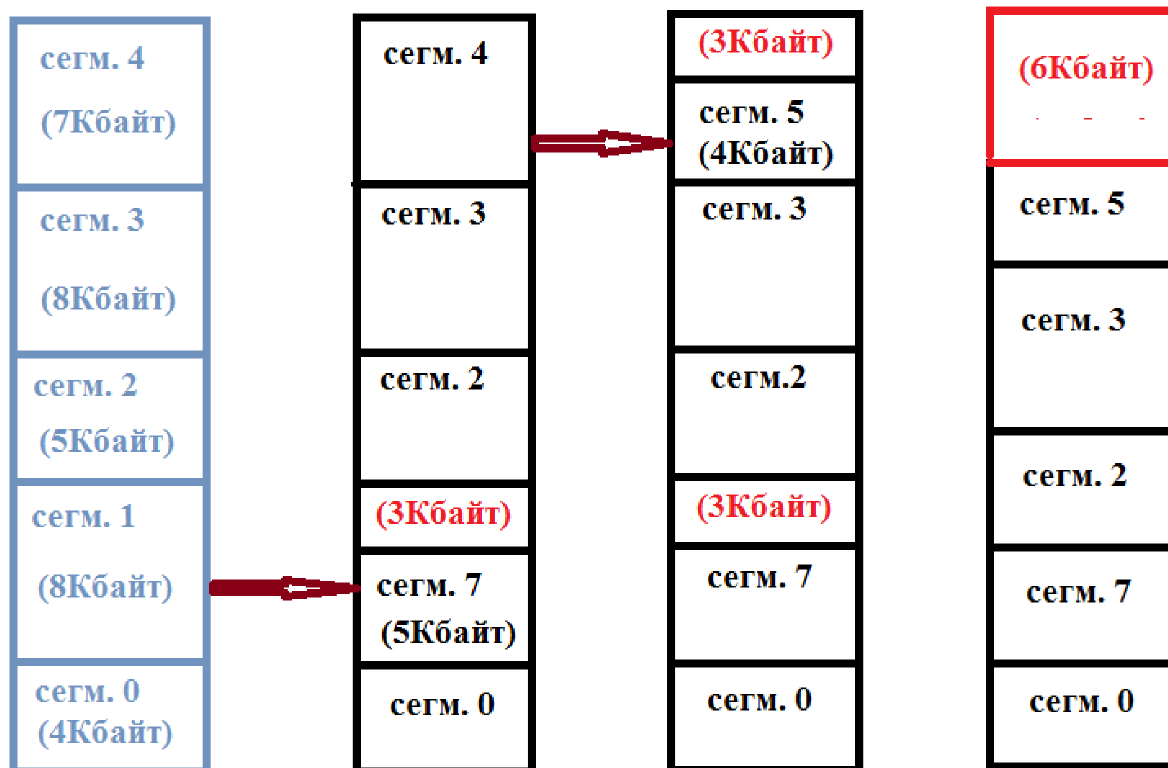


Лекция 4.

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**.



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.

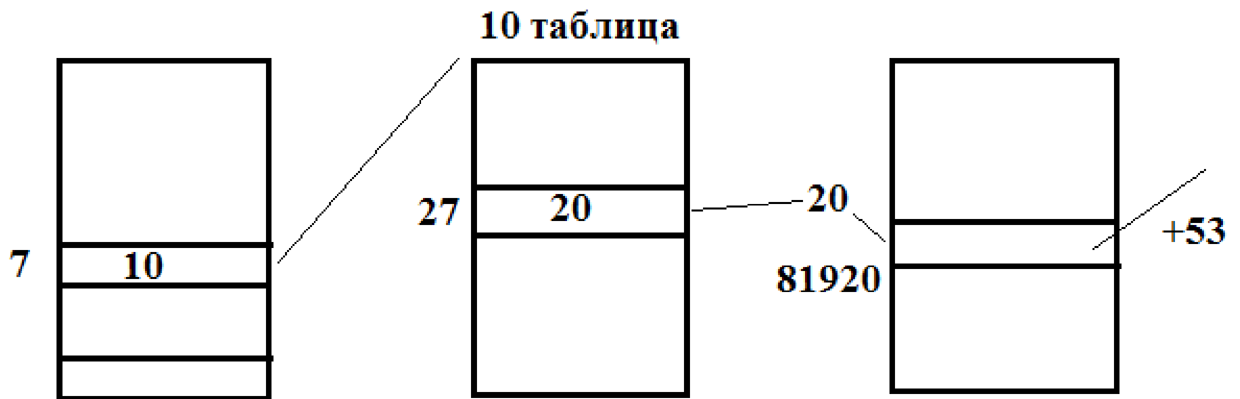
То есть если основываться на простой подкачке сегментов, не разделяя их на страницы, мы все время будем сталкиваться с проблемой внешней фрагментации. Поэтому сегменты разбивают на страницы фиксированного размера. При этом линейный адрес, полученный путем сложения базы дескриптора со смещением, уже не будет представлять собой физический адрес, а станет указателем для следующей системы таблиц. Вернемся к примеру, приведенному 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.

Предположим, что мы получили линейный адрес 0x01C1B035.

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 0x00014035.



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 ARM7, 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 ARM7 .

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

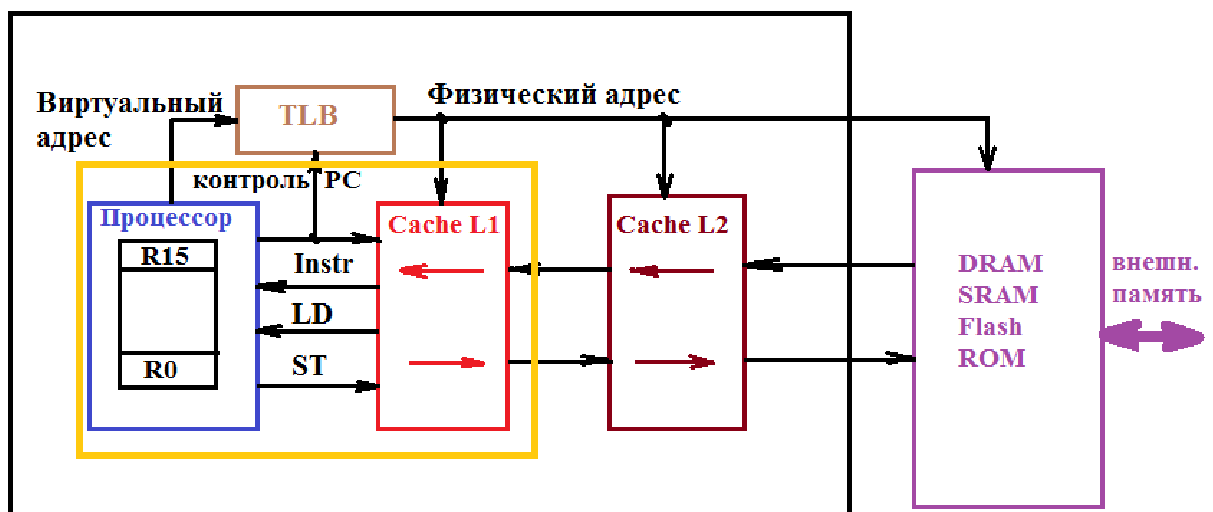
Соответственно, смещение будет определяться в 1 варианте – на 12 битах, во втором – на 16, в третьем – 20, а в четвертом – 24.

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.

Лекция 5

Структура микропроцессора.

При рассмотрении структуры процессора прежде всего вспоминают о его производительности, и способах ее повышения. Но в микропроцессорной системе есть еще блок памяти, и обычно основная память базируется на динамических ячейках. Поэтому в системе возникают проблемы задержек доставки операнда (время ожидания) и пропускной способности шины (объема данных, передаваемого в единицу времени). Одним из способов решения этих проблем 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



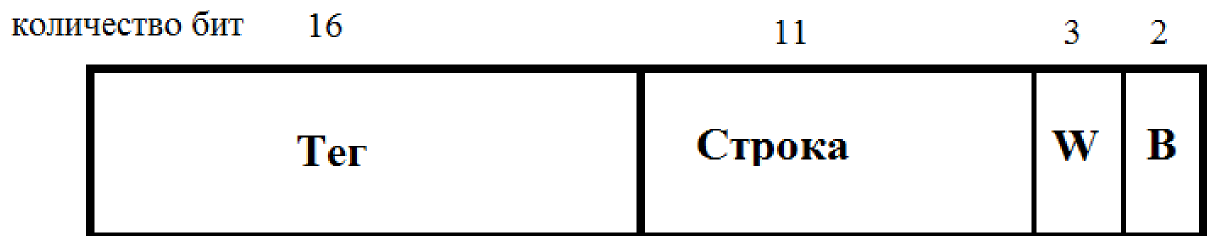
Теперь кратко обратимся к структуре кэш-памяти, общее представление о которой мы уже имеем.

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

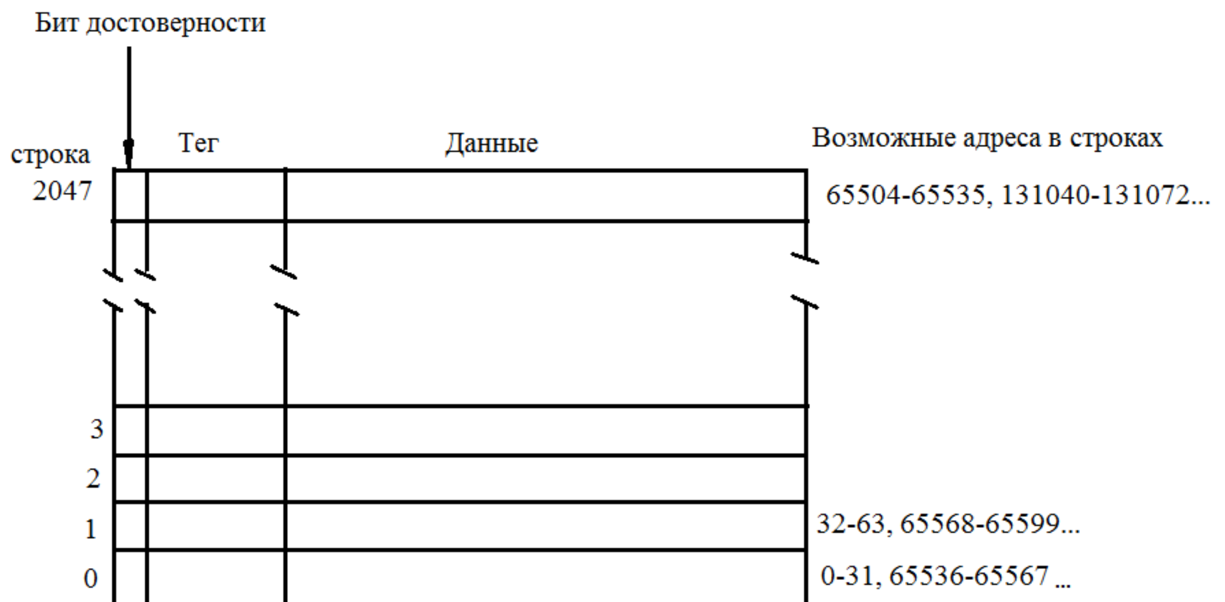
Так как общий доступный объем памяти по 32-разрядной шине составляет 4Гбайта, то мы имеем

$$2^{32} / 2^{16} = 2^{16}$$

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

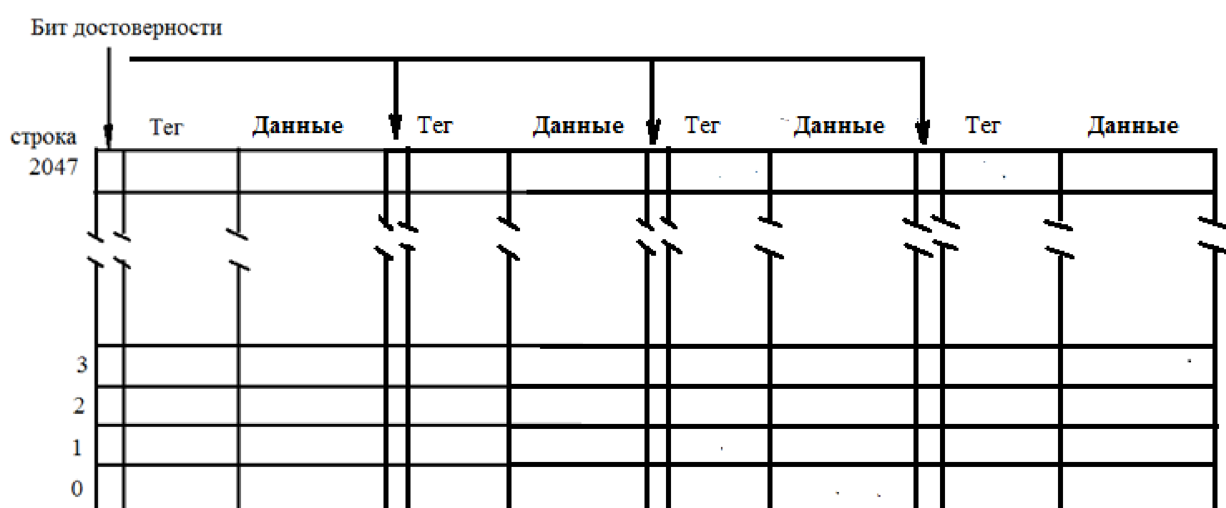


Каждая страница содержит 2048 строк по 8 слов в каждой. Размер слова – 4 байта. Структура такой кэш



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.



В процессоре CortexA9 кэш первого уровня – ассоциативная 4-входовая. Длина строкового элемента 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.

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.

В структуру процессора, кроме известного уже блока регистров, позволяющих осуществлять операции прямой и обратной загрузки, входят:

- блок вызова команд;

- блок очереди;

- блок прогнозирования переходов;

- блок переключений вычислительных функций;

- АЛУ и блок вычислений с плавающей точкой;

- интерфейсы для кэш L2, GIC и т.п.

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

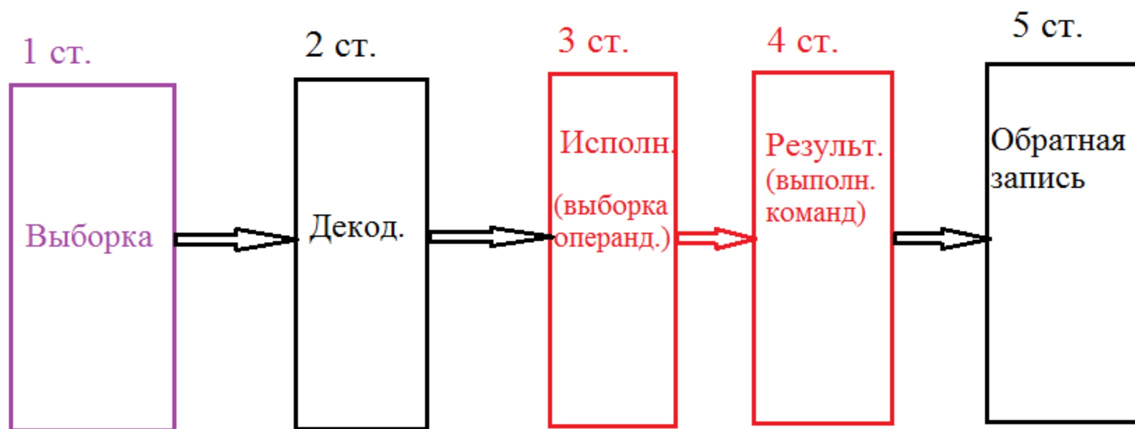
Блок переключений вычислительных функций позволяет определить, какой тип обработки данных должно поддерживать ядро процессора. Это может быть ядро, позволяющее векторные вычисления над целочисленными величинами, или числами с плавающей точкой – NEON MPE(Media Processing Engine). Оно поддерживает медийные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

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

Лекция 6.

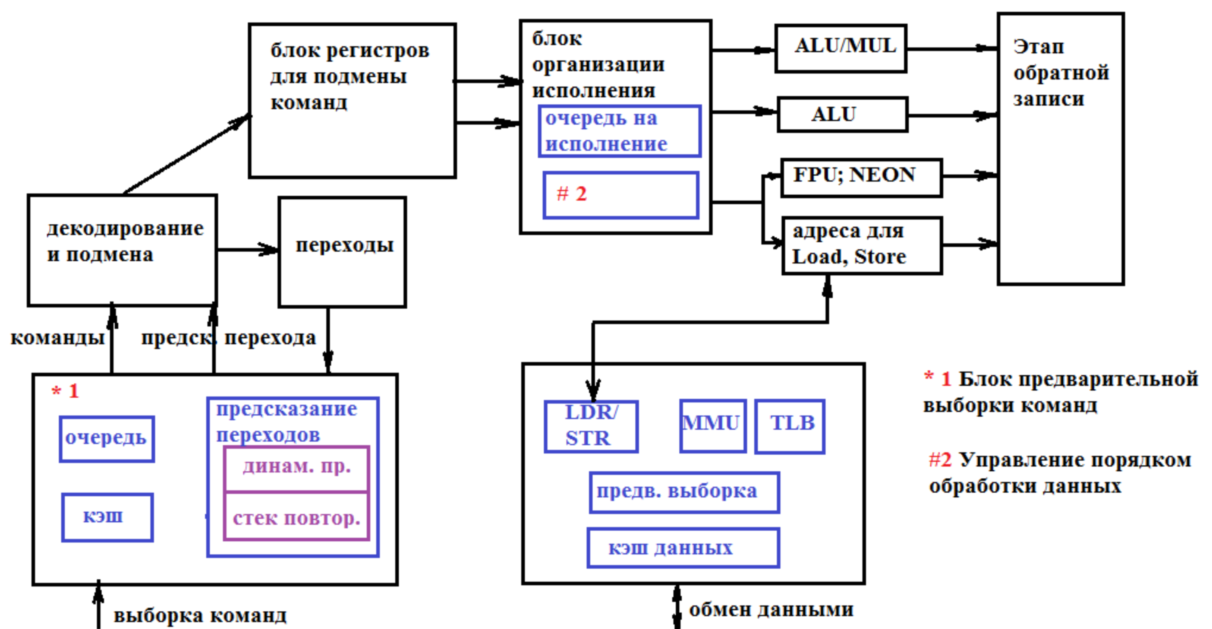
Структура процессора.

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.



В представленной 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)



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:

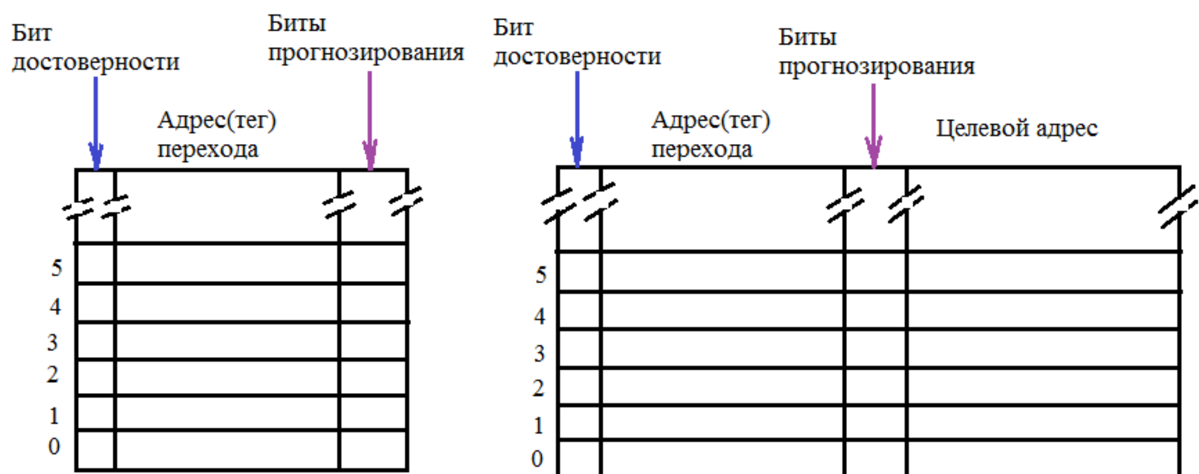
```
********
        CMP R3, #0
        BGE MET1
        MOV R5, R3
        B MET2
MET1: ADD R2, R1, R3
**********
```

Мы знаем, что на момент исполнения команды условного перехода по MET1, следующая команда уже находится на этапе дешифрации, поэтому обязательно должна исполниться. Но в программе, если содержимое R3 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. Один из таких способов связан с добавлением нового набора команд, где в структуру кода добавляется бит, по которому компилятор определяет, нужно совершать переход, или нет. Другой способ связан с накоплением информации о совершаемых в ходе программы переходах и

передаче ее компилятору. На основе этой информации компилятор управляет работой аппаратуры.

Динамическое прогнозирование переходов выполняется во время работы программы, приспосабливаясь к текущему режиму. Такая технология 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). Адрес перехода, исключая два младших разряда, заносится в таблицу с установкой бита достоверности. Биты прогнозирования показывают, совершался ли переход в прошлый раз (младший бит) и прогноз перехода (старший бит). Таблицы могут учитывать непосредственно адрес перехода – А, или адрес команды перехода и целевой адрес – Б.



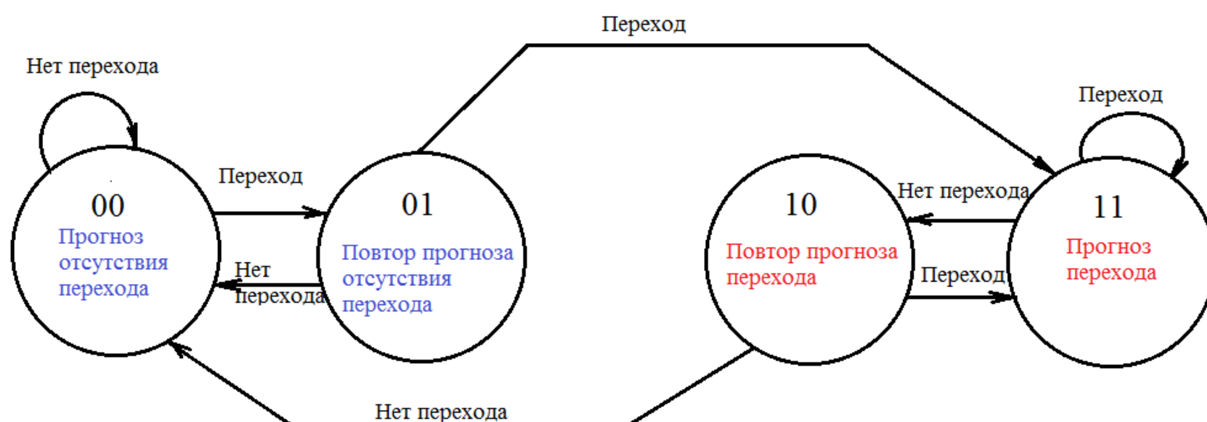
А – таблица с указанием адреса перехода;

Б – таблица с указанием адреса команды перехода и целевого адреса.

Иногда используют только один бит - 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. 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**.

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.



В тех случаях, когда программа выполняет небольшие невложенные циклы (работа в плотном цикле), этот цикл загружается в специальную кэш или в стек повторений. Это снижает общее энергопотребление.

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 R4, R1, R0

ADD R2, R4, R3

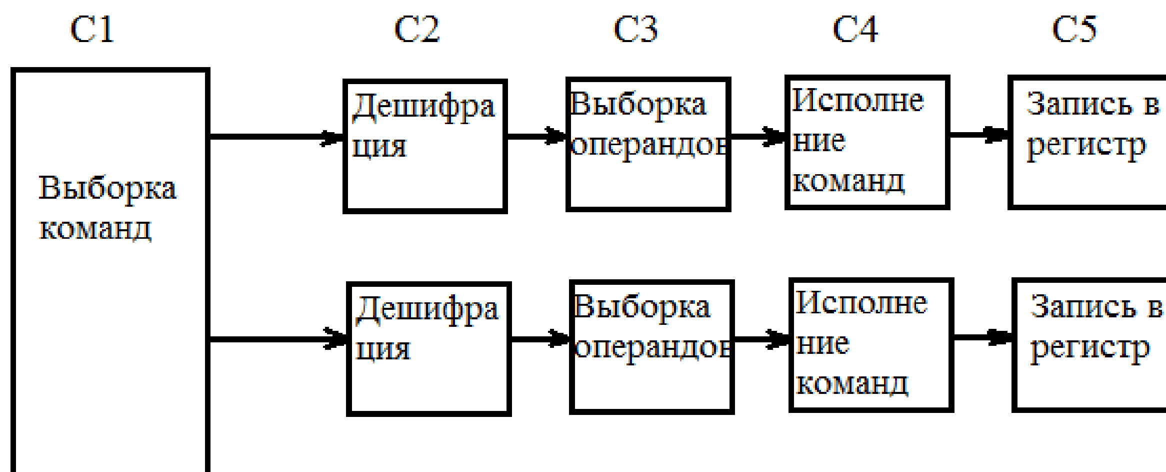
EOR R5, R5

Команда сложения не может быть запущена на исполнение, так как состояние R4 еще не записано (RAW). В этом случае может быть вместо нее исполнена команда логического сравнения.

Лекция 7.

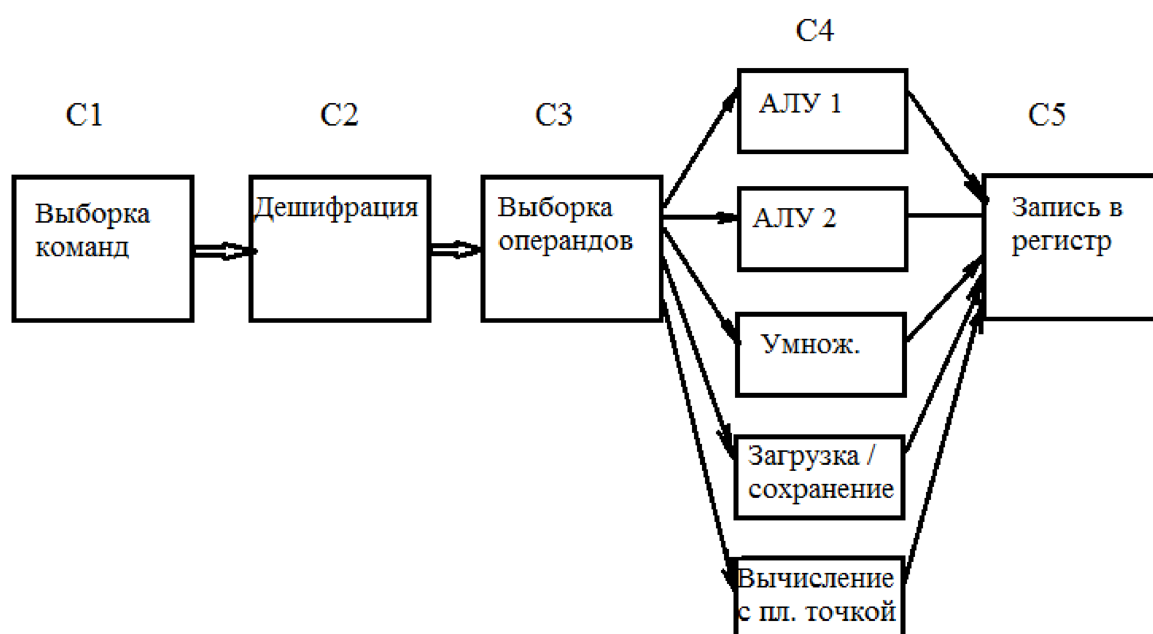
Структура микропроцессора.

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 начали применять сдвоенный конвейер. При этом блок выборки команд выставлял на конвейер сразу две команды.



При этом 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. Подобная схема применялась в Pentium, но там был один основной конвейер, а второй вспомогательный, для простых команд. 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. Поэтому распараллеливание начали делать на этапе С4 – исполнение команд. Такой подход к построению конвейера носит название **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 R3, R0, R1
2. ADD R4, R0, R2
3. ADD R5, R0, R1
4. ADD R6, R1, R4
5. MUL R7, R1, R2
6. SUB R1, R0, R2

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. Это необходимо для устранения возможных конфликтов.
Запишем порядок работы в виде таблиц
В порядке следования из блока предварительной выборки

№ цикла	команды	выдача	Результат в регистре	комментарии
1	MUL R3, R0, R1 ADD R4, R0, R2	1 2	- -	R0,R1-rd; R3-wr R0,R2-rd; R4-wr конфликтов нет
2	ADD R5, R0, R1 ADD R6, R1, R4	3 -	- -	R0,R1-rd; R5-wr R1-rd, R4 (RAW) Выдача приостановлена
3				Не готов р-т MUL
4			1 2 3	 Свободен для чтения R4
5		4		Свободен для чтения R4

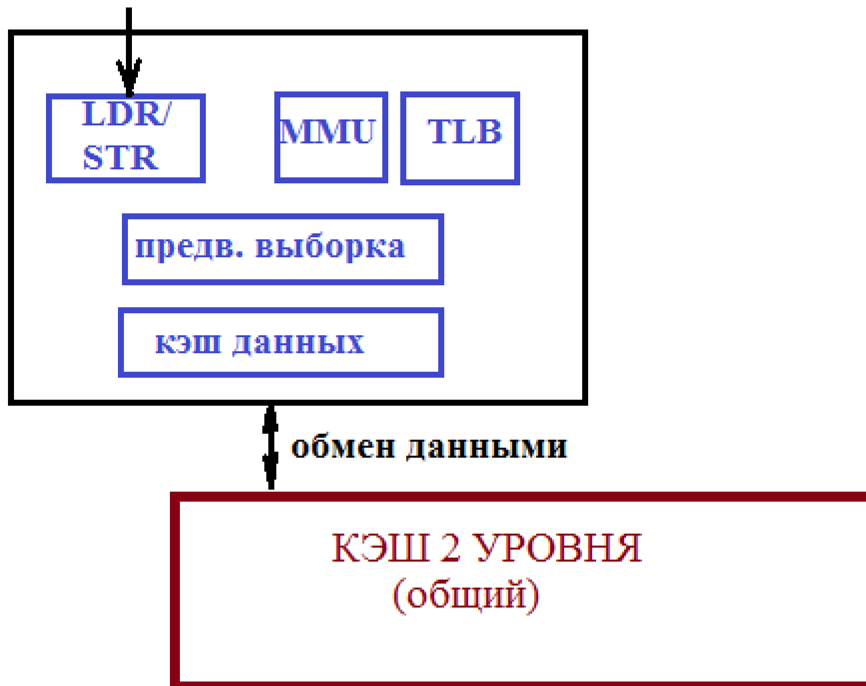
	MUL R7, R1, R2	5		R1,R2-rd; R7-wr
6	SUB R1, R0, R2			R1(WAR)
7			4	
8			5	Свободен для записи R1
9		6		Свободен для записи R1
10				
11			6	

С подменой команд и регистров.

№ цикла	команды	выдача	Результат в регистре	комментарии
1	MUL R3, R0, R1 ADD R4, R0, R2	1 2	- -	R0,R1-rd; R3-wr R0,R2-rd; R4-wr конфликтов нет
2	ADD R5, R0, R1 ADD R6, R1, R4	3 -	- -	R0,R1-rd; R5-wr R1-rd, R4 (RAW) Замена команды
3	MUL R7, R1, R2 SUB R1, R0, R2	5 6	 2	 R1(WAR) в 6 команде заменяется на S1. Свободен для чтения R4
4		4	 1 3	Свободен для чтения R4
5			6	
6			4 5	

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 суперскалярной архитектуры, не может полностью уберечь от простоев, если в обработку идет один программный поток. Простои вызываются конфликтами, а те, в свою

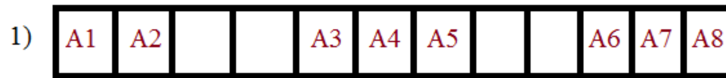
очередь, или сложностью команды, или отсутствием операнда в кэш 1-го уровня.

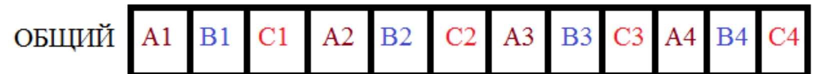


Кроме того, простои могут быть вызваны при неверном прогнозировании условного перехода и при исполнении безусловного.

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.

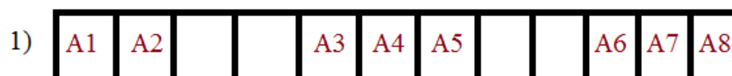
В **мелкомодульной многопоточности** выборка команд из разных потоков идет «по кругу», не дожидаясь простоя.



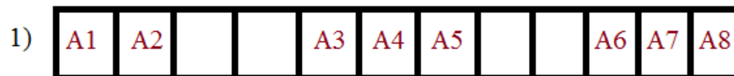


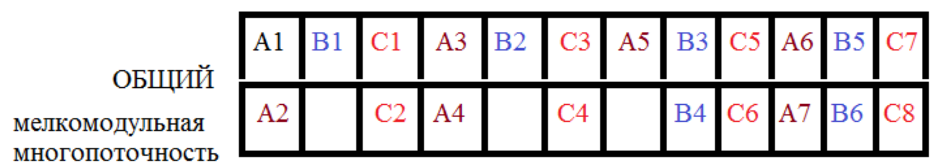
Каждой команде, поступающей на обработку, присваивается индекс потока, поэтому задачи не смешиваются, и путаницы не возникает. Если количество потоков соответствует количеству ступеней конвейера, то простои практически исключаются, но возникает иная трудность. Командам различных потоков нужны свои наборы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

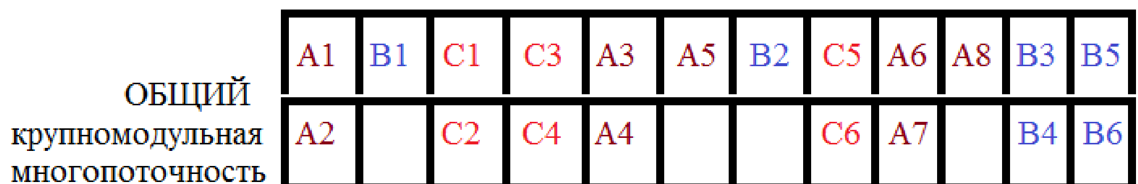
В случае **крупномодульной многопоточности** обращение к следующему потоку происходит только в случае возникновения простоя. Перед запуском каждого следующего потока конвейер очищается. Решение о запуске другого потока производится сразу после декодирования команды.

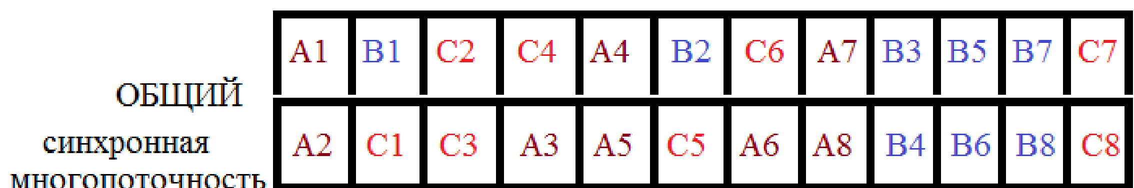


Все вышеизложенное относилось к случаю выдачи одной команды за цикл. Если же рассмотреть минимальную суперскалярную архитектуру конвейера, то оба метода многопоточности не спасают от возможности простоев. В суперскалярных процессорах используется метод **синхронной многопоточности**. Это усовершенствованная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









Лекция 8, 9.

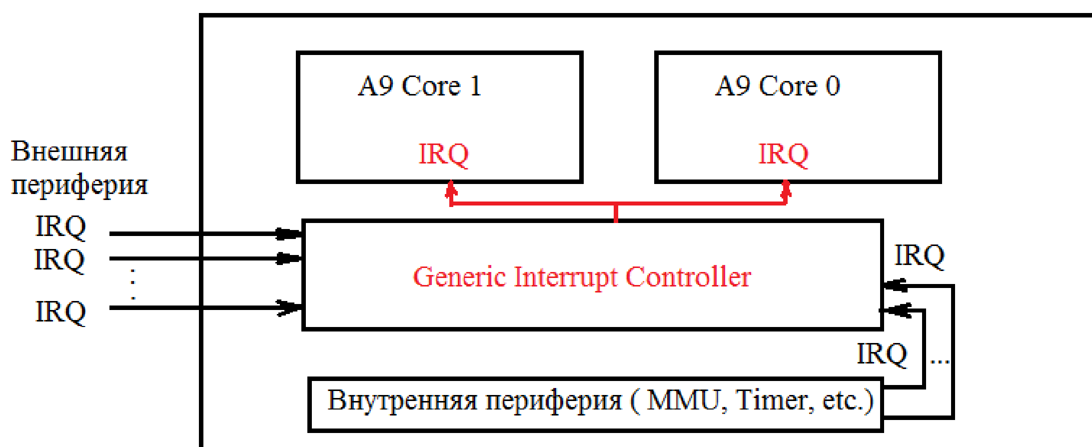
Обработка прерываний. Контроллер прерываний.

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.

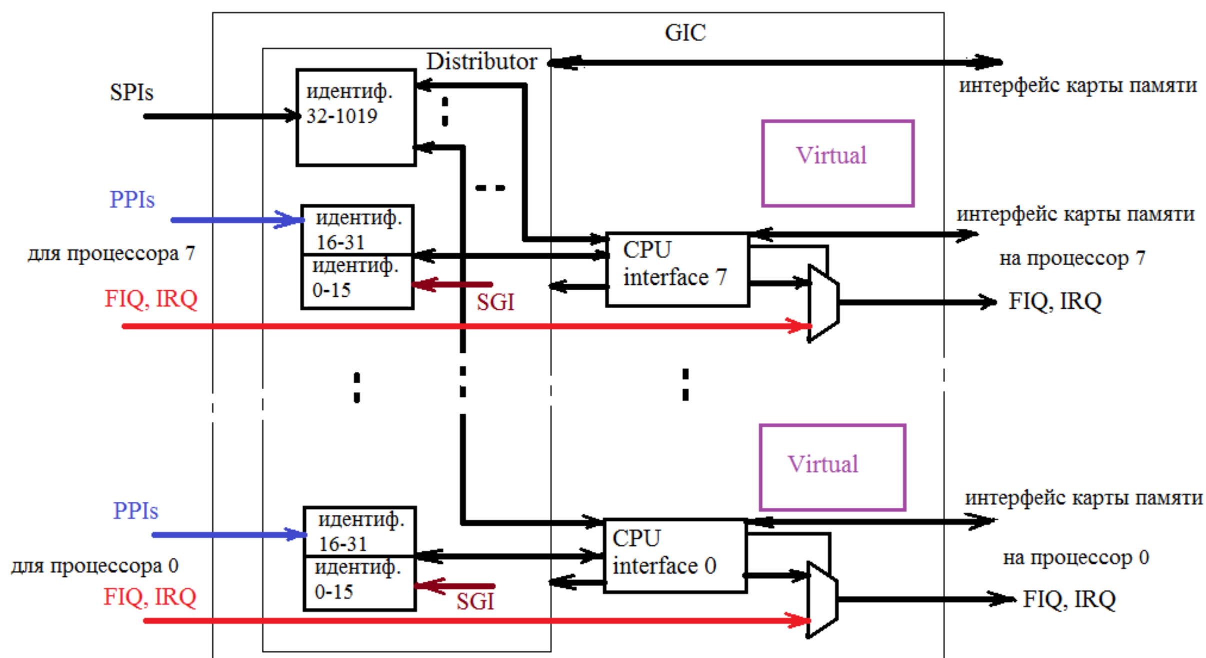
В зависимости от типа прерывания процессор должен установиться в определенный режим: Supervisor – программные прерывания или reset; Abort и Undefined – прерывания компилятора (ошибки); IRQ и FIQ – аппаратные прерывания. Прерывания отладчика (ловушки) обычно организуются в программе отладчика с помощью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

Аппаратные прерывания в процессорных системах линейки ARM (начиная с ARM7) обрабатываются с помощью контроллера прерываний GIC – Generic Interrupt Controller.



Контроллер входит в структуру объединенного ядра процессора (например, CortexA9), принимая запросы как от внешних, так и от внутренних источников, способных генерировать запросы на аппаратные прерывания. Для CortexA9 GIC способен обработать запросы от 255 источников. Все источники прерываний имеют свои идентификационные номера. Интерфейс CPU способен распознать 1020 источников. Группы по ID образуются в распределителе. Группа запросов под обслуживание любым процессором имеет ID от 32 по 1019. Группы индивидуальных запросов имеют ID от 0 до 31. При этом, программно-генерируемые и внешние источники объединяются в банки для каждого из процессоров. SGI имеют ID – от 0 до 15, PPI имеют ID от 16 до 31. Количество обслуживаемых процессоров в объединенном ядре возможно от 1 до 8-ми. Контроллер имеет два основных

блока: распределитель и интерфейсный блок. Интерфейсный блок определяет степень приоритета поступившего запроса, сравнивая его с приоритетом обрабатываемого прерывания. Производит анализ маски приоритета и установки вытеснения прерывания. Далее, при положительном результате анализа, запрос передается на процессор.



Типы сигналов запросов на прерывания, поступающие на контроллер.

Прерывания от внешних источников могут быть:

- индивидуальными для каждого процессора **(PPI – Private Peripheral Interrupt);**

- общие, т.е. поступающие на обработку любому свободному процессору **(SPI – Shared Peripheral Interrupt).**

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

Прерывания от внутренних источников, это программно генерируемые прерывания **(SGI – Software-generated Interrupt).** Запрос на прерывание этого типа всегда определяется с помощью краевого захвата.

Есть также запросы, обслуживаемые виртуальной машиной. Они не поступают на распределитель.

Различают следующие состояния сигнала прерывания, поступающего от контроллера на процессор:

Неактивный - прерывание не активно, или подвешено;

Подвешенный – запрос опознан и принят, но поставлен на ожидание;

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

Активный и подвешенный – процессор обслуживает запрос, а контроллер на это время подвешивает следующий запрос от этого же источника.

Регистры распределителя

Base Address	31 … 24	23 … 16	15 … 8	7 … 1 0	Register name
0xFFFED000	не используются			E	ICDDCR регистр управления
0xFFFED100	Set-enable bits				ICDISERn регистр установки разрешения
	Set-enable bits				
0xFFFED180	Clear-enable bits				ICDICERn регистр сброса разрешения
	Clear-enable bits				
0xFFFED400	Priority, offset 3	Priority, offset 2	Priority, offset 1	Priority, offset 0	ICDIPRn установка уровня приоритета для любого ID
	Priority, offset 3	Priority, offset 2	Priority, offset 1	Priority, offset 0	
0xFFFED800	CPUs. offset 3	CPUs. offset 2	CPUs. offset 1	CPUs. offset 0	ICDIPTRn установка номера входа на процессор
	CPUs. offset 3	CPUs. offset 2	CPUs. offset 1	CPUs. offset 0	
0xFFFEDC00	F15 F14 F13 F12	F11 F10 F9 F8	F7 F6 F5 F4	F3 F2 F1 F0	ICDICFRn установка типа приема запроса (уровень/фронт)
	F15 F14 F13 F12	F11 F10 F9 F8	F7 F6 F5 F4	F3 F2 F1 F0	

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

- программирование **возможности** пересылки **любого** прерывания на CPU-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.

Регистры CPU-интерфейса.

A;

Address	31 ... 10 9 8 7 ... 1	0	Registr name
0xFFFEC100	Unused	E	ICCICR регистр управления
0xFFFEC104	Unused (31–8)	Priority (7–0)	ICCPMR маска приоритета
0xFFFEC10C	Unused (31–10)	Interrupt ID (9–0)	ICCIAR регистр подтверждения
0xFFFEC110	Unused (31–10)	Interrupt ID (9–0)	ICCEOIR регистр завершения прерывания

CPU-интерфейс формирует:

- возможность посылки запроса;

- подтверждение прерывания;

- индикацию завершения обработки прерывания процессором;

- определение порядка вытеснения прерывания для процессора;

- распределение степени приоритета подвешенных прерываний для процессора.

Лекция 10.

Повторение пройденного материала. Ответы на вопросы. Подготовка к зачету.