Архитектура процессоров

К.Ю. Богачев

Настоящий документ последний раз обновлялся 12 мая 1999

Настоящий документ представляет собой материалы спецкурса К.Ю. Богачева “Архитектура процессоров”, читавшегося на механико-математическом факультете Московского государственного университета им. М.В. Ломоносова

Издано механико-математическим факультетом Московского государственного университета им. М.В. Ломоносова

1 Общая архитектура процессоров

1.1 Производительность

В настоящей главе мы обсудим факторы, определяющие производительность процессоров и пути ее повышения за счет оптимального выбора архитектуры.

1.1.1 Производительность как произведение трех факторов

Время, затрачиваемое процессором на задачу, может быть вычислено по формуле `C*T*I`, где

`C` число циклов на инструкцию,

`T` время на цикл,

`I` число инструкций на задачу.

1.1.2 Точка зрения CISC

Разработчики CISC (Complete Instruction Set Computer) стремились обеспечить большую поддержку для языков высокого уровня и операционных систем, поскольку улучшения в технике производства полупроводников сделали возможным производить все более сложные интегрированные цепи. Основной целью было уменьшение фактора `I`. Однако, представляется очевидным, что на этом пути архитектура должна становиться все более сложной, поскольку технологическая эволюция делает возможным включение в состав процессора все более сложных полупроводниковых устройств. С другой стороны, очень трудно уменьшить два других фактора: `C` поскольку инструкции сложные и требуют `программного декодирования` и `T` в силу `аппаратной сложности`.

1.1.3 Точка зрения RISC

Концепция RISC (Reduced Instruction Set Computer) возникла из статистического анализа того, как программное обеспечение использует ресурсы процессора. Исследования системных ядер и объектных модулей, порожденных оптимизирующими компиляторами, показали подавляющее доминирование простейших инструкций даже в коде для CISC машин. Сложные инструкции используются редко, поскольку микрокод обычно не содержит в точности те процедуры, которые нужны для поддержки различных языков высокого уровня и сред исполнения программ. Поэтому разработчики RISC процессоров убрали реализованные в микрокоде процедуры и передали программному обеспечению низкоуровневое управление машиной. Это позволило заменить процессорный микрокод в ROM на подпрограмму в более быстрой RAM, организованной как кэш инструкций.

1.1.4 Увеличение размера кода в RISC процессорах

Разработчики RISC процессоров улучшили производительность за счет уменьшения двух факторов: C и T. Однако, изменения, внесенные для уменьшения числа циклов на инструкцию и времени на цикл, имеют тенденцию к увеличению числа инструкций на задачу. Этот момент был в центре внимания критиков RISC архитектуры. Однако, использование оптимизирующих компиляторов и других технических приемов, сводит эту проблему на нет.

Тем не менее, последние исследования поставили под сомнение статистику, использованную при обосновании RISC архитектуры, по крайней мере в области встраиваемых систем и систем реального времени, где в некоторых случаях CISC процессоры могут превосходить RISC. Выбор между RISC и CISC может зависеть от конкретного приложения. Проблема состоит в том, что очень трудно оптимизировать приложение реального времени, поскольку в программном обеспечении, призванном реагировать на внешние события, данные и код "размазаны" по памяти. Поэтому для таких приложений очень важно иметь выбор в процессорной технологии, наилучшим образом адаптированной для данной задачи, причем выбор не только между RISC и CISC, но и даже между различными видами RISC процессоров.

1.1.5 Основные черты RISC архитектуры

Поскольку все инструкции имеют одинаковый формат, то декодирование производится аппаратно, т.е. микрокод не требуется. Каждая инструкция состоит из битовых полей, определяющих ее код и идентифицирующих операнды. В силу одинакового строения всех инструкций процессор может декодировать несколько полей одновременно для ускорения этого процесса.

Инструкции, производящие операции в памяти, обычно либо увеличивают время цикла, либо число циклов на инструкцию. Такие инструкции требуют дополнительного времени для своего исполнения, так как требуется вычислить адреса операндов, считать их из памяти, вычислить результат операции и записать его обратно в память. Для уменьшения негативного влияния таких инструкций, разработчики RISC процессоров выбрали архитектуру чтение/запись, в которой все операции выполняются над операндами в регистрах процессора, а основная память доступна только посредством инструкций чтения/записи. Для эффективности этого подхода RISC процессоры имеют большое количество регистров. Архитектура чтение/запись также позволяет уменьшить количество режимов адресации памяти, что позволяет упростить декодирование инструкций.

Для CISC архитектур время исполнения инструкции обычно измеряется в числе циклов на инструкцию. Разработчики RISC архитектур, однако, стремились получить скорость выполнения инструкции, равную одной инструкции за цикл.

Существует тесная взаимосвязь между оптимизирующими компиляторами и RISC архитектурами: компилятор может наилучшим образом оптимизировать код именно для RISC архитектур, а RISC процессоры во многих случаях именно компиляторам позволяют реализовать все свои возможности. Программирование на языке ассемблера, таким образом, исчезает для RISC приложений, так как компиляторы языков высоко-

го уровня могут производить очень сильную оптимизацию. Это обеспечивает лучшую переносимость исходного кода между различными архитектурами.

1.2 Пути повышения производительности

Каждое семейство процессоров использует механизмы повышения производительности за счет увеличения параллелизма операций в процессоре. Однако, такие улучшения делают управление такими процессорами более сложным в случае внешних асинхронных событий. Это следует учитывать при построении систем реального времени.

1.2.1 Конвейеризация (уменьшение C - числа циклов на инструкцию)

Процессор принимает новую инструкцию каждый цикл даже если предыдущие инструкции не завершены. В результате выполнение нескольких инструкций перекрывается и в процессоре находятся сразу несколько инструкций в разной степени готовности.

1.2.1.1 Исполнение нескольких инструкций на разных стадиях

Исполнение инструкций может быть разделено на несколько стадий: **выборка**, **декодирование**, **исполнение**, **запись результатов**. **Конвейер инструкций** может уменьшить число циклов на инструкцию посредством одновременного исполнения нескольких инструкций, находящихся на разных стадиях. При правильной аппаратной реализации конвейер, имеющий n стадий может одновременно исполнять n последовательных инструкций. Новая инструкция может приниматься к исполнению на каждом цикле, и эффективная скорость исполнения, таким образом, есть один цикл на инструкцию. Однако, это предполагает, что конвейер всегда заполнен полезными инструкциями и нет задержек в прохождении инструкций через конвейер.

1.2.1.2 Условия оптимального функционирования конвейера

Управление конвейером инструкций требует надлежащего эффективного управления такими событиями, как **переходы**, **исключения** или **прерывания**, которые могут полностью нарушить поток инструкций. Например, результат условного перехода известен, только когда эта инструкция будет исполнена. Если конвейер был заполнен инструкциями, следующими за инструкцией условного перехода и переход состоялся, то все эти инструкции должны быть выброшены из конвейера.

Более того, внутри конвейера могут оказаться **взаимозависимые** инструкции. Например, если инструкция в стадии декодирования должна читать из ячейки памяти, значение которой является результатом работы инструкции, находящейся в стадии исполнения, то конвейер будет остановлен на один цикл, поскольку этот результат будет доступен только после стадии записи результатов. Поэтому компилятору необходимо **переупорядочить** инструкции в программе так, чтобы по-возможности избежать зависимостей между инструкциями внутри конвейера.

Задержки внутри конвейера могут быть также вызваны временем доступа к оперативной памяти DRAM, которое на много превышает время цикла. Эта проблема в значительной степени снимается при использовании кэш памяти и буфера предвыборки инструкций (очереди инструкций).

Так как поток инструкций в CISC процессоре не регулярный и время исполнения одной инструкции (C*T) не постоянно, то конвейеризация в этом случае имеет серьезный недостаток, делающий ее малопригодной к использованию в CISC процессорах: именно, она приводит к очень сильному усложнению процессора.

RISC процессоры используют один и тот же формат для всех инструкций для того, чтобы ускорить декодирование и упростить управление конвейером, поэтому все инструкции исполняются за один цикл.

1.2.1.3 Суперконвейерная архитектура

Одним из способов повышения быстродействия является конвейеризация стадий конвейера. При таком подходе каждая стадия конвейера, такая как кэш или АЛУ, может принимать новую инструкцию каждый цикл, даже если эта стадия не завершила исполнение текущей инструкции. Однако, добавление новых уровней конвейеризации имеет смысл только в случае, если разработчик может значительно увеличить частоту процессора. Однако, увеличение производительности за счет увеличения внутренней частоты процессора имеет ряд недостатков. Во-первых, это увеличивает потребление энергии процессором, что делает суперконвейерные процессоры малопригодными для встраиваемых систем. Во-вторых, это вводит новые трудности в сопряжении процессора с памятью нижнего уровня, такой как DRAM. Быстродействие этой памяти растет не так быстро, как скорость процессоров, поэтому чем быстрее процессор, тем больше разрыв в производительности между ним и основной памятью.

1.2.1.4 Суперскалярная архитектура

Другим способом увеличения производительности процессоров является выполнение более чем одной операции одновременно. Суперскалярные процессоры имеют два или более конвейеров инструкций, работающих параллельно, что значительно увеличивает скорость обработки потока инструкций. Одним из достоинств суперскалярной архитектуры является возможность увеличения производительности без необходимости увеличения частоты процессора. Суперскалярному процессору требуется более широкий доступ к памяти, так, чтобы он мог брать сразу группу из нескольких инструкций для исполнения. Диспетчер анализирует эти группы и заполняет каждый из конвейеров так, чтобы снизить взаимозависимость данных и конфликты регистров. Компилятор должен оптимизировать код для обеспечения заполнения всех конвейеров.

1.2.1.5 Конвейерные процессоры и прерывания

Конвейерный процессор должен закончить обработку всех инструкций, находящихся в конвейере, перед тем, как перейти к обработке процедуры прерывания. Это увеличивает время реакции на внешние прерывания. Частые прерывания, требующие

переключения контекста задачи, ведут к значительному снижению производительности из-за постоянного сброса и заполнения конвейера.

1.2.1.6 Суперскалярные процессоры и ввод/вывод

Суперскалярный процессор берет из входного потока сразу несколько инструкций, но в порядке поступления. Инструкции берутся по порядку, но завершаться могут не по порядку. Но программа, осуществляющая ввод/вывод не может выполняться не по порядку. Поэтому существует последовательный режим исполнения инструкций для обеспечения их правильного порядка.

1.2.2 Независимые устройства

Требование одного и того же ресурса несколькими инструкциями блокирует их продвижение по конвейеру и приводит к вставке циклов ожидания требуемого ресурса. Суперскалярная архитектура с тремя исполняющими устройствами будет полностью эффективной, только если поток инструкций обеспечивает одновременное использование этих трех устройств. Выполнение инструкций может быть не по порядку поступления, так, чтобы команды перехода были проанализированы раньше, убирая задержки в случае осуществления перехода.

1.2.2.1 Целочисленное устройство (IU)

Это устройство выполняет целочисленные операции (арифметические, логические и операции сравнения) в своем АЛУ.

1.2.2.2 Устройство для работы с плавающей точкой (FPU)

Обычно устройство для работы с данными с плавающей точкой отделено от целочисленного устройства, которое работает только с целыми числами и числами с фиксированной точкой. Большинство FPU совместимы со стандартом ANSI/IEEE для двоичной арифметики с плавающей точкой.

1.2.2.3 Устройство управления памятью (MMU)

Потребность в MMU объясняется необходимостью получения виртуальной памяти. MMU вычисляет реальный физический адрес по виртуальному адресу.

В многозадачной системе каждая задача может адресовать всю память, поэтому управление использованием ресурсов берет на себя операционная система. Реализация виртуальной памяти, даваемая MMU, является аппаратным решением. Программные приложения работают только с виртуальными адресами, которые MMU транслирует в физические. Виртуальное адресное пространство всегда шире, чем физическое, для того, чтобы сделать возможным выполнение нескольких задач, требующих суммарный объем памяти больший, чем размер физической памяти.

При вычисление физического адреса память разбивается на блоки, имеющие одинаковы свойства. Существуют два основных метода для генерации физического адреса:

`разбиение на сегменты`
: - это трансляция между эффективным сегментным адресом и виртуальным сегментным адресом;

`разбиение на страницы`
: - это трансляция между виртуальным страничным адресом и реальным страничным адресом.

Эти страничные или сегментные адреса вычисляются по исходному эффективному адресу через дескрипторы. В каждом дескрипторе хранится два вида информации: указатель на следующую таблицу дескрипторов и специфические атрибуты памяти, которые защищают результирующий блок физической памяти:

`кэшируемость`
: (пространство ввода/вывода объявляется не кэшируемым);

`права доступа`
: (суперпользователь, пользователь);

`способ доступа`
: (только чтение, только запись, чтение и запись).

1.2.2.4 Устройство переходов (BU)

Основной целью этого устройства является предсказание условных переходов, для того, чтобы избежать простоя конвейера в ожидании результата вычисления условия перехода. Существует несколько приемов, ускоряющих обработку переходов.

`'Отложенные слоты' (delay slots)`
: Инструкцию, передающую управление от одной части программы другой, трудно исполнить за один цикл. Обычно загрузка процессорного указателя на следующую инструкцию требует один цикл, предвыборка новой инструкции требует еще один. Для избежания простоя, некоторые RISC процессоры (например, SPARC) позволяют вставить дополнительную инструкцию в так называемый '`отложенный слот`'. Эта инструкция, которая расположена непосредственно после команды перехода, но будет выполнена до того, как будет совершен переход.

  Однако, для суперскалярных RISC процессоров отложенные слоты работают не очень хорошо. Задержка при переходе может быть два цикла, а суперскалярный процессор, который выполняет за цикл `n` инструкций, должен найти `n` инструкций для помещения в конвейеры.

`'Спекулятивное' исполнение инструкций`
: Некоторые RISC процессоры (например, старшие модели семейств PowerPC и SPARC) используют так называемое 'спекулятивное' исполнение инструкций: процессор загружает в конвейер и начинает исполнять инструкции, находящиеся за точкой ветвления, еще не зная, произойдет переход или нет. При этом часто выбирается наиболее вероятная ветвь программы (на основе того или иного подхода, см. ниже). Если после исполнения команды перехода оказалось, что процессор начал исполнять не ту ветвь,

то все загруженные в конвейер инструкции из этой ветви и результаты их обработки сбрасываются, и загружается правильная ветвь.

Биты предсказания перехода в инструкции

Некоторые RISC процессоры (например, PowerPC) используют **биты предсказания перехода**, которые устанавливает компилятор в инструкции перехода, и предсказывающие, будет или нет совершен переход.

Эвристическое предсказание переходов

Некоторые RISC процессоры уменьшают задержки, вносимые переходами, за счет использования **встроенного предсказателя переходов**. Он предсказывает, что переходы вперед (проверки) произведены не будут, а переходы назад (циклы) - будут.

Для эффективной работы устройства предсказания перехода важно, чтобы код условия для условного перехода был вычислен как можно раньше (за несколько инструкций до самой команды перехода). Этого добиваются несколькими способами.

Независимость арифметических операций и кода условия

В CISC архитектурах все арифметические операции выставляют код условия по своему результату. Это сделано для уменьшения фактора I - числа инструкций на задачу, поскольку есть вероятность того, что следующая инструкция будет вычислять код условия по результату предыдущей инструкции и, следовательно, может быть удалена. Однако это приводит к тому, что между командой вычисления кода условия и командой перехода очень трудно вставить полезные инструкции, так как они изменят код условия. В RISC архитектурах арифметические операции **не изменяют код условия** (если противное явно не указано в инструкции, см. ниже). Поэтому возможно между инструкцией, вычисляющей код условия, и командой перехода вставить другие инструкции (переупорядочив их). Это позволит заранее узнать, произойдет или нет переход и загрузить конвейер инструкциями.

Поскольку возможна ситуация, когда следующая инструкция будет вычислять код условия по результату предыдущей инструкции, то в RISC архитектурах часть (SPARC) или все (PowerPC) арифметические операции также имеют вторую форму, в которой будет выставляться код условия по их результату. Таким образом, часть или все арифметические операции присутствуют в двух вариантах: один не изменяет код условия (подавляющее большинство случаев использования), а другой вычисляет код условия по результату операции.

Использование нескольких равноправных регистров с кодом условия

Некоторые RISC процессоры (например, PowerPC) используют несколько равноправных регистров, в которых образуется результат вычисления условия. Над этими регистрами определены логические операции, что иногда позволяет оптимизирующему компилятору заменить команды перехода при вычислении сложных логических выражений на команды логических операций с этими регистрами.

`Использование кода условия в каждой инструкции`

Некоторые RISC процессоры (например, ARM) используют код условия в каждой инструкции. В формате каждой инструкции предусмотрено поле, где компилятором записывается код условия, при котором она будет выполнена. Если в момент исполнения инструкции код условия не такой, как в инструкции, то она игнорируется. Это позволяет вообще обойтись без команд перехода при вычислении результатов условных операций.

1.2.3 Оптимизация внутренних ресурсов

Для суперскалярного RISC процессора обеспечение постоянной обработки сразу нескольких инструкций является нелегкой задачей. Проблемы, связанные с зависимостью данных, обостряются по сравнению с процессором, имеющим один конвейер, поскольку в каждый момент времени требуется принимать во внимание больше инструкций. Это требует более сложной управляющей логики. Техника, подобная переименованию регистров и таблицам регистров, позволяет обнаруживать и минимизировать `зависимости данных` и `конфликты регистров`.

1.2.3.1 Таблицы регистров (register scoreboarding)

Таблица регистров позволяет проследить за использованием регистров. Она имеет бит для каждого регистра процессора. Если этот бит установлен, то регистр находится в состоянии ожидания записи результата. После записи результата этот бит сбрасывается, разрешая использование этого регистра.

1.2.3.2 Переименование регистров (register renaming)

Переименование регистров является аппаратной техникой уменьшения конфликтов из-за регистровых ресурсов. Компиляторы преобразуют языки высокого уровня в ассемблерный код, назначая регистрам те или иные значения. В суперскалярном процессоре операция может потребовать регистр до того, как предыдущая инструкция закончила использование этого регистра. Это состояние `не является конфликтом данных`, поскольку этой операции не требуется значение регистра, а только сам регистр. Однако, эта ситуация приводит к остановке конвейера до освобождения регистра. Идея разрешения этой проблемы состоит в следующем: берем свободный регистр, переименовываем его для соответствия параметрам инструкции, и даем инструкции его использовать в качестве требуемого ей регистра.

1.2.4 Кэш память (уменьшение `T` - времени на цикл)

Время, необходимое для выборки инструкций, в-основном, зависит от подсистемы памяти и часто является ограничивающим фактором для RISC процессоров в силу высокой скорости исполнения инструкций. Например, если процессор может брать инструкции только из DRAM с временем доступа 70 ns, то скорость их обработки (при расчете одна инструкция за цикл) будет соответствовать тактовой частоте 14 MHz. Эта проблема в значительной степени снимается за счет использования `кэш памяти`.

1.2.4.1 Внутренняя или внешняя быстрая память SRAM

Кэш память - это быстрая SRAM, вставленная между исполнительными устройствами и системной RAM. Она сохраняет последние использованные инструкции и данные, так, что циклы и операции с массивами будут выполняться быстрее. Когда исполняющему устройству нужны данные и они не находятся в кэш памяти, то это **кэш-промах**: процессор должен обратиться к внешней памяти для выборки данных. Если требуемые данные находятся в кэше, то это **кэш-попадание**: доступ к внешней памяти не требуется.

Таким образом, кэши разгружают внешние шины, уменьшая потребность в них процессора. Это позволяет нескольким процессорам разделять внешние шины без уменьшения производительности каждого из них.

Кэш содержит строки из нескольких последовательных байтов (обычно 32 байта), которые загружаются процессором, используя так называемый **импульсный** (или блочный) **доступ** (burst access). Даже если CPU нужен один байт, все равно будет загружена целая строка, так как вероятно, что тем самым будут загружены следующие выполняемые инструкции или используемые данные. Блочные передачи обеспечивает высокие скорости передачи для инструкций или данных в последовательных адресах памяти. При таких передачах только адрес первой инструкции или данного будет послан в подсистему внешней памяти. Все последующие запросы инструкций или данных в последовательных адресах памяти не требуют дополнительной передачи адреса. Например, загрузка 16 байтов требует 5 циклов, если MC68040 делает блочную передачу для загрузки строки кэша, и 8 циклов, если память не поддерживает блочный режим передачи.

1.2.4.2 Единый кэш или раздельные кэши для инструкций и данных

Кэш, в котором вместе хранятся данные и инструкции, называется **единым кэшем**. Одним из способов повышения производительности является введение в процессоре **трех** шин: **адреса**, **инструкций** и **данных**. Эта архитектура, называемая **Гарвардской**, впервые была использована в процессоре DSP, который создавался для быстрых вычислений. В этой архитектуре возможно разделить кэши для инструкций и данных для удвоения эффективности кэш памяти.

В типичной Гарвардской архитектуре присутствуют три вида кэш памяти: специальные кэши (например, **TLB**), внутренние кэши инструкций и данных (**первого уровня**) и внешний единый кэш (**второго уровня**).

1.2.4.3 Кэш с прямой и обратной записью

Кэши данных в зависимости от их поведения при записи данных в кэш разделяют на два вида.

Кэш с прямой записью (write-through cache)

Этот вид кэш памяти при записи в нее сразу инициирует цикл записи во внешнюю память. Основным достоинством такого кэша является просто-

та и то, что данные в кэше и в памяти всегда идентичны, что упрощает построение многопроцессорных систем.

Кэш с обратной записью (write-back cache)

Этот вид кэш памяти при записи в нее не записывает данные во внешнюю память. Запись в память осуществляется при выходе строки из кэша или по запросу системы синхронизации в многопроцессорных системах. Такая организация кэш памяти может значительно ускорить выполнение циклов, в которых обновляется одна и та же ячейка памяти (будет записано только последнее, а не все промежуточные значения как в кэше с прямой записью). Другим достоинством является уменьшение потребности процессора во внешней шине, что позволяет разделять ее нескольким процессорам. Недостатком такой организации является усложнение схемы синхронизации кэшей в многопроцессорных системах.

В силу его значительно большей эффективности, большинство современных процессоров используют кэш с обратной записью.

1.2.4.4 Организация кэша

Кэш основан на **сравнении адреса**. Для каждой строки кэша хранится адрес ее первого элемента, называемый адресом строки. Для уменьшения объема дополнительно хранимой информации (адресов строк) и ускорения поиска адреса используют несколько технических приемов.

- Пусть длина строки есть 2 в степени b байт. Адреса строк выровнены на границу своего размера, т.е. последние b бит адреса - нулевые и потому не хранятся (т.е. размер адреса уменьшен до 32-b).
- Фиксируется некоторое i. Строки хранятся как один или несколько (N) массивов, отсортированными по порядку i младших битов адреса (т.е. младших среди оставшихся 32-b). Таким образом, k-й элемент массива имеет адрес, биты которого в позициях от 32-i-b+1 до 32-b образуют число, равное k. Это позволяет не хранить эти биты (т.е. размер адреса уменьшен до 32-b-i).
- Комбинация чисел b и i подбирается так, чтобы биты b+i логического адреса совпадали бы с соответствующими битами физического адреса при страничном преобразовании (т.е. были бы смещением в странице). Это позволяет параллельно производить трансляцию адреса и поиск в кэше (т.е. параллельно работать MMU и кэшу). При типичном размере страницы 4Kb это означает b+i=12.
- Определение того, содержится ли данный адрес в кэше, производится следующим образом. Берутся биты в позициях от 32-i-b+1 до 32-b, образующие число k. Затем берутся элементы с номером k в каждом из N массивов и у полученных N строк сравниваются адреса с 32-i-b битами адреса (которые уже транслированы MMU в физический адрес). Если обнаружено совпадение (т.е. имеет место кэш-попадание), то берется байт с номером b в строке.

Если рассматривается внешний кэш, то согласовывать его работу с MMU не требуется, поскольку внешний кэш работает уже с физическим адресом.

Пример: для PowerPC 603 выбрано b=5 (т.е. длина строки 32 байта), i=7 (т.е. длина массива 127), N=2 (т.е. используются два массива).

Для каждой строки кэша с обратной записью помимо адреса хранится также признак того, что эта строка содержит корректные данные, т.е. данные в кэш памяти и в основной памяти совпадают. Этот признак используется для записи строки в память при ее выходе из кэш памяти, а также в многопроцессорных системах.

1.2.4.5 Алгоритмы замены данных

Если все строки кэш памяти содержат корректные данные, то для обеспечения кэширования новых областей памяти необходимо выбрать строку, которая будет перезаписана. Эта строка выходит из кэша и, если требуется, ее содержимое будет записано обратно в память.

Существуют три алгоритма замены данных в кэше:

- `вероятностный` алгоритм: в качестве номера перезаписываемой строки используется случайное число;
- `FIFO` алгоритм: первая записанная строка будет первой перезаписана;
- `LRU` (Last Recently Used) алгоритм: наименее используемая строка будет заменена новой.

1.2.4.6 Специальные кэши

Для повышения производительности процессора вводятся ряд специальных кэшей.

`TLB` (Translation Look-aside Buffers) - это кэш память, используемая MMU для хранения результатов последних трансляций логического адреса в физический. Содержит пары: логический адрес и соответствующий физический адрес.

`BTC` (Branch Target Cache) - это кэш память, используемая BU для хранения адреса предыдущего перехода и первой инструкции, выполненной после перехода. Имеет целью без задержки заполнить конвейер инструкцией, если переход уже ранее состоялся. BTC может значительно повысить производительность процессора, учитывая время, которое он бы простаивал в ожидании заполнения конвейера после перехода.

1.2.4.7 Согласование кэшей в мультипроцессорных системах

Если несколько процессоров подсоединены к одной и той же шине адреса и данных и разделяют одну и ту же внешнюю память, то должен быть реализован определенный следящий механизм (snooping) для того, чтобы все внутрипроцессорные кэши всегда содержали `одни и те же` данные.

Рассмотрим, например, систему, содержащую два процессора, каждый из которых может брать управление общей шиной. Если процессор 1, управляющий в данный момент шиной, записывает в ячейку памяти, которая кэширована процессором 2, то данные в кэше последнего становятся устаревшими. Следящий механизм позволяет второму процессору отслеживать состояние шины адреса, даже если он не является в данный момент главным (т.е. управляющим внешней шиной). Если на шине появился адрес кэшированных данных, то эти данные помечаются в кэше как некорректные.

Когда второй процессор станет главным, он должен будет выбрать в случае необходимости обновленные данные из разделяемой памяти.

Если процессор 1, управляющий в данный момент шиной, читает из ячейки памяти, которая кэширована процессором 2, то возможно, что реальные данные находятся в кэше процессора 2 (еще не записаны в память, т.е. реализован кэш с обратной записью). Если это так, то следящий механизм инициирует цикл записи строки кэша процессора 2, содержащей затребованные процессором 1 данные, в разделяюмую память. После этого эти данные становятся доступными процессору 1 и цикл чтения процессора 1 продолжается.

1.2.5 Управление разделяемыми ресурсами

В многозадачной или(и) многопроцессорной системе доступ к критическим разделяемым ресурсам должен быть управляем. Установка семафора запрещает доступ к ресурсу другим задачам и процессорам, т.е. периферийные устройства и разделяемые данные могут быть доступны только одной задаче и процессору.

Задача или процессор, собирающийся взять управление разделяемым ресурсом, начинают с чтения значения семафора. Если он установлен, то задача или процессор должны ждать, пока ресурс станет доступным. Если семафор обнулен, то задача или процессор немедленно его устанавливают, чтобы показать, что контролируют ресурс. В процессе изменения семафора можно выделить три фазы: `чтение`, `изменение`, `запись`.

Если на стадии чтения возникнет переключение задач или другой процессор станет главным на шине, то может возникнуть ошибка, так как две задачи или два процессора контролируют один и тот же ресурс. Аналогично, если переключение контекста произойдет между циклом чтения и записи, то два процесса могут установить семафор, что тоже приведет к системной ошибке.

Для решения этой проблемы большинство процессоров имеют инструкцию, выполняющую неделимый цикл чтение-изменение-запись.

1.2.6 Прерывания

В большинстве случаев внешние события достигают процессора посредством прерываний. Основными характеристиками процессора при обработке прерываний являются

`Время реакции на прерывание`
: это время, которое проходит от поступления внешнего события (источника прерывания) до начала выполнения процедуры обработки прерывания.

`Детерминированность`
: это свойство времени реакции на внешние события быть предсказуемым. В-основном это свойство относится к программному обеспечению.

Системы реального времени, в-основном, и создаются для реакции на внешние события, поэтому эти характеристики являются критически важными для таких систем.

Описанные выше механизмы повышения производительности процессоров приводят к увеличению времени реакции: RISC процессоры имеют большое число регистров,

которые необходимо сохранить перед переходом к выполнению процедуры обработки прерывания; глубоко конвейеризированные процессоры, содержащие несколько исполняющих устройств, должны закончить обработку всех инструкций на конвейерах перед переходом к обработке прерывания.

Время реакции в-основном складывается из следующих четырех составляющих:

1. время, необходимое процессору для определения (типа) внешнего события;
2. время, необходимое для корректной остановки работающего в момент прерывания процесса (включая сохранение регистров);
3. время, необходимое для подготовки и загрузки процесса, обслуживающего прерывание;
4. время, необходимое для начала выполнения нового процесса.

Для уменьшения времени реакции на внешние события (прерывания), используют следующие решения.

- Таблица прерываний может храниться во внутренней памяти процессора, что делает не нужной выборку из внешней памяти (Intel 80960).
- Процессор может включать теневые регистры, что делает не нужным сохранение контекста текущей задачи в простых процедурах обработки прерываний (HP-PA).
- Критические процедуры обработки прерываний могут быть заблокированы в кэше инструкций (Motorola 68060).
- Таблица прерываний может хранить первые инструкции обработчика прерываний, что уменьшает простой конвейера (SPARC).

1.2.7 Многопроцессорность

Увеличение размера кэша имеет двоякие последствия. С одной стороны, это ускоряет выполнение программ и уменьшает обмен с памятью. С другой стороны, это делает системную память все более несоответствующей реальности, поскольку часть обновленных данных находится в кэше (еще не записаны в память). Поэтому современные процессоры включают в себя механизмы синхронизации, позволяющие нескольким процессорам разделять одни и те же шины. Эти механизмы призваны также максимально увеличить производительность за счет параллелизма.

Выделяют несколько типов построения многопроцессорных систем в зависимости от степени связи между отдельными процессорами в системе.

1.2.7.1 Сильно связанные процессоры (симметричные мультипроцессорные системы, SMP)

Все процессоры разделяют общую шины и общую память, могут выполнять одну и ту же задачу, причем задача может переходить от одного процессора другому. Если один процессор отказывает, он может быть заменен другим.

SMP подразумевает наличие аппаратного протокола синхронизации кэшей всех процессоров (см. выше).

Типичный пример: плата с двумя процессорами Pentium.

1.2.7.2 Слабо связанные процессоры

Часть системной памяти может быть разделяема, но переход задачи от одного процессора к другому невозможен.

Механизмы синхронизации специфичны для каждой системы (почтовые ящики, DPRAM, прерывания).

Типичный пример: стойка VME с несколькими процессорными платами и разделяемой памятью на одной из плат.

1.2.7.3 Распределенные процессоры

Несколько процессоров не разделяют ни одного общего ресурса, за исключением линии связи.

Типичный пример: соединенные посредством Ethernet рабочие станции.

1.2.7.4 Комбинированные архитектуры

Архитектура SMP является самой дорогой с точки зрения аппаратной реализации и самой дешевой с точки зрения разработки программного обеспечения. И наоборот, распределенные процессоры почти не требуют аппаратных затрат, но являются самым дорогим решением с точки зрения разработки ПО. Для достижения оптимального компромисса для круга решаемых задач используют различные комбинации описанных выше технологий.

`Гибридные схемы SMP`

: используются для ускорения работы программ, требующих интенсивной работы с оперативной памятью. Описанная выше схема SMP ускоряет выполнение задач, которым не нужен постоянный обмен с RAM, поскольку все процессоры разделяют одну шину и в каждый момент времени только один процессор может работать с памятью. Поэтому иногда (это очень дорогое решение) поступают следующим образом: система строится на базе модулей, каждый из которых является самостоятельной процессорной платой с двумя (или четырьмя) процессорами, включенными по схеме SMP. В каждом модуле расположена своя оперативная память, к которой через общую шину имеют доступ процессоры этого модуля. Логически память каждого модуля включена в общую память системы, т.е. память всей системы образована как объединение памяти каждого из модулей. Физически доступ к памяти других модулей осуществляется через быстродействующую коммуникационную шину, которая может быть как общей для всех модулей, так и быть соединением типа "точка-точка" по принципу "все со всеми" (обычно используется комбинация этих способов организации шины). Если программа может быть организована так, что в основном группе процессоров нужна не вся память, а только ее часть (размером с память модуля), и обмены между этими частями идут не часто (это весьма реалистичные предположения, особенно, если память каждого из модулей достаточно велика), то такая программа может эффективно работать на описанной архитектуре.

`Гибридные схемы слабо связанных (распределенных) процессоров`
: используются для повышения надежности работы систем на основе архитектуры со слабо связанными (распределенными) процессорами. Используются для ускорения работы систем, в которых требуется обрабатывать одновременно много процессов и обеспечить отказоустойчивость. Вместо слабо связанных (распределенных) процессоров используют слабо связанные (распределенные) процессорные модули, каждый из которых является самостоятельной процессорной платой с двумя (или четырьмя) процессорами, включенными по схеме SMP. В случае отказа одного из процессоров в модуле, может оказаться возможным передать все процессы оставшимся процессорам.

`Кластерная организация процессоров`
: является частным случаем описанных выше архитектур и используются для ускорения работы систем, в которых требуется обрабатывать одновременно много процессов. Физически кластер строится либо на базе распределенных процессоров, либо на базе слабо связанных процессоров (без разделяемой памяти, фактически несколько независимых процессорных систем, объединенных общим корпусом, источниками питания и системами ввода-вывода). Первый способ был более распространен в момент появления понятия "кластер", второй становится популярным в настоящее время в связи с миниатюризацией процессорных систем. Логически кластер представляет собой систему, в которой каждый из процессоров независим и может независимо от других принимать к исполнению задания. Операционная система направляет вновь пришедший процесс на исполнение тому процессору, который в данный момент менее загружен (т.е. для пользователя вся система выглядит как единая многопроцессорная установка).

  Если коммуникационный канал, связывающий отдельные процессорные модули, имеет высокую пропускную способность, то операционная система может эмулировать на такой системе поведение описанной выше системы гибридной SMP, а именно, она может позволить исполнять одно и то же приложение на нескольких процессорах. Это относится не только к кластерам, но и ко всем архитектурам с распределенными процессорами.

1.2.7.5 Обанкротившиеся архитектуры

В этом разделе мы рассмотрим многопроцессорные архитектуры, которые в чистом виде уже не применяются в новых разработках. Однако они оказали значительное влияние на развитие вычислительной техники и некоторые их черты, уже в новом технологическом воплощении, присутствуют и в современных архитектурах.

Основными причинами, приведшими к выходу из употребления этих архитектур являются

- необходимость разрабатывать программное обеспечение специально под конкретную архитектуру;
- конкуренция с параллельными архитектурами;

- снижение расходов на оборонные разработки (большинство проектов по высокопроизводительным вычислениям финансировалось из военных источников).

Основные архитектуры:

`Векторные процессоры`
: это не многопроцессорные архитектуры, а одиночные процессоры, способные выполнять операции с векторами как примитивные инструкции. Это может значительно ускорить выполнение программ вычислительной математики, поскольку реальные задачи работают с пространственными координатами (двух- и трехмерные векторы).

`Транспьютеры`
: это многопроцессорные архитектуры, состоящие из независимых процессоров (обычно кратных по количеству 4), каждый из которых имеет свою подсистему памяти. Процессоры связаны между собой высокоскоростными соединениями, организованными по принципу "точка-точка". Каждый процессор связан с четырьмя другими.

1.3 Организация доступа к внешней памяти

В этом разделе мы рассмотрим способы адресации оперативной памяти, а также кратко расскажем об организации самой подсистемы памяти.

1.3.1 Способы нумерации байтов внешней памяти

Если в 32-битной архитектуре минимальным адресуемым элементом оперативной памяти является 32-битное слово, то вопрос о нумерации байтов в слове не встает. Рассмотрим ситуацию, когда минимальным адресуемым элементом оперативной памяти является байт (8 битное слово). В этом случае данные большего размера образуются как объединение подряд идущих байт. Выбор нумерации байт в 32 битном (4 байта) слове может быть произвольным, что дает 24=4! способа. На практике используются только два: ABCD (big-endian) и DCBA (little-endian).

В big-endian модели байты в слове нумеруются от наиболее значимого к наименее значимому. В little-endian модели байты в слове нумеруются от наименее значимого к наиболее значимому. Примеры big-endian процессоров: Motorola 68xxx, PowerPC (по умолчанию), SPARC, пример little-endian процессора: Intel 80x86. Процессоры PowerPC, Intel 80960, ARM, SPARC (64-битные модели) могут работать как big-endian режиме, так и в little-endian.

Внутри байта также можно вводить различные порядки. В настоящее время используется только два: биты в байте нумеруются от старшего к младшему (PowerPC), или биты в байте нумеруются от младшему к старшему (Motorola 68xxx, Intel 80x86, SPARC, Intel 80960, ARM).

1.3.2 Организация данных с плавающей точкой

Все современные процессоры поддерживают стандарт ANSI/IEEE 754-1985 в организации данных с плавающей точкой.

`floating-point single`

- Размер - 4 байта,
- знак - бит 31 (1 бит)
- показатель - биты 23...30 (8 бит)
- мантисса - биты 0...22 (23 бит)

Нормализованное значение (при показателе, большем нуля и меньшем 255) - (-1) в степени знак умножить на 2 в степени (показатель - 127) умножить на 1.мантисса.

`floating-point double`

- Размер - 8 байт,
- знак - бит 63 (1 бит)
- показатель - биты 52...62 (11 бит)
- мантисса - биты 0...51 (52 бит)

Нормализованное значение (при показателе, большем нуля и меньшем 2047) - (-1) в степени знак умножить на 2 в степени (показатель - 1023) умножить на 1.мантисса.

`floating-point quad`

(поддерживается не всеми процессорами)

- Размер - 16 байт,
- знак - бит 127 (1 бит)
- показатель - биты 112...126 (15 бит)
- мантисса - биты 0...111 (112 бит)

Нормализованное значение (при показателе, большем нуля и меньшем 32767) - (-1) в степени знак умножить на 2 в степени (показатель - 16382) умножить на 1.мантисса.

1.3.3 Пути повышения производительности оперативной памяти

Для повышения производительности оперативной памяти памяти применяют несколько приемов.

- Увеличение ширины шины данных: переход от SIMM к DIMM модулям при построении подсистемы памяти.
- Введение небольшой статической памяти SRAM для буферизации DRAM модулей: буферизованные DIMM модули.
- Введение конвейера в модули DRAM: SDRAM.
- "Расслоение" оперативной памяти. Поскольку процессор обменивается с памятью только блоками размером со строку кэша, то можно разделить этот блок на N частей (N обычно 2, 4, 8) и передать каждую из частей своей подсистеме памяти. В результате получаются N подсистем памяти, работающих параллельно. Если программа требует последовательные адреса памяти (т.е. стратегия предвыборки строки кэша себя оправдывает), то этот подход может в N раз увеличить производительность подсистемы памяти.

2 Архитектура процессоров Motorola 68xxx

2.1 Общий обзор процессоров Motorola 68xxx

Прежде всего, семейство процессоров Motorola 68xxx (сокращенно M68k) характеризуется простотой реализации как аппаратных, так и программных решений на его базе. Первым процессором семейства являлся хорошо известный M68000. Хотя он появился в 1979 году, он является процессором с полной 32-битной внутренней архитектурой. Подобно всем последующим процессорам, его линейная организация памяти и единое с памятью пространство ввода/вывода облегчает аппаратные и программные разработки. Наконец, специальные выводы процессора устанавливаются на каждом цикле шины для того, чтобы различить супервизорское и пользовательское адресные пространства. Если эти выводы декодировать при вычислении адреса, то системные ресурсы будут изолированы от несанкционированного доступа пользовательских программ.

Следующие поколения процессоров M68k, включая M68060, с программной точки зрения вносили только новые режимы адресации памяти и некоторые дополнительные инструкции, и поэтому программируются так же легко, как M68000. Эволюция затронула аппаратную архитектуру процессоров: в их составе появились и развивались конвейеры, кэши, MMU, FPU и т.д.

Основываясь на архитектуре M68k Motorola разработала семейство контроллеров MC683xx, которые разделяют на 3 группы: группу 68000, CPU32 и CPU32+. В этих контроллерах вычислительная мощность M68k сочетается с интегрированными периферийными процессорами, образуя высокопроизводительные контроллеры. Наиболее специализированные, ориентированные на ввод/вывод контроллеры (68302 и 68360) включают непрограммируемый RISC процессор, управляющий последовательным коммуникационным каналом. Это дает возможность одному устройству управлять таким сложным последовательным каналом, как Ethernet или ISDN.

Все члены семейства MC683xx имеют межмодульную шину (intermodule bus, IMB). IMB обеспечивает общий интерфейс для всех модулей семейства MC683xx, что позволяет фирме Motorola быстро разрабатывать новые устройства, используя библиотеки существующих модулей.

2.2 Основные члены семейства Motorola 68xxx

Каждый процессор в приведенных ниже технических данных обладает всеми возможностями предыдущих процессоров для обеспечения совместимости.

M68000 (1979, 68000 транзисторов, макс. частота: 16.67MHz, 1 MIPS)

- асинхронные передачи данных (кроме области интерфейса 6800)
- 16-битная шина данных, отсутствует динамическое изменение ширины шины
- 24-битная шина адреса
- 14 режимов адресации памяти (включая косвенные регистровые)

- 15 32-битных регистров общего назначения
- два уровня привилегий: пользовательский и супервизорский
- два указателя стека для разделения пользовательского и супервизорского стеков
- фиксированная в памяти таблица прерываний (начиная с адреса 0)
- одна неделимая инструкция TAS (Test And Set) для установки семафоров

M68010 (1983, 68000 транзисторов, макс. частота: 16.67MHz, 1 MIPS)

- перемещаемая таблица прерываний
- поддержка механизма виртуальной памяти

M68020 (1984, 195000 транзисторов, макс. частота: 25MHz, 5 MIPS)

- асинхронные передачи данных
- 32-битная шина данных, динамическое изменение ширины шины (адаптируется к 8-ми, 16-ти и 32-х битным обменам с внешними устройствами)
- 32-битная шина адреса
- 18 режимов адресации памяти (включая косвенные через память)
- 256-байт кэш инструкций
- 3-х стадийный конвейер, что позволяет одновременно обрабатывать до трех слов одной операции или три последовательные инструкции
- интерфейс сопроцессора, что позволяет подключить внешнее FPU (MC68881/MC68882) и MMU (MC68851)
- дополнительный указатель стека для разделения аппаратных и программных прерываний
- две дополнительных неделимых инструкции: CAS и CAS2 (Compare And Swap 32 или 64 бита) для установки семафоров
- новые инструкции для работы с битовыми полями

M68030 (1986, 300000 транзисторов, макс. частота: 50MHz, 8 MIPS)

- Гарвардская архитектура
- два различных кэша: 256-байт кэш инструкций + 256-байт кэш данных
- асинхронные передачи данных
- синхронный интерфейс, что позволяет осуществлять блочные (burst) передачи в/из кэша
- MMU, позволяющее работать со страницами размером от 256 байт до 256 килобайт

M68040 (1989, 1200000 транзисторов, макс. частота шины: 40MHz, 8 MIPS, 3.5Mflops)

- синхронные передачи данных

- отсутствует динамическое изменение ширины шины
- частота процессора равна удвоенной частоте шины (максимально 80MHz)
- 6-ти стадийный конвейер
- FPU
- 4-килобайт кэш инструкций + 4-килобайт кэш данных
- механизм синхронизации шины (bus snooping) (арбитраж шины должен быть внешним, процессор имеет вывод запроса шины), это обеспечивает согласование кэшей в многопроцессорных системах

M68060 (1994, 2500000 транзисторов, макс. частота шины: 66MHz, 100 MIPS)

- суперконвейерный, суперскалярный 32 битный гибридный CISC-RISC процессор
- набор инструкций M68040 реализован аппаратной логикой, а не микрокодом
- основные устройства: буфер инструкций, 4-х стадийное конвейерное устройство предвыборки, два 4-х стадийных конвейерных целочисленных устройства, устройство переходов, FPU повышенной точности
- кэш переходов (BTC)
- поток инструкций разделяется на два конвейера на FIFO стадии
- устройство предвыборки преобразует входной поток M680x0 инструкций, имеющих переменную длину, в поток RISC инструкций фиксированной длины
- M68060 может может исполнять за один цикл 4 инструкции M680x0: две целочисленных инструкции, одну инструкцию перехода и одну инструкцию с плавающей точкой; этот параллелизм обеспечивает высокую скорость исполнения даже для кода, не перекомпилированного специально для M68060
- 4-килобайт кэш инструкций + 4-килобайт кэш данных
- автоматическое уменьшение потребляемой мощности: внутренние функциональные блоки автоматически выключаются, если они не используются в течении ряда циклов

M68302 (микроконтроллер группы 68000)

- IMP (Integrated Multiprotocol Processor)
- M68302 состоит из процессора M68000, System Integration Block (SIB) и Communication Processor (CP)
- SIB содержит контроллер DMA, два 16-битных таймера общего назначения, контроллер памяти и контроллер прерываний
- CP является выделенным RISC процессором, обслуживающим 6 последовательных портов, и отвечает за работу с последовательными каналами по выбранному пользователем протоколу (ISDN, UART, HDLC, BSC и другие)

`M68360 (или QUICC) (микроконтроллер группы CPU32+)`

- M68360 состоит из процессора CPU32+, SIM60 (System Integration Module) и CPM (Communication Processor Module)
- процессор CPU32+ является процессором M68020 без кэша и сопроцессорного интерфейса; система команд процессора M68020 расширена CPU32+ специфическими инструкциями табличной интерполяции
- SIM60 интегрирует основные устройства общего назначения, которые могут быть полезны в любых 32-битных процессорных системах: синтезатор частоты, таймеры-будильники, таймер периодического прерывания, контроллер памяти, способный без дополнительной логики управлять 32-битными DRAM
- CPM содержит 2 DMA контроллера, 4 таймера общего назначения, контроллер прерываний, 7 коммуникационных контроллеров, управляющих 7-ю последовательными физическими каналами
- поддерживаются протоколы UART, ISDN, HDLC, BSC, AppleTalk
- протокол Ethernet поддерживается в версии MC68EN360

2.3 Программная модель семейства Motorola 68xxx

В этом разделе мы рассмотрим процессоры семейства Motorola 68xxx с точки зрения программиста (или компилятора).

2.3.1 Набор регистров процессоров Motorola 68xxx

Прикладной программе доступны 16 регистров общего назначения и несколько служебных регистров.

`Регистры d0 - d7 (регистры данных)`

Могут содержать целочисленные данные следующих типов

1. Бит (M68020 и выше, только инструкции, работающие с битовыми полями)
2. Двоично-закодированные десятичные числа (BCD); байт содержит одну цифру, существуют инструкции, работающие с двумя цифрами в одном байте
3. Байт (8 бит); при записи в регистр старшая часть не используется и не изменяется
4. Слово (16 бит); при записи в регистр старшая часть не используется и не изменяется
5. Длинное слово (32 бит)
6. Счетверенное слово (64 бит); используются любые два регистра, над такими операндами определена только инструкция пересылки (MOVEM).

`Регистры a0 - a7 (регистры данных)`

Могут содержать целочисленные данные следующих типов

1. Слово (16 бит); при записи в регистр старшая часть заполняется знаковым битом источника
2. Длинное слово (32 бит)

`Регистр pc (program counter)`
: Содержит адрес следующей инструкции. При записи в этот регистр происходит переход по адресу, который был записан.

`Регистр кода условия ccr (condition code register)`
: является частью регистра статуса (status register, SR), который не доступен пользовательской программе как регистр. Коды условия:

  `X - extend`
  : используется в командах сдвига и арифметических операциях, не используется в командах условного перехода

  `N - negative`
  : устанавливается в 1, если результат меньше 0, иначе - 0

  `Z - zero` устанавливается в 1, если результат равен 0, иначе - 0

  `V - overflow`
  : устанавливается в 1, если результат не входит в диапазон представимых значений, иначе - 0

  `C - carry` устанавливается в 1, если в результате произошел перенос в самом старшем разряде, иначе - 0

  Коды условия устанавливаются по результату арифметических операций или специальными инструкциями, и используются в командах условного перехода.

Следующие специальные регистры недоступны пользовательской программе и их набор различен у разных моделей:

`SSP (Supervisor Stack Pointer)`
: указатель стека супервизора (иногда называемый Master Stack Pointer)

`ISP (Interrupt Stack Pointer)`
: указатель стека прерываний (M68020 и выше)

`VBR (Vector Base Register)`
: регистр базы таблицы прерываний (M68010 и выше)

`SR (Status Register)`
: регистр статуса, содержит CCR

`SFC, DFC (Alternate Function Code Registers)`
: позволяют супервизору получать доступ к любому адресному пространству

`CACR (CAshe Control Register), CAAR (CAshe Address Register)`
: регистры, управляющие кэшем (M68020 и выше)

2.3.2 Режимы адресации памяти процессоров Motorola 68xxx

Поддерживаются следующие режимы

1. Data Register Direct
   - Ассемблерный синтаксис: `Dn`
   - Значение: `Dn`
2. Address Register Direct
   - Ассемблерный синтаксис: `An`
   - Значение: `An`
3. Address Register Indirect
   - Ассемблерный синтаксис: `An@`
   - Значение: ячейка памяти с адресом `An`
4. Address Register Indirect with Postincrement
   - Ассемблерный синтаксис: `An@+`
   - Значение: ячейка памяти с адресом `An`, регистр `An` увеличивается на размер операнда (т.е. этой ячейки памяти)
5. Address Register Indirect with Predecrement
   - Ассемблерный синтаксис: `An@-`
   - Значение: вначале регистр `An` уменьшается на размер операнда, значением является ячейка памяти с адресом `An`
6. Address Register Indirect with Displacement
   - Ассемблерный синтаксис: `An@(d16)`
   - Значение: ячейка памяти с адресом `An+d16`
7. Address Register Indirect with Index (8-bit Displacement)
   - Ассемблерный синтаксис: `An@(d8,Xn)`
   - Значение: ячейка памяти с адресом `An+Xn+d8`
8. Address Register Indirect with Index (Base Displacement) (M68020 и выше)
   - Ассемблерный синтаксис: `An@(bd,Xn)`
   - Значение: ячейка памяти с адресом `An+Xn+bd`
9. Memory Indirect Postindexed (M68020 и выше)
   - Ассемблерный синтаксис: `An@(bd)@(od,Xn)`
   - Значение: ячейка памяти с адресом `An@(bd)+Xn+od`
10. Memory Indirect Preindexed (M68020 и выше)
    - Ассемблерный синтаксис: `An@(bd,Xn)@(od)`
    - Значение: ячейка памяти с адресом `An@(bd,Xn)+od`
11. Program Counter Indirect with Displacement
    - Ассемблерный синтаксис: `PC@(d16)`
    - Значение: ячейка памяти с адресом `PC+d16`
12. Program Counter Indirect with Index (8-bit Displacement)

   - Ассемблерный синтаксис: `PC@(d8,Xn)`
   - Значение: ячейка памяти с адресом `PC+Xn+d8`
13. Program Counter Indirect with Index (Base Displacement) (M68020 и выше)
   - Ассемблерный синтаксис: `PC@(bd,Xn)`
   - Значение: ячейка памяти с адресом `PC+Xn+bd`
14. Program Counter Memory Indirect Postindexed (M68020 и выше)
   - Ассемблерный синтаксис: `PC@(bd)@(od,Xn)`
   - Значение: ячейка памяти с адресом `PC@(bd)+Xn+od`
15. Program Counter Memory Indirect Preindexed (M68020 и выше)
   - Ассемблерный синтаксис: `PC@(bd,Xn)@(od)`
   - Значение: ячейка памяти с адресом `PC@(bd,Xn)+od`
16. Absolute Short
   - Ассемблерный синтаксис: `(xxx):W`
   - Значение: ячейка памяти с адресом `(xxx):W`
17. Absolute Long
   - Ассемблерный синтаксис: `(xxx):L`
   - Значение: ячейка памяти с адресом `(xxx):L`
18. Immediate
   - Ассемблерный синтаксис: `#(data)`
   - Значение: `#(data)`

Здесь использованы следующие обозначения:

- `Dn` - регистр данных, `D0-D7`
- `An` - адресный регистр, `A0-A7`
- `d8`, `d16` - знакорасширяемые смещения, размер 8 (`d8`) или 16 (`d16`) бит; в случае отсутствия ассемблер использует значение 0
- `Xn` - адресный регистр или регистр данных, используемые в качестве индекса; у M68020 и выше может иметь форму `Xn`.SIZE*SCALE, где SIZE = W или L (размер 16 или 32 бит), SCALE = 1, 2, 4 или 8 (индексный регистр умножается на SCALE); SIZE и SCALE могут быть опущены
- `bd` - знакорасширяемое смещение, размер 16 или 32 бит
- `od` - знакорасширяемое смещение при адресации через память, размер 16 или 32 бит
- `data` - непосредственное значение, размер 8, 16 или 32 бита.

2.3.3 Основные инструкции процессоров Motorola 68xxx

Инструкции процессоров Motorola 68xxx имеют от нуля до трех операндов. Большинство инструкций (пересылки, арифметические, логические) имеют два операнда, один из которых не изменяется в операции (источник), а другой является результатом операции (приемник). Обычно используется синтаксис, в котором источник является левым операндом, а приемник - правым.

Существует несколько инструкций, неявно использующих указатель стека a7. Это стековые операции и вызовы функций, см. ниже описание таких инструкций, как LINK, UNLK, PEA, BSR, RTS.

Рассмотрим основные группы инструкций процессоров Motorola 68xxx.

2.3.3.1 Инструкции пересылки данных процессоров Motorola 68xxx

EXG

- Синтаксис операндов: Rn, Rm
- Размер операндов: 32
- Операция: обменять значения регистров Rn и Rm

LEA

- Синтаксис операндов: <ea>, An
- Размер операндов: 32
- Операция: <ea> -> An

LINK

- Синтаксис операндов: An, #<d>
- Размер операндов: 16, 32
- Операция: SP-4 -> SP, An -> SP@, SP -> An, SP+<d> -> SP

MOVE

- Синтаксис операндов: <ea>, <ea>
- Размер операндов: 8, 16, 32
- Операция: источник -> приемник

MOVE16 (M68040 и выше)

- Синтаксис операндов: <ea>, <ea>
- Размер операндов: 16 байт
- Операция: выровненный 16-байтовый блок -> приемник

MOVEA

- Синтаксис операндов: <ea>, An
- Размер операндов: 16, 32 -> 32
- Операция: источник -> приемник

MOVEM

- Синтаксис операндов: list, <ea> ; <ea>, list
- Размер операндов: 16, 32 ; 16, 32 -> 32
- Операция: перечисленные регистры -> приемник, источник -> перечисленные регистры

MOVQ

- Синтаксис операндов: #<data>, Dn
- Размер операндов: 8 -> 32
- Операция: #<data> -> Dn

PEA

- Синтаксис операндов: <ea>
- Размер операндов: 32
- Операция: SP-4 -> SP, <ea> -> SP@

UNLK

- Синтаксис операндов: An
- Размер операндов: 32
- Операция: An -> SP, SP@ -> An, SP+4 -> SP

2.3.3.2 Целочисленные арифметические инструкции процессоров Motorola 68xxx

ADD

- Синтаксис операндов: Dn, <ea>; <ea>, Dn
- Размер операндов: 8, 16, 32
- Операция: источник + приемник -> приемник

ADDA

- Синтаксис операндов: <ea>, An
- Размер операндов: 16, 32
- Операция: источник + приемник -> приемник

ADDI
ADDQ

- Синтаксис операндов: #<data>, <ea>
- Размер операндов: 8, 16, 32
- Операция: источник + приемник -> приемник

ADDX

- Синтаксис операндов: Dn, Dm; An@-, Am@-
- Размер операндов: 8, 16, 32
- Операция: источник + приемник + X -> приемник

CLR

- Синтаксис операндов: <ea>
- Размер операндов: 8, 16, 32
- Операция: 0 -> приемник

`CMP`

- Синтаксис операндов: `<ea>, Dn`
- Размер операндов: 8, 16, 32
- Операция: приемник - источник

`CMPA`

- Синтаксис операндов: `<ea>, An`
- Размер операндов: 16, 32
- Операция: приемник - источник

`CMPI`

- Синтаксис операндов: `#<data>, <ea>`
- Размер операндов: 8, 16, 32
- Операция: приемник - источник

`CMPM`

- Синтаксис операндов: `An@+, Am@+`
- Размер операндов: 8, 16, 32
- Операция: приемник - источник

`CMP2`

- Синтаксис операндов: `<ea>, Rn`
- Размер операндов: 8, 16, 32
- Операция: нижняя граница <= Rn <= верхняя граница

`DIVS/DIVU`

- Синтаксис операндов: `<ea>, Dn`; `<ea>, Dn:Dm`; `<ea>, Dn`
- Размер операндов: 32/16 -> 16:16; 64/32 -> 32:32; 32/32 -> 32
- Операция: приемник / источник -> приемник (знаково или нет)

`DIVSL/DIVUL`

- Синтаксис операндов: `<ea>, Dn:Dm`
- Размер операндов: 32/32 -> 32:32
- Операция: приемник / источник -> приемник (знаково или нет)

`EXT`

- Синтаксис операндов: `Dn`
- Размер операндов: 8 -> 16; 16 -> 32
- Операция: знаковое расширение `Dn -> Dn`

`EXTB`

- Синтаксис операндов: `Dn`
- Размер операндов: 8 -> 32

- Операция: знаковое расширение `Dn -> Dn`

`MULS/MULU`

- Синтаксис операндов: `<ea>, Dn`; `<ea>, Dn`; `<ea>, Dn:Dm`
- Размер операндов: 16x16 `->` 32; 32x32 `->` 32; 32x32 `->` 64
- Операция: приемник x источник `->` приемник (знаково или нет)

`NEG`

- Синтаксис операндов: `<ea>`
- Размер операндов: 8, 16, 32
- Операция: 0 - приемник `->` приемник

`NEGX`

- Синтаксис операндов: `<ea>`
- Размер операндов: 8, 16, 32
- Операция: 0 - приемник - X `->` приемник

`SUB`

- Синтаксис операндов: `Dn, <ea>`; `<ea>, Dn`
- Размер операндов: 8, 16, 32
- Операция: приемник - источник `->` приемник

`SUBA`

- Синтаксис операндов: `<ea>, An`
- Размер операндов: 16, 32
- Операция: приемник - источник `->` приемник

`SUBI, SUBQ`

- Синтаксис операндов: `#<data>, <ea>`
- Размер операндов: 8, 16, 32
- Операция: приемник - источник `->` приемник

`SUBX`

- Синтаксис операндов: `Dn, Dm`; `An@-, Am@-`
- Размер операндов: 8, 16, 32
- Операция: приемник - источник - X `->` приемник

2.3.3.3 Логические инструкции процессоров Motorola 68xxx

`AND`

- Синтаксис операндов: `<ea>, Dn`; `Dn, <ea>`
- Размер операндов: 8, 16, 32
- Операция: источник `AND` приемник `->` приемник

`ANDI`

- Синтаксис операндов: `#<data>, <ea>`
- Размер операндов: 8, 16, 32
- Операция: источник `AND` приемник `->` приемник

`EOR`

- Синтаксис операндов: `Dn, <ea>`
- Размер операндов: 8, 16, 32
- Операция: источник `EOR` приемник `->` приемник

`EORI`

- Синтаксис операндов: `#<data>, <ea>`
- Размер операндов: 8, 16, 32
- Операция: источник `EOR` приемник `->` приемник

`NOT`

- Синтаксис операндов: `<ea>`
- Размер операндов: 8, 16, 32
- Операция: `NOT` приемник `->` приемник

`OR`

- Синтаксис операндов: `<ea>, Dn`; `Dn, <ea>`
- Размер операндов: 8, 16, 32
- Операция: источник `OR` приемник `->` приемник

`ORI`

- Синтаксис операндов: `#<data>, <ea>`
- Размер операндов: 8, 16, 32
- Операция: источник `OR` приемник `->` приемник

2.3.3.4 Инструкции сдвига процессоров Motorola 68xxx

`ASL`

- Синтаксис операндов: `Dm, Dn`; `#<data>, Dn`; `<ea>`
- Размер операндов: 8, 16, 32; 8, 16, 32; 16
- Операция: сдвинуть приемник влево на `Dm`, `#<data>`, 1 бит соответственно, вдвигаются нули

`ASR`

- Синтаксис операндов: `Dm, Dn`; `#<data>, Dn`; `<ea>`
- Размер операндов: 8, 16, 32; 8, 16, 32; 16
- Операция: сдвинуть приемник вправо на `Dm`, `#<data>`, 1 бит соответственно, вдвигается знаковый бит

`LSL`

- Синтаксис операндов: `Dm, Dn`; `#<data>, Dn`; `<ea>`
- Размер операндов: 8, 16, 32; 8, 16, 32; 16
- Операция: совпадает с `ASL`

`LSR`

- Синтаксис операндов: `Dm, Dn`; `#<data>, Dn`; `<ea>`
- Размер операндов: 8, 16, 32; 8, 16, 32; 16
- Операция: сдвинуть приемник вправо на `Dm`, `#<data>`, 1 бит соответственно, вдвигаются нули

`ROL`

- Синтаксис операндов: `Dm, Dn`; `#<data>, Dn`; `<ea>`
- Размер операндов: 8, 16, 32; 8, 16, 32; 16
- Операция: циклически сдвинуть приемник влево на `Dm`, `#<data>`, 1 бит соответственно

`ROR`

- Синтаксис операндов: `Dm, Dn`; `#<data>, Dn`; `<ea>`
- Размер операндов: 8, 16, 32; 8, 16, 32; 16
- Операция: циклически сдвинуть приемник влево на `Dm`, `#<data>`, 1 бит соответственно

`ROXL`

- Синтаксис операндов: `Dm, Dn`; `#<data>, Dn`; `<ea>`
- Размер операндов: 8, 16, 32; 8, 16, 32; 16
- Операция: циклически сдвинуть приемник влево на `Dm`, `#<data>`, 1 бит соответственно, X (extend) бит участвует в ротации

`ROXR`

- Синтаксис операндов: `Dm, Dn`; `#<data>, Dn`; `<ea>`
- Размер операндов: 8, 16, 32; 8, 16, 32; 16
- Операция: циклически сдвинуть приемник влево на `Dm`, `#<data>`, 1 бит соответственно, X (extend) бит участвует в ротации

`SWAP`

- Синтаксис операндов: `Dm`
- Размер операндов: 32
- Операция: обменять местами слова в `Dm`

2.3.3.5 Инструкции управления процессоров Motorola 68xxx

Условные переходы:

`Bcc`

- Синтаксис операндов: `<label>`
- Размер операндов: 8, 16, 32
- Операция: если условие истинно, то `PC + offset(<label>) -> PC`

`DBcc`

- Синтаксис операндов: `Dn, <label>`
- Размер операндов: 16
- Операция: если условие истинно, `Dn-1 -> Dn`, если `Dn` не равен -1, то `PC + offset(<label>) -> PC`

`Scc`

- Синтаксис операндов: `<ea>`
- Размер операндов: 8
- Операция: если условие истинно, то 1' `->` приемник, иначе 0' `->` приемник

Коды условия `cc` здесь

- CC - carry clear
- CS - carry set
- EQ - equal
- GE - greater or equal
- GT - greater than
- HI - high
- LE - less or equal
- LS - low or same
- LT - less than
- MI - minus
- NE - not equal
- PL - plus
- VC - overflow clear
- VS - overflow set

В инструкциях `DBcc` и `Scc` могут использоваться также коды

- T - always true
- F - never true

Безусловные переходы:

`BRA`

- Синтаксис операндов: `<label>`
- Размер операндов: 8, 16, 32
- Операция: `PC + offset(<label>) -> PC`

`BSR`

- Синтаксис операндов: `<label>`
- Размер операндов: 8, 16, 32
- Операция: `SP-4 -> SP`, `PC -> (SP)@`, `PC + offset(<label>) -> PC`

`JMP`

- Синтаксис операндов: `<ea>`
- Размер операндов: нет (всегда 32)
- Операция: приемник `-> PC`

`JSR`

- Синтаксис операндов: `<label>`
- Размер операндов: нет (всегда 32)
- Операция: `SP-4 -> SP`, `PC -> (SP)@`, приемник `-> PC`

`NOP`

- Синтаксис операндов: нет
- Размер операндов: нет
- Операция: `PC+2 -> PC`

Возврат из процедуры

`RTD`

- Синтаксис операндов: `#<d>`
- Размер операндов: 16
- Операция: `(SP)@ -> PC`, `SP+4+<d> -> SP`

`RTR`

- Синтаксис операндов: нет
- Размер операндов: нет
- Операция: `(SP)@ -> CCR`, `SP+2 -> SP`, `(SP)@ -> PC`, `SP+4 -> SP`

`RTS`

- Синтаксис операндов: нет
- Размер операндов: нет
- Операция: `(SP)@ -> PC`, `SP+4 -> SP`

Проверка операндов:

`TST`

- Синтаксис операндов: `<ea>`
- Размер операндов: 8, 16, 32
- Операция: установить коды условия

2.3.3.6 Мультипроцессорные инструкции процессоров Motorola 68xxx

`CAS`

- Синтаксис операндов: `Dn, Dm, <ea>`
- Размер операндов: 8, 16, 32
- Операция: приемник - `Dn -> CC`, если `Z`, то `Dm ->` приемник, иначе приемник `-> Dn`

`CAS2`

- Синтаксис операндов: `Dn1:Dn2, Dm1:Dm2, (Rn)@:(Rm)@`
- Размер операндов: 16, 32
- Операция: то же, что `CAS`, но с двойными операндами

`TAS`

- Синтаксис операндов: `<ea>`
- Размер операндов: 8
- Операция: приемник - `0 -> CC`, `1 ->` бит 7 приемника

2.4 Примеры кода для семейства Motorola 68xxx

Мы используем ассемблерный синтаксис и соглашения о вызовах операционной системы SunOS для SUN-3. Основные соглашения:

`Stack Pointer (SP)` **(диктуется системой команд)**
= регистр `a7`

`Frame Pointer (FP)`
= регистр `a6`

`Адрес возврата из функции` **(диктуется системой команд)**
= `mem(SP)` (т.е. ячейка памяти с адресом `SP`), здесь `SP` - в точке входа в функцию

`Возвращаемое значение функции`
= регистр `d0`

`Аргументы функции`
= `mem(SP+4)`, `mem(SP+8)`, ..., здесь `SP` - в точке входа в функцию

`Регистры, не сохраняемые при вызове функции`
= `d0`, `d1`, `a0`, `a1`

`Регистры, сохраняемые при вызове функции`
= `d2`, ..., `d7`, `a2`, ..., `a7`

1.
   - Исходный текст

     ```
     int f () { return 0; }
     ```

   - Ассемблерный код

```
_f:
        link a6,#0
        clrl d0
        unlk a6
        rts
```

2.

- Исходный текст

```
extern int a;
int f () { return a; }
```

- Ассемблерный код

```
_f:
        link a6,#0
        movel _a,d0
        unlk a6
        rts
```

3.

- Исходный текст

```
extern int a;
int * f () { return &a; }
```

- Ассемблерный код

```
_f:
        link a6,#0
        movel #_a,d0
        unlk a6
        rts
```

4.

- Исходный текст

```
int f (int a) { return a; }
int g () { return f (1); }
```

- Ассемблерный код

```
_f:
        link a6,#0
        movel a6@(8),d0
        unlk a6
        rts
_g:
        link a6,#0
        pea 1:w
        jbsr _f
        unlk a6
        rts
```

5.

- Исходный текст

```
int f (int a, int b) { return a + b; }
```

```
extern int a;
int g () { return f (1, a); }
```

– Ассемблерный код

```
_f:
        link a6,#0
        movel a6@(8),d0
        addl a6@(12),d0
        unlk a6
        rts
_g:
        link a6,#0
        movel _a,sp@-
        pea 1:w
        jbsr _f
        unlk a6
        rts
```

6.

– Исходный текст

```
int f (int * a, int n)
{
  int i, s;
  for (i = 0, s = 0; i < n; i++)
    s += a[i];
  return s;
}
extern int a[];
extern int n;
int g ()
{
  return f (a, n);
}
```

– Ассемблерный код

```
_f:
        link a6,#0
        movel a6@(12),a1
        clrl d1
        clrl d0
        cmpl d0,a1
        jle L3
        movel a6@(8),a0
L5:
        addl a0@+,d0
        addql #1,d1
        cmpl d1,a1
        jgt L5
L3:
        unlk a6
```

```
        rts
_g:
        link a6,#0
        movel _n,sp@-
        pea _a
        jbsr _f
        unlk a6
        rts
```

7.

– Исходный текст

```
int f (int * a, int * b, int n)
{
  int i, s;
  for (i = 0, s = 0; i < n; i++)
    s += a[i] * b[i];
  return s;
}
```

– Ассемблерный код

```
_f:
        link a6,#0
        movel d3,sp@-
        movel d2,sp@-
        movel a6@(16),d3
        clrl d1
        clrl d2
        cmpl d2,d3
        jle L3
        movel a6@(12),a1
        movel a6@(8),a0
L5:
        movel a0@+,d0
        mulsl a1@+,d0
        addl d0,d2
        addql #1,d1
        cmpl d1,d3
        jgt L5
L3:
        movel d2,d0
        movel a6@(-8),d2
        movel a6@(-4),d3
        unlk a6
        rts
```

8.

– Исходный текст

```
int f (int a, int b, int c)
{
  int max;
```

```
  if (a >= b && a >= c)
    max = a;
  else if (b >= c)
    max = b;
  else
    max = c;
  return max;
}
```

– Ассемблерный код

```
_f:
        link a6,#0
        movel a6@(8),d0
        movel a6@(12),d1
        movel a6@(16),a0
        cmpl d0,d1
        jgt L2
        cmpl d0,a0
        jle L3
L2:
        movel a0,d0
        cmpl d1,d0
        jgt L3
        movel d1,d0
L3:
        unlk a6
        rts
```

3 Архитектура процессоров Intel 80x86

3.1 Общий обзор процессоров Intel 80x86

Первый представитель семейства - i8086, появился в 1979г. Это был 16-ти битный процессор, обеспечивающий совместимость с предыдущими 8-ми битными процессорами i8080 и i8085. Это наложило определенный отпечаток на программную архитектуру i8086:

- устаревший набор инструкций,
- множество нелогичных ограничений на операнды.

Основные общие черты семейства i80x86

- пространство ввода/вывода отделено от пространства памяти,
- сегментная организация памяти,
- малое число регистров,
- невзаимозаменяемость регистров (много инструкций, неявно использующих жестко закрепленные регистры).

Процессор i80386 был первым 32-х битным процессором в семействе и уже мог работать с плоской моделью памяти. Однако, этот процессор сохранил программную совместимость с предыдущими членами семейства. Это еще больше увеличило его сложность и законсервировало устаревшую программную архитектуру i8086.

3.2 Основные члены семейства Intel 80x86

Каждый процессор в приведенных ниже технических данных обладает всеми возможностями предыдущих процессоров для обеспечения совместимости.

`i80386 (1987, 275000 транзисторов, 33MHz, 8 MIPS)`

- `32-бит` шина данных и `32-бит` шина адреса
- 8 32-битных регистров общего назначения
- динамическое изменение ширины шины (16 или 32 бит)
- пространство памяти отделено от 64Кб пространства ввода/вывода
- буфер предвыборки 16 байт
- MMU (страничное или(и) сегментное) с кэшем результатов трансляции; может работать в 3-х режимах:
  - `real mode` : 24 бит сегментный адрес, нет трансляции
  - `protected mode` : 32 бит адрес, страничное или(и) сегментное преобразование
  - `virtual 8086 mode` : 24 бит сегментный адрес, страничная трансляция
- внешнее FPU i80387
- внешний контроллер (внешней) кэш памяти i82385

- поддержка многозадачности посредством регистра задачи и дескрипторов задач

i80486 (1989, 1200000 транзисторов, частота шины 33MHz, 20 MIPS)

- внутренняя частота процессора равна либо частоте шины (DX 33MHz), либо удвоенной частоте шины (DX2 66MHz), либо утроенной частоте шины (DX4 100MHz)
- единый 8Кб (16Кб) кэш инструкций и данных (с поддержкой блоковых передач (burst access))
- кэш сквозной записи (write through)/ обратной записи (write back)
- механизм синхронизации кэшей в многопроцессорных системах (bus snooping)
- 32 байт буфер предвыборки
- встроенное FPU (без конвейера)
- 5-ти стадийный конвейер, использующий таблицу регистров (register scoreboard)

Pentium (1993, 3100000 транзисторов, частота шины 50/60/66MHz, 100 specint92, 80 specfp92)

- внутренняя частота процессора равна либо частоте шины (60/66MHz), либо 1.5 частоты шины (75/90/100MHz), удвоенной частоте шины (120/133MHz), 2.5 частоты шины (150/166MHz), либо утроенной частоте шины (180/200MHz)
- 64-бит внешняя шина данных, но 32-битная внутренняя архитектура
- суперскалярная архитектура: 2 IU
- аппаратное декодирование
- устройство предсказания переходов (BU)
- 5-ти стадийный конвейер в IU, 8-ми стадийный конвейер в FPU
- Гарвардская архитектура: 8Кб кэш инструкций и 8Кб кэш данных
- механизм синхронизации кэшей MESI (Modified, Exclusive, Shared, Invalid), используемый в PowerPC
- гибридная CISC/RISC архитектура

Pentium Pro (1996, 5500000 транзисторов, 150MHz, 366 specint92, 283 specfp92)

- суперскалярный процессор: до 5 инструкций за цикл
- 14-ти стадийные конвейеры (2 для IU и 1 для FPU)
- Гарвардская архитектура: 16Кб кэш инструкций и 8Кб кэш данных
- архитектура DIB (Dual Independent Bus): одна шина связывает процессор с кэш памятью второго уровня, а вторая - с ОЗУ
- 256(512)Кб встроенный кэш второго уровня; расположен на отдельном кристалле, но в том же корпусе, связан с процессором 64-битной шиной, работающей на частоте процессора
- исполнение инструкций не по порядку

- спекулятивное исполнение: выполнение инструкций за точкой ветвления
- переименование регистров (register renaming)

`Pentium-II (1997, 7500000 транзисторов, 233/266/300MHz)`

- Pentium Pro ядро с архитектурой DIB
- 512Кб встроенный кэш второго уровня; расположен на отдельном кристалле, связан с процессором 64-битной шиной, работающей на половине частоты процессора

`P7 (Intel и HP, 1998)`

- 64-бит архитектура
- RISC процессор с набором команд PA-RISC, способен выполнять инструкции i8086

3.3 Программная модель семейства Intel 80x86

В этом разделе мы рассмотрим процессоры семейства Intel 80x86 с точки зрения программиста (или компилятора).

3.3.1 Набор регистров процессоров Intel 80x86

Прикладной программе доступны 8 регистров общего назначения и несколько служебных регистров.

`Регистры eax, ebx, ecx, edx`

Могут содержать целочисленные данные следующих типов

1. Бит (только инструкции, работающие с битовыми полями)
2. Двоично-закодированные десятичные числа (BCD); байт содержит одну цифру, существуют инструкции, работающие с двумя цифрами в одном байте
3. Байт (8 бит); при записи в регистр старшая часть не используется и не изменяется; возможен доступ к битам 0...7 и битам 8...15 соответственно по именам `al`, `bl`, `cl`, `dl` и `ah`, `bh`, `ch`, `dh`
4. Слово (16 бит); при записи в регистр старшая часть не используется и не изменяется; обращение осуществляется по имени `ax`, `bx`, `cx`, `dx` соответственно
5. Длинное слово (32 бит)

`Регистры esi, edi, ebp, esp`

Могут содержать целочисленные данные следующих типов

1. Бит (только инструкции, работающие с битовыми полями)
2. Двоично-закодированные десятичные числа (BCD); байт содержит одну цифру, существуют инструкции, работающие с двумя цифрами в одном байте

3. Слово (16 бит); при записи в регистр старшая часть не используется и не изменяется; обращение осуществляется по имени `si`, `di`, `bp`, `sp` соответственно
4. Длинное слово (32 бит)

`Регистр кода условия ccr (condition code register)`
: является частью регистра флагов (EFLAGS), который не доступен пользовательской программе как регистр. Коды условия:

  `OF - overflow flag`
  : устанавливается в 1, если результат не входит в диапазон представимых значений, иначе - 0

  `SF - sign flag`
  : устанавливается в 1, если результат меньше 0, иначе - 0

  `ZF - zero flag`
  : устанавливается в 1, если результат равен 0, иначе - 0

  `CF - carry flag`
  : устанавливается в 1, если в результате произошел перенос в самом старшем разряде, иначе - 0

  `AF - auxiliary carry flag`
  : устанавливается в 1, если в результате произошел перенос в самом 3-м разряде, иначе - 0 (используется для работы с BCD данными)

  `PF - parity flag`
  : устанавливается в 1, если в младшем байте результата четное число единичных битов, иначе - 0

  `DF - direction flaf`
  : используется в строковых командах, не используется в командах условного перехода

  Коды условия устанавливаются по результату арифметических операций или специальными инструкциями и используются в командах условного перехода.

Следующие регистры доступны пользовательской программе, но не используются в прикладных программах для UNIX систем (поскольку они работают в плоской модели памяти):

`Сегментные регистры gs, fs, es, ds, cs, ss`
: содержат 16-бит селекторы сегментов, неявно участвуют при формировании адреса

Следующие специальные регистры недоступны пользовательской программе и их набор различен у разных моделей (ниже описаны регистры для i80486):

`Регистр EFLAGS`
: регистр флагов, содержит CCR

Регистр EIP (Instruction Pointer)
: указатель на следующую исполняемую инструкцию

Регистры управления памятью:

GDTR (Global Descriptor Table Register)
: Содержит 32-битный адрес и 16-битный предел глобальной таблицы дескрипторов (GDT)

LDTR (Local Descriptor Table Register)
: Содержит 32-битный адрес и 16-битный предел локальной таблицы дескрипторов (LDT)

IDTR (Interrupt Descriptor Table Register)
: Содержит 32-битный адрес и 16-битный предел таблицы дескрипторов прерываний (IDT)

TR (Task Register)
: Содержит 32-битный адрес, 16-битный предел, атрибуты дескриптора и 16-битный селектор сегмента исполняемой задачи, адресует дескриптор сегмента состояния задачи (TSS) в глобальной таблице дескрипторов (GDT)

Регистры управления CR0, CR1, CR2, CR3
: управляют состоянием процессора (например, включить/выключить кэш, кэш прямой/обратной записи, включить/выключить страничную трансляцию адреса и т.д.)

Отладочные регистры DR0 - DR7
: позволяют устанавливать точки прерывания

Проверочные регистры TR3 - TR7
: не являются частью архитектуры и зависят от реализации, служат для проверки TLB (translation lookaside buffer) кэша

3.3.2 Режимы адресации памяти процессоров Intel 80x86

Все режимы адресации можно записать одной формулой: адрес ячейки памяти есть

```
base + index*scale + displacement
```

где

base базовый регистр: **eax**, **ebx**, **ecx**, **edx**, **esi**, **edi**, **ebp**, **esp**

index индексный регистр: **eax**, **ebx**, **ecx**, **edx**, **esi**, **edi**, **ebp**

scale целая константа 1, 2, 4, 8

displacement
: смещение 8 или 32 бит

Любой из элементов адреса может отсутствовать (с одним исключением: если отсутствует **index**, то должен отсутствовать и **scale**). Ассемблерный синтаксис для адреса (различается в разных системах):

```
displacement(base,index,scale)
```

3.3.3 Основные инструкции процессоров Intel 80x86

Инструкции процессоров Intel 80x86 имеют от нуля до трех операндов (явных или неявных). Большинство инструкций (пересылки, арифметические, логические) имеют два операнда, один из которых не изменяется в операции (источник), а другой является результатом операции (приемник). Фирма Intel использует синтаксис, в котором приемник является левым операндом, а источник - правым. Но большинство ассемблеров в UNIX системах использует синтаксис ala Motorola 68xxx: левый операнд - источник, правый - приемник. В дальнейшем мы будем придерживаться этого синтаксиса.

У большинства двухоперандных инструкций один из операндов (источник или приемник) может быть регистром или ячейкой памяти, другой тогда может быть регистром или непосредственным значением. Это позволяет разделить двухоперандные инструкции на следующие группы:

- регистр - регистр
- регистр - память
- память - регистр
- непосредственное значение - регистр
- непосредственное значение - память

Существуют инструкции, (неявно) осуществляющие операции типа "память - память". Это строковые инструкции и операции со стеком.

Инструкции, неявно использующие жестко закрепленные регистры:

- умножение и деление с двойной точностью
- ввод/вывод
- работа со строками
- циклы
- сдвиги
- операции со стеком (включая вызовы функций)
- инструкция трансляции

Рассмотрим основные группы инструкций процессоров Intel 80x86. Ниже мы будем обозначать `Rn` - регистр, `<mem>` - ячейка памяти с адресом `<ea>`, `<ea>` - сам этот адрес, `<imm>` - непосредственное значение. Мы будем рассматривать только 32-битный режим работы процессора. В этом режиме большинство инструкций имеют операнды размером 8 или 32 бит; использовать операнды размером 16 бит можно при использовании специального префикса инструкции. Однако, изменить размер смещения в адресе (8 или 32 бит) таким способом невозможно.

3.3.3.1 Инструкции пересылки данных процессоров Intel 80x86

`CDQ`

- Синтаксис операндов: нет
- Размер операндов: 32 `->` 64

- Операция: `eax -> edx:eax` со знаковым расширением

`CWDE`

- Синтаксис операндов: нет
- Размер операндов: 16 `->` 32
- Операция: `ax -> eax` со знаковым расширением

`LEA`

- Синтаксис операндов: `<ea>, Rn`
- Размер операндов: 32
- Операция: `<ea> -> Rn`

`MOV`

- Синтаксис операндов: `Rn/<imm>, Rm/<mem>`; `<mem>, Rn`
- Размер операндов: 8, 32
- Операция: источник `->` приемник

`MOVS`

- Синтаксис операндов: нет
- Размер операндов: 8, 32
- Операция: источник `->` приемник; источник - ячейка памяти с адресом `esi`, приемник - ячейка памяти с адресом `edi`; после операции регистры `esi` и `edi` увеличиваются/уменьшаются на размер операнда (в зависимости от того, установлен ли флаг `DF`); при задании префикса `REP` эта операция будет выполнена `ecx` раз

`MOVSX`

- Синтаксис операндов: `Rn/<mem>, Rm`
- Размер операндов: 8, 16 `->` 32
- Операция: источник `->` приемник со знаковым расширением

`MOVZX`

- Синтаксис операндов: `Rn/<mem>, Rm`
- Размер операндов: 8, 16 `->` 32
- Операция: источник `->` приемник с расширением нулем

`POP`

- Синтаксис операндов: `Rn/<mem>`
- Размер операндов: 32
- Операция: `esp@ -> Rn/<mem>`, `esp+4 -> esp`

`POPA`

- Синтаксис операндов: нет
- Размер операндов: 32

- Операция: POP edi, POP esi, POP ebp, esp+4 -> esp, POP ebx, POP edx, POP ecx, POP eax

POPF

- Синтаксис операндов: нет
- Размер операндов: 32
- Операция: esp@ -> EFLAGS, esp+4 -> esp

PUSH

- Синтаксис операндов: Rn/<mem>/<imm>
- Размер операндов: 32
- Операция: esp-4 -> esp, Rn/<mem>/<imm> -> esp@

PUSHA

- Синтаксис операндов: нет
- Размер операндов: 32
- Операция: PUSH eax, PUSH ecx, PUSH edx, PUSH ebx, PUSH esp (исходный), PUSH ebp, PUSH esi, PUSH edi

PUSHF

- Синтаксис операндов: нет
- Размер операндов: 32
- Операция: esp-4 -> esp, EFLAGS -> esp@

XCHG

- Синтаксис операндов: Rn/<mem>,Rm; Rn,Rm/<mem>
- Размер операндов: 8, 32
- Операция: обменять значения операндов

3.3.3.2 Целочисленные арифметические инструкции процессоров Intel 80x86

ADD

- Синтаксис операндов: Rn/<inm>, Rm/<mem>; <mem>, Rn
- Размер операндов: 8, 32
- Операция: источник + приемник -> приемник

ADC

- Синтаксис операндов: Rn/<inm>, Rm/<mem>; <mem>, Rn
- Размер операндов: 8, 32
- Операция: источник + приемник + CF -> приемник

CMP

- Синтаксис операндов: Rn/<inm>, Rm/<mem>; <mem>, Rn

- Размер операндов: 8, 32
- Операция: приемник - источник

`CMPS`

- Синтаксис операндов: нет
- Размер операндов: 8, 32
- Операция: источник - приемник; источник - ячейка памяти с адресом `esi`, приемник - ячейка памяти с адресом `edi`; после операции регистры `esi` и `edi` увеличиваются/уменьшаются на размер операнда (в зависимости от того, установлен ли флаг `DF`); при задании префикса `REPE` эта операция будет выполняться пока операнды равны, но не более `ecx` раз; при задании префикса `REPNE` эта операция будет выполняться пока операнды не равны, но не более `ecx` раз

`DEC`

- Синтаксис операндов: `Rn/<mem>`
- Размер операндов: 8, 32
- Операция: приемник - 1 `->` приемник

`DIV`

- Синтаксис операндов: `Rn/<mem>, al`; `Rn/<mem>, eax`
- Размер операндов: 8, 32
- Операция: приемник / источник `->` приемник (беззнаково); источник = `ax` / `edx:eax`; результат:остаток = `al:ah` / `eax:edx`

`IDIV`

- Синтаксис операндов: `Rn/<mem>, al`; `Rn/<mem>, eax`
- Размер операндов: 8, 32
- Операция: приемник / источник `->` приемник (знаково); результат:остаток = `al:ah` / `eax:edx`

`IMUL`

- Синтаксис операндов: `Rn/<mem>`; `Rn/<mem>/<imm>, Rm`; `Rn/<mem>, <imm>, Rm`
- Размер операндов: 8, 32
- Операция: умножение со знаком: `al/eax * источник -> ax/edx:eax`; приемник `*` источник `->` приемник; `Rn/<mem> * <imm> -> Rm`

`INC`

- Синтаксис операндов: `Rn/<mem>`
- Размер операндов: 8, 32
- Операция: приемник + 1 `->` приемник

`MUL`

- Синтаксис операндов: `Rn/<mem>, al`; `Rn/<mem>, eax`
- Размер операндов: 8, 32
- Операция: умножение без знака: `al/eax *` источник `-> ax/edx:eax`

`NEG`

- Синтаксис операндов: `Rn/<mem>`
- Размер операндов: 8, 32
- Операция: 0 - приемник `->` приемник

`SUB`

- Синтаксис операндов: `Rn/<inm>, Rm/<mem>`; `<mem>, Rn`
- Размер операндов: 8, 32
- Операция: приемник - источник `->` приемник

`SBB`

- Синтаксис операндов: `Rn/<inm>, Rm/<mem>`; `<mem>, Rn`
- Размер операндов: 8, 32
- Операция: приемник - источник - CF `->` приемник

3.3.3.3 Логические инструкции процессоров Intel 80x86

`AND`

- Синтаксис операндов: `Rn/<inm>, Rm/<mem>`; `<mem>, Rn`
- Размер операндов: 8, 32
- Операция: источник `AND` приемник `->` приемник

`NOT`

- Синтаксис операндов: `Rn/<mem>`
- Размер операндов: 8, 32
- Операция: `NOT` приемник `->` приемник

`OR`

- Синтаксис операндов: `Rn/<inm>, Rm/<mem>`; `<mem>, Rn`
- Размер операндов: 8, 32
- Операция: источник `OR` приемник `->` приемник

`XOR`

- Синтаксис операндов: `Rn/<inm>, Rm/<mem>`; `<mem>, Rn`
- Размер операндов: 8, 32
- Операция: источник `EOR` приемник `->` приемник

`TEST`

- Синтаксис операндов: `Rn/<inm>, Rm/<mem>`; `<mem>, Rn`
- Размер операндов: 8, 32
- Операция: источник `AND` приемник `->` коды условия

3.3.3.4 Инструкции сдвига процессоров Intel 80x86

`SAL`

- Синтаксис операндов: `<imm>/cl, Rn/<mem>`
- Размер операндов: 8, 32
- Операция: сдвинуть приемник влево на `<imm>/cl` бит, вдвигаются нули

`SAR`

- Синтаксис операндов: `<imm>/cl, Rn/<mem>`
- Размер операндов: 8, 32
- Операция: сдвинуть приемник вправо на `<imm>/cl` бит, вдвигается знаковый бит

`SHL`

- Синтаксис операндов: `<imm>/cl, Rn/<mem>`
- Размер операндов: 8, 32
- Операция: совпадает с `SAL`

`SHR`

- Синтаксис операндов: `<imm>/cl, Rn/<mem>`
- Размер операндов: 8, 32
- Операция: сдвинуть приемник вправо на `<imm>/cl` бит, вдвигаются нули

`SHLD`

- Синтаксис операндов: `<imm>/cl, Rn/<mem>, Rm`
- Размер операндов: 32:32
- Операция: сдвинуть приемник `Rn/<mem>, Rm` влево на `<imm>/cl` бит, вдвигаются нули

`SHRD`

- Синтаксис операндов: `<imm>/cl, Rn/<mem>, Rm`
- Размер операндов: 32:32
- Операция: сдвинуть приемник `Rn/<mem>, Rm` вправо на `<imm>/cl` бит, вдвигаются нули

`RCL`

- Синтаксис операндов: `<imm>/cl, Rn/<mem>`
- Размер операндов: 8, 32
- Операция: циклически сдвинуть приемник `(CF, Rn/<mem>)` влево на `<imm>/cl` бит

`RCR`

- Синтаксис операндов: `<imm>/cl, Rn/<mem>`

- Размер операндов: 8, 32
- Операция: циклически сдвинуть приемник (CF, Rn/<mem>) вправо на <imm>/cl бит

ROL

- Синтаксис операндов: <imm>/cl, Rn/<mem>
- Размер операндов: 8, 32
- Операция: циклически сдвинуть приемник влево на <imm>/cl бит

ROR

- Синтаксис операндов: <imm>/cl, Rn/<mem>
- Размер операндов: 8, 32
- Операция: циклически сдвинуть приемник вправо на <imm>/cl бит

3.3.3.5 Инструкции управления процессоров Intel 80x86

Условные переходы:

Jcc

- Синтаксис операндов: <label>
- Размер операндов: 8, 32
- Операция: если условие истинно, то PC + offset(<label>) -> PC

Коды условия cc здесь, беззнаковые:

- A/NBE - above/not below nor equal
- AE/NB - above or equal/not below
- B/NAE - below/not above nor equal
- BE/NA - below or equal/not above
- C - carry
- E/Z - equal/zero
- NC - not carry
- NE/NZ - not equal/not zero
- NP/PO - not parity/parity odd
- P/PE - parity/parity even

знаковые:

- G/NLE - greater/not less nor equal
- GE/NL - greater or equal/not less
- L/NGE - less/not greater nor equal
- LE/NG - less or equal/not greater
- NO - not overflow
- NS - not sign (non-negative)
- O - overflow

– S - sign (negative)

Безусловные переходы:

`CALL`

– Синтаксис операндов: `<label>`; `Rn/<mem>`
– Размер операндов: нет (всегда 32)
– Операция: `esp-4 -> esp, PC -> (esp)@, PC + offset(<label>) -> PC`; `esp-4 -> esp, PC -> (esp)@, Rn/<mem> -> PC`

`JMP`

– Синтаксис операндов: `<label>`; `Rn/<mem>`
– Размер операндов: 8, 32; 32
– Операция: `PC + offset(<label>) -> PC`; `Rn/<mem> -> PC`

`NOP`

– Синтаксис операндов: нет
– Размер операндов: нет
– Операция: `PC+1 -> PC`

Возврат из процедуры

`RET`

– Синтаксис операндов: нет; `<imm>`
– Размер операндов: нет; 16
– Операция: `(esp)@ -> PC, esp+4 -> esp; (esp)@ -> PC, esp+4+<imm> -> esp`

3.3.3.6 Мультипроцессорные инструкции процессоров Intel 80x86

У многих инструкций можно поставить префикс `LOCK`, вызывающий блокировку шины на время операции, т.е. операция становится неделимой. По умолчанию этот префикс имеет только описанная выше инструкция `XCHG`. Однако, для работы с семафорами более удобна следующая инструкция (которую для этих целей надо использовать с префиксом `LOCK`)

`CMPXCHG`

– Синтаксис операндов: `Rn, Rm/<mem>`
– Размер операндов: 8, 32
– Операция: `al/eax - Rm/<mem> -> CC`, если `ZF`, то `Rn -> Rm/<mem>`, иначе `Rm/<mem> -> al/eax`

3.4 Примеры кода для семейства Intel 80x86

Мы используем ассемблерный синтаксис и соглашения о вызовах операционной системы Linux. Основные соглашения:

`Stack Pointer (SP)` **(диктуется системой команд)**
: = регистр `esp`

`Frame Pointer (FP)`
: = регистр `ebp`

`Адрес возврата из функции` **(диктуется системой команд)**
: = `mem(SP)` (т.е. ячейка памяти с адресом `SP`), здесь `SP` - в точке входа в функцию

`Возвращаемое значение функции`
: = регистр `eax`

`Аргументы функции`
: = `mem(SP+4)`, `mem(SP+8)`, ..., здесь `SP` - в точке входа в функцию

`Регистры, не сохраняемые при вызове функции`
: = `eax`, `ecx`, `edx`

`Регистры, сохраняемые при вызове функции`
: = `ebx`, `esi`, `edi`, `ebp`, `esp`

Мы используем те же примеры исходного кода, что и для процессоров Motorola 68xxx.

1.
```
f:
        pushl %ebp
        movl %esp,%ebp
        xorl %eax,%eax
        movl %ebp,%esp
        popl %ebp
        ret
```

2.
```
f:
        pushl %ebp
        movl %esp,%ebp
        movl a,%eax
        movl %ebp,%esp
        popl %ebp
        ret
```

3.
```
f:
        pushl %ebp
        movl %esp,%ebp
        movl $a,%eax
        movl %ebp,%esp
```

```
        popl %ebp
        ret
```

4.

```
f:
        pushl %ebp
        movl %esp,%ebp
        movl 8(%ebp),%eax
        movl %ebp,%esp
        popl %ebp
        ret
g:
        pushl %ebp
        movl %esp,%ebp
        pushl $1
        call f
        movl %ebp,%esp
        popl %ebp
        ret
```

5.

```
f:
        pushl %ebp
        movl %esp,%ebp
        movl 8(%ebp),%eax
        addl 12(%ebp),%eax
        movl %ebp,%esp
        popl %ebp
        ret
g:
        pushl %ebp
        movl %esp,%ebp
        movl a,%eax
        pushl %eax
        pushl $1
        call f
        movl %ebp,%esp
        popl %ebp
        ret
```

6.

```
f:
        pushl %ebp
        movl %esp,%ebp
        pushl %ebx
        movl 8(%ebp),%ebx
        movl 12(%ebp),%ecx
        xorl %edx,%edx
        xorl %eax,%eax
        cmpl %ecx,%eax
```

```
        jge .L3
        .align 4
.L5:
        addl (%ebx,%edx,4),%eax
        incl %edx
        cmpl %ecx,%edx
        jl .L5
.L3:
        movl -4(%ebp),%ebx
        movl %ebp,%esp
        popl %ebp
        ret
g:
        pushl %ebp
        movl %esp,%ebp
        movl n,%eax
        pushl %eax
        pushl $a
        call f
        movl %ebp,%esp
        popl %ebp
        ret
```

7.

```
f:
        pushl %ebp
        movl %esp,%ebp
        pushl %edi
        pushl %esi
        pushl %ebx
        movl 8(%ebp),%edi
        movl 12(%ebp),%esi
        movl 16(%ebp),%ebx
        xorl %edx,%edx
        xorl %ecx,%ecx
        cmpl %ebx,%ecx
        jge .L3
        .align 4
.L5:
        movl (%edi,%edx,4),%eax
        imull (%esi,%edx,4),%eax
        addl %eax,%ecx
        incl %edx
        cmpl %ebx,%edx
        jl .L5
.L3:
        movl %ecx,%eax
        leal -12(%ebp),%esp
        popl %ebx
```

```
        popl %esi
        popl %edi
        movl %ebp,%esp
        popl %ebp
        ret
```

8.

```
f:
        pushl %ebp
        movl %esp,%ebp
        movl 8(%ebp),%eax
        movl 12(%ebp),%edx
        movl 16(%ebp),%ecx
        cmpl %edx,%eax
        jl .L2
        cmpl %ecx,%eax
        jge .L3
.L2:
        movl %ecx,%eax
        cmpl %eax,%edx
        jl .L3
        movl %edx,%eax
.L3:
        movl %ebp,%esp
        popl %ebp
        ret
```

4 Архитектура процессоров PowerPC

4.1 Общий обзор процессоров PowerPC

Архитектура PowerPC (Performance Optimized With Enhanced Risc Personal Computer) была разработана совместно IBM, Motorola и Apple. Эта архитектура определяет три архитектурных уровня:

- `UISA` (User Instruction Set Architecture) определяет уровень архитектуры, которому должно удовлетворять пользовательское программное обеспечение. UISA задает программную модель и модель памяти для пользовательских программ. Именно, UISA определяет доступный прикладной программе набор инструкций, набор регистров, типы данных, соглашения о хранении чисел с плавающей точкой в памяти, модель исключений, видимую прикладной программой.
- `VEA` (Virtual Environment Architecture) определяет дополнительный уровень архитектуры, которому должно удовлетворять пользовательское программное обеспечение, выходящее за рамки обычных требований прикладных программ. VEA задает модель памяти для окружений, в которых множество устройств могут получать доступ к памяти, определяет модель кэша и инструкции управления кэшем.

- `OEA` (Operating Environment Architecture) определяет уровень архитектуры, которому должно удовлетворять супервизорское программное обеспечение (операционные системы). OEA определяет модель управления памятью, регистры уровня суперпользователя, требования к синхронизации процессов, модель исключений.

Эти спецификации позволяют разрабатывать новые процессоры семейства, сохраняя программную совместимость с существующими и будущими процессорами PowerPC. Впервые в истории разработки процессоров важнейшие усилия по спецификации были предприняты до появления первого процессора.

В дополнение к этим архитектурным определениям три производителя (IBM, Motorola, Apple) разработали `эталонную платформу` для разработки плат на основе PowerPC. `CHRP` (Common Hardware Reference Platform) является открытой спецификацией для разработки компьютерных систем на основе PowerPC. Отметим два важнейших аспекта этой спецификации. Во-первых, спецификация описывает устройства, интерфейсы и форматы данных, требуемые для разработки и построения законченной компьютерной системы. Она описывает методы абстрагирования аппаратных деталей от операционной системы. Более того, CHRP предписывает использовать вторую PCI шину, так, что шина PowerPC разделяется только L2 кэшем, контроллером памяти и мостом PCI. Это позволяет разработчикам использовать новые процессоры PowerPC без изменений в остальной части платы. Во-вторых, спецификация описывает эталонную реализацию операционной системы, согласованную с известными операционными системами для PowerPC: AIX, Windows NT.

4.2 Основные члены семейства PowerPC

Каждый процессор в приведенных ниже технических данных обладает всеми возможностями предыдущих процессоров для обеспечения совместимости.

PowerPC 603 (1993, 1600000 транзисторов)

- 8Кб кэш инструкций + 8Кб кэш данных (32-бит Гарвардская архитектура)
- 5 независимых исполняющих устройств: 1 IU + 1 FPU + 1 BU + 1 load/store unit + 1 system register unit
- FPU конвейеризировано, так, что инструкции сложения и умножения одинарной точности могут обрабатываться каждый цикл
- 4-х стадийный конвейер
- MMU, выполняющее сегментную и страничную трансляцию адреса из 52-битного логического адреса в 32-битный физический адрес
- два 64-входовых TLB
- потребление 3Вт на частоте 80MHz, четыре программно контролируемых режима экономии энергии

PowerPC 603e (1993, 1600000 транзисторов)

Вариант PowerPC 603 для настольных систем, частота до 240MHz

PowerPC 604 (1994, 3600000 транзисторов)

- суперскалярный процессор (4 инструкции за цикл)
- 16Кб кэш инструкций + 16Кб кэш данных (32-бит Гарвардская архитектура)
- 6 независимых исполняющих устройств: 3 IU + 1 FPU + 1 BU + 1 load/store unit
- 6-ти стадийный конвейер
- два 128-входовых TLB
- возможность последовательного выполнения инструкций
- переименование регистров (register renaming)
- динамическое предсказание переходов
- поддержка многопроцессорности посредством специального протокола синхронизации кэшей
- потребление 10Вт на частоте 100MHz, один программно контролируемый режим экономии энергии

PowerPC 604e (1994, 3600000 транзисторов)

Вариант PowerPC 604 для настольных систем, частота до 300MHz

PowerPC 620 (1995, 7000000 транзисторов, на частоте 133MHz 225 specint 92, 300 specfp 92)

- 128-бит шина данных, 40-бит шина адреса, 64-бит регистры
- суперскалярный процессор (6 инструкций за цикл)
- 32Кб кэш инструкций + 32Кб кэш данных (64-бит Гарвардская архитектура)
- 6 независимых исполняющих устройств: 2 IU + 1 FPU + 1 BU + 1 load/store unit + 1 complex unit

- 5-ти стадийный конвейер
- 2-х уровневое MMU, первичное MMU имеет 64-входовый TLB, вторичное MMU имеет 128-входовый TLB
- для ускорения переходов count register (`ctr`) имеет теневой регистр, в который он переименовывается во время перехода
- интегрированный контроллер L2 кэша
- потребление 30Вт на частоте 100MHz, один программно контролируемый режим экономии энергии

`PowerPC G3 (1997, не более 30000000 транзисторов, частота 300MHz)`
: развитие PowerPC 620, интегрированный L2 кэш

`PowerPC G4 (1998, не более 50000000 транзисторов, частота 400MHz)`
: развитие PowerPC G3

`IBM PowerPC 403 (1994)`

- PowerPC микроконтроллер
- 2Кб кэш инструкций + 1Кб кэш данных (32-бит Гарвардская архитектура)
- периферийные интерфейсные устройства: интерфейс шины, DMA контроллер, контроллер прерываний (на 6 запросов), последовательный порт
- до восьми интерфейсов банков памяти и устройств ввода/вывода
- 4 таймера
- низкое потребление энергии

`Motorola MPC500 (1994, 40 MIPS, 25MHz)`

- PowerPC микроконтроллер
- 4Кб кэш инструкций
- 4Кб SRAM
- 4 независимых исполняющих устройств: 1 IU + 1 FPU + 1 BU + 1 load/store unit
- контроллер прерываний на 32 запроса (время задержки 1mks на частоте 25MHz)
- контроллер памяти, способный управлять 12 микросхемами памяти
- встроенный автотест
- очень низкое потребление энергии (530mW)

`Motorola PowerQUICC (1996, 52 MIPS, 40MHz)`

- улучшенная версия QUICC (M68360)
- ядро CPU32+ заменено на ядро MPC500
- 4 высокоскоростных последовательных коммуникационных канала контролируются выделенным RISC коммуникационным процессором, работающим независимо от основного процессора
- PowerQUICC по сравнению с QUICC имеет контроллер PCMCIA 2.01 и аналог процессора DSP

4.3 Программная модель семейства PowerPC

В этом разделе мы рассмотрим процессоры семейства PowerPC с точки зрения программиста (или компилятора).

4.3.1 Набор регистров процессоров PowerPC

Прикладной программе доступны

`32 целочисленных регистра общего назначения (РОН) r0 - r31`
: содержат слово (32/64 бит)

`32 регистра с плавающей точкой f0 - f31`
: содержат значение с плавающей точкой (64 бит)

`cr (condition register)`
: это 32-битный регистр, разделенный на восемь 4-х битных полей `cr0-cr7`. Поля регистра `cr` могут быть установлены одним из следующих способов.

- Указанное поле регистра `cr` может быть установлено с помощью инструкции пересылки `mtcrf` в `cr` из РОН.
- Указанное поле регистра `cr` может быть установлено с помощью инструкции пересылки `mcrf` в `cr` из другого поля `cr`.
- Указанное поле регистра `xer` может быть скопировано в регистр `cr` с помощью инструкции `mcrxr`.
- Указанное поле регистра `fpscr` может быть скопировано в регистр `cr` с помощью инструкции `mcrfs`.
- Поля регистра `cr` могут быть изменены с помощью логических операций, определенных над полями `cr`.
- `cr0` может быть неявным результатом целочисленной инструкции. Все целочисленные инструкции имеют бит `Rc`; если его установить, то биты в `cr0` будут установлены сравнением результата инструкции с нулем:
  - `cr0` бит 0 - Negative (LT)
  - `cr0` бит 1 - Positive (GT)
  - `cr0` бит 2 - Zero (EQ)
  - `cr0` бит 3 - Summary overflow (SO) (копируется из регистра `xer`)
- `cr1` может быть неявным результатом инструкции с плавающей точкой и указывать на статус исключения с плавающей точкой. Все инструкции с плавающей точкой имеют бит `Rc`; если его установить, то биты в `cr1` будут установлены копированием соответствующих битов из `fpscr`:
  - `cr1` бит 0 - Floating-point exception (FX)
  - `cr1` бит 1 - Floating-point enabled exception (FEX)
  - `cr1` бит 2 - Floating-point invalid exception (VX)
  - `cr1` бит 3 - Floating-point overflow exception (OX)

- Указанное поле регистра `cr` может быть результатом целочисленной или вещественной инструкции сравнения:
  - `crN` бит 0 - Less than (LT)
  - `crN` бит 1 - Greater than (GT)
  - `crN` бит 2 - Equal (EQ)
  - `crN` бит 3 - для целочисленных сравнений: Summary overflow (SO) (копируется из регистра `xer`); для сравнений с плавающей точкой: один или оба операнда есть Not a Number (NaN)

`fpscr (Floating Point Status and Control Register)`
: это 32-битный регистр, содержащий все биты сигналов исключений для операций с плавающей точкой, биты суммарных исключений, биты разрешения исключений, биты управления округлением, необходимые для удовлетворения стандарту IEEE 754.

`xer register`
: это 32-битный регистр, содержащий флаги переполнения и переносов для целочисленных операций:
  - `xer` бит 0 - Summary Overflow (SO)
  - `xer` бит 1 - Overflow (OV)
  - `xer` бит 2 - Carry (CA)
  - `xer` биты 3-24 - Зарезервированы
  - `xer` биты 25-31 - Содержит число байтов, которые нужно передать в инструкциях Load String Word Indexed (`lswx`) или Store String Word Indexed (`stswx`)

`lr (link register)`
: это 32/64-битный регистр, содержащий адрес перехода для инструкций Branch Conditional to Link Register (`bclr`*x*) и Branch and Link (`bl`).

`ctr (count register)`
: это 32/64-битный регистр, содержащий счетчик цикла, который может быть декрементирован в течение выполнения надлежащим образом закодированных инструкций перехода. Регистр `ctr` может также содержать адрес перехода для инструкции Branch Conditional to Count Register (`bcctr`*x*).

4.3.2 Режимы адресации памяти процессоров PowerPC

Поддерживаются следующие два режима адресации

- `<ea> = (rA|0)+offset` (включая `offset`=0)
- `<ea> = (rA|0)+rB`

где обозначено

– `(rA|0)` - РОН `r1`...`r31`, если `rA` не равно `r0`, иначе 0
– `offset` - 16-бит смещение (знаково расширяемое)
– `rB` - РОН `r0`...`r31`

4.3.3 Основные инструкции процессоров PowerPC

Все инструкции (за исключением load/store) имеют операнды в регистрах и потому размер всех операндов равен размеру слова (32/64 бит). Мы будем обозначать размер регистра через REG_SIZE (32 или 64 бит).

4.3.3.1 Инструкции пересылки данных процессоров PowerPC

В load инструкциях можно установить код условия `cr0` по результату.

`LBZ`
`LHZ`
`LWZ`

- Синтаксис операндов: `rD, d(rA)`
- Размер операндов: соответственно 8, 16, 32 `->` REG_SIZE
- Операция: `mem((rA|0)+d) -> rD` с расширением нулем

`LBZX`
`LHZX`
`LWZX`

- Синтаксис операндов: `rD, rA, rB`
- Размер операндов: соответственно 8, 16, 32 `->` REG_SIZE
- Операция: `mem((rA|0)+rB) -> rD` с расширением нулем

`LBZU`
`LHZU`
`LWZU`

- Синтаксис операндов: `rD, d(rA)`
- Размер операндов: соответственно 8, 16, 32 `->` REG_SIZE
- Операция: `mem((rA|0)+d) -> rD` с расширением нулем, `rA+d -> rA`

`LBZUX`
`LHZUX`
`LWZUX`

- Синтаксис операндов: `rD, rA, rB`
- Размер операндов: соответственно 8, 16, 32 `->` REG_SIZE
- Операция: `mem((rA|0)+rB) -> rD` с расширением нулем, `rA+rB -> rA`

`LHA`
`LWA`

- Синтаксис операндов: `rD, d(rA)`
- Размер операндов: соответственно 16, 32 `->` REG_SIZE
- Операция: `mem((rA|0)+d) -> rD` со знаковым расширением

`LHAX`
`LWAX`

- Синтаксис операндов: `rD, rA, rB`
- Размер операндов: соответственно 16, 32 `->` REG_SIZE
- Операция: `mem((rA|0)+rB) -> rD` со знаковым расширением

`LHAU`

- Синтаксис операндов: `rD, d(rA)`
- Размер операндов: 16 `->` REG_SIZE
- Операция: `mem((rA|0)+d) -> rD` со знаковым расширением, `rA+d -> rA`

`LHAUX`
`LWAUX`

- Синтаксис операндов: `rD, rA, rB`
- Размер операндов: соответственно 8, 16, 32 `->` REG_SIZE
- Операция: `mem((rA|0)+rB) -> rD` со знаковым расширением, `rA+rB -> rA`

`STB`
`STH`
`STW`

- Синтаксис операндов: `rS, d(rA)`
- Размер операндов: REG_SIZE `->` 8, 16, 32 соответственно
- Операция: `rS -> mem((rA|0)+d)`

`STBX`
`STHX`
`STWX`

- Синтаксис операндов: `rS, rA, rB`
- Размер операндов: REG_SIZE `->` 8, 16, 32 соответственно
- Операция: `rS -> mem((rA|0)+rB)`

`STBU`
`STHU`
`STWU`

- Синтаксис операндов: `rS, d(rA)`
- Размер операндов: REG_SIZE `->` 8, 16, 32 соответственно
- Операция: `rS -> mem((rA|0)+d)`, `rA+d -> rA`

`STBUX`
`STHUX`
`STWUX`

- Синтаксис операндов: `rS, rA, rB`
- Размер операндов: REG_SIZE `->` 8, 16, 32 соответственно
- Операция: `rS -> mem((rA|0)+rB)`, `rA+rB -> rA`

`STMW`

- Синтаксис операндов: `rS, d(rA)`
- Размер операндов: REG_SIZE `->` 32
- Операция: регистры с номерами от `S` до 31 `->` `mem((rA|0)+d)`

`LMW`

- Синтаксис операндов: `rD, d(rA)`
- Размер операндов: 32 `->` REG_SIZE
- Операция: `mem((rA|0)+d)` `->` регистры с номерами от `D` до 31

Инструкции пересылки, определенные с `cr` (condition register):

`MCRF`

- Синтаксис операндов: `crfD, crfS`
- Размер операндов: нет
- Операция: `(поле crfS)` `->` `(поле crfD)`

`MTCRF`

- Синтаксис операндов: `CRM, rS`
- Размер операндов: REG_SIZE, `CRM` - 8 бит
- Операция: строится `mask`, в которой 1 стоят в позициях тех 4-бит полей `cr`, которые указаны в `CRM`; `(rS & mask) | (cr & ~mask)` `->` `cr`

`MFCR`

- Синтаксис операндов: `rD`
- Размер операндов: REG_SIZE
- Операция: `cr` `->` `rD`

`MCRXR`

- Синтаксис операндов: `crfD`
- Размер операндов: нет
- Операция: `xer[0-3]` `->` `crfD`

Инструкции пересылки, определенные с регистрами специального назначения SPR (Special-Purpose Registers) (например, с `lr`, `ctr`, `xer` и т.д.):

`MTSPR`

- Синтаксис операндов: `SPR, rS`
- Размер операндов: REG_SIZE
- Операция: `rS` `->` `SPR`

`MFSPR`

- Синтаксис операндов: `rD, SPR`
- Размер операндов: REG_SIZE
- Операция: `SPR` `->` `rD`

4.3.3.2 Целочисленные арифметические инструкции процессоров PowerPC

В большинстве инструкций можно установить код условия `cr0` по результату.

`ADDI`

- Синтаксис операндов: `rD, rA, SIMM`
- Размер операндов: REG_SIZE
- Операция: `(rA|0)+SIMM -> rD`

`ADDIC`
`ADDIC.`

- Синтаксис операндов: `rD, rA, SIMM`
- Размер операндов: REG_SIZE
- Операция: `rA+SIMM -> rD` и установить по результату флаг переноса (`CA`) в регистре `xer`; если присутствует суффикс '`.`', то установить по результату поле `cr0` в регистре `cr`

`ADDIS`

- Синтаксис операндов: `rD, rA, SIMM`
- Размер операндов: REG_SIZE
- Операция: `(rA|0)+(SIMM<<16) -> rD`

`ADD`
`ADD.`
`ADDO`
`ADDO.`
`ADDC`
`ADDC.`
`ADDCO`
`ADDCO.`

- Синтаксис операндов: `rD, rA, rB`
- Размер операндов: REG_SIZE
- Операция: `rA+rB -> rD`; если присутствует суффикс '`.`', то установить по результату поле `cr0` в регистре `cr`; если присутствует суффикс '`C`', то установить по результату флаг переноса (`CA`) в регистре `xer`; если присутствует суффикс '`O`', то установить по результату флаги переполнения (`OV`, `SO`) в регистре `xer`

`ADDE`
`ADDE.`
`ADDEO`
`ADDEO.`

- Синтаксис операндов: `rD, rA, rB`
- Размер операндов: REG_SIZE

– Операция: `rA+rB+CA -> rD`; если присутствует суффикс '`.`', то установить по результату поле `cr0` в регистре `cr`; если присутствует суффикс '`O`', то установить по результату флаги переполнения (`OV`, `SO`) в регистре `xer`

`ADDZE`
`ADDZE.`
`ADDZEO`
`ADDZEO.`

– Синтаксис операндов: `rD, rA`
– Размер операндов: REG_SIZE
– Операция: `rA+CA -> rD`; если присутствует суффикс '`.`', то установить по результату поле `cr0` в регистре `cr`; если присутствует суффикс '`O`', то установить по результату флаги переполнения (`OV`, `SO`) в регистре `xer`

`ADDME`
`ADDME.`
`ADDMEO`
`ADDMEO.`

– Синтаксис операндов: `rD, rA`
– Размер операндов: REG_SIZE
– Операция: `rA+CA-1 -> rD`; если присутствует суффикс '`.`', то установить по результату поле `cr0` в регистре `cr`; если присутствует суффикс '`O`', то установить по результату флаги переполнения (`OV`, `SO`) в регистре `xer`

`SUBFIC`

– Синтаксис операндов: `rD, rA, SIMM`
– Размер операндов: REG_SIZE
– Операция: `SIMM-rA -> rD` (или `~(rA)+SIMM+1 -> rD`) и установить по результату флаг переноса (`CA`) в регистре `xer`

`SUBF`
`SUBF.`
`SUBFO`
`SUBFO.`
`SUBFC`
`SUBFC.`
`SUBFCO`
`SUBFCO.`

– Синтаксис операндов: `rD, rA, rB`
– Размер операндов: REG_SIZE
– Операция: `rB-rA -> rD` (или `~(rA)+rB+1 -> rD`); если присутствует суффикс '`.`', то установить по результату поле `cr0` в регистре `cr`; если

присутствует суффикс ‘C’, то установить по результату флаг переноса (CA) в регистре xer; если присутствует суффикс ‘O’, то установить по результату флаги переполнения (OV, SO) в регистре xer

SUBFE
SUBFE.
SUBFEO
SUBFEO.

- Синтаксис операндов: rD, rA, rB
- Размер операндов: REG_SIZE
- Операция: rB-rA+CA-1 -> rD (или ~(rA)+rB+CA -> rD); если присутствует суффикс ‘.’, то установить по результату поле cr0 в регистре cr; если присутствует суффикс ‘O’, то установить по результату флаги переполнения (OV, SO) в регистре xer

SUBFZE
SUBFZE.
SUBFZEO
SUBFZEO.

- Синтаксис операндов: rD, rA
- Размер операндов: REG_SIZE
- Операция: CA-1-rA -> rD (или ~(rA)+CA -> rD); если присутствует суффикс ‘.’, то установить по результату поле cr0 в регистре cr; если присутствует суффикс ‘O’, то установить по результату флаги переполнения (OV, SO) в регистре xer

SUBFME
SUBFME.
SUBFMEO
SUBFMEO.

- Синтаксис операндов: rD, rA
- Размер операндов: REG_SIZE
- Операция: CA-rA-2 -> rD (или ~(rA)+CA-1 -> rD); если присутствует суффикс ‘.’, то установить по результату поле cr0 в регистре cr; если присутствует суффикс ‘O’, то установить по результату флаги переполнения (OV, SO) в регистре xer

NEG
NEG.
NEGO
NEGO.

- Синтаксис операндов: rD, rA
- Размер операндов: REG_SIZE
- Операция: -rA -> rD; если присутствует суффикс ‘.’, то установить по результату поле cr0 в регистре cr; если присутствует суффикс ‘O’, то установить по результату флаги переполнения (OV, SO) в регистре xer

`MULLI`

- Синтаксис операндов: `rD, rA, SIMM`
- Размер операндов: REG_SIZE
- Операция: `rA*SIMM -> rD` (получить младших 32/64 бит результата)

`MULLW`
`MULLW.`
`MULLWO`
`MULLWO.`

- Синтаксис операндов: `rD, rA, rB`
- Размер операндов: REG_SIZE
- Операция: `rA*rB -> rD` (получить младших 32/64 бит результата); если присутствует суффикс '`.`', то установить по результату поле `cr0` в регистре `cr`; если присутствует суффикс '`O`', то установить по результату флаги переполнения (`OV`, `SO`) в регистре `xer`

`MULHW`
`MULHW.`
`MULHWU`
`MULHWU.`

- Синтаксис операндов: `rD, rA, rB`
- Размер операндов: REG_SIZE
- Операция: `rA*rB -> rD` (получить старших 32/64 бит результата); если присутствует суффикс '`.`', то установить по результату поле `cr0` в регистре `cr`; если присутствует суффикс '`U`', то выполнить беззнаковую операцию

`DIVW`
`DIVW.`
`DIVWO`
`DIVWO.`
`DIVWU`
`DIVWU.`
`DIVWUO`
`DIVWUO.`

- Синтаксис операндов: `rD, rA, rB`
- Размер операндов: REG_SIZE
- Операция: `rA/rB -> rD`; если присутствует суффикс '`.`', то установить по результату поле `cr0` в регистре `cr`; если присутствует суффикс '`O`', то установить по результату флаги переполнения (`OV`, `SO`) в регистре `xer`; если присутствует суффикс '`U`', то выполнить беззнаковую операцию

`CMP`
`CMPL`

- Синтаксис операндов: `crfD, rA, rB`
- Размер операндов: REG_SIZE
- Операция: содержимое `rA` сравнивается с содержимым `rB` и устанавливаются коды условия в поле `crfD` регистра `cr`; операнды трактуются как знаковые целые (без суффикса 'L') или беззнаковые целые (с суффиксом 'L')

`CMPI`

- Синтаксис операндов: `crfD, rA, SIMM`
- Размер операндов: REG_SIZE
- Операция: содержимое `rA` сравнивается с содержимым `rB` и устанавливаются коды условия в поле `crfD` регистра `cr`; операнды трактуются как знаковые целые

`CMPLI`

- Синтаксис операндов: `crfD, rA, UIMM`
- Размер операндов: REG_SIZE
- Операция: содержимое `rA` сравнивается с содержимым `rB` и устанавливаются коды условия в поле `crfD` регистра `cr`; операнды трактуются как беззнаковые целые

4.3.3.3 Логические инструкции процессоров PowerPC

В большинстве инструкций можно установить код условия `cr0` по результату.

`ANDI.`

- Синтаксис операндов: `rD, rA, UIMM`
- Размер операндов: REG_SIZE
- Операция: `rA & UIMM -> rD` и установить по результату поле `cr0` в регистре `cr`

`ANDIS.`

- Синтаксис операндов: `rD, rA, UIMM`
- Размер операндов: REG_SIZE
- Операция: `rA & (UIMM<<16) -> rD` и установить по результату поле `cr0` в регистре `cr`

`AND`
`AND.`

- Синтаксис операндов: `rD, rA, rB`
- Размер операндов: REG_SIZE
- Операция: `rA & rB -> rD`; если присутствует суффикс '.', то установить по результату поле `cr0` в регистре `cr`

`NAND`
`NAND.`

- Синтаксис операндов: `rD, rA, rB`
- Размер операндов: REG_SIZE
- Операция: `~(rA & rB) -> rD`; если присутствует суффикс '.', то установить по результату поле `cr0` в регистре `cr`

`ANDC`
`ANDC.`

- Синтаксис операндов: `rD, rA, rB`
- Размер операндов: REG_SIZE
- Операция: `rA & ~rB -> rD`; если присутствует суффикс '.', то установить по результату поле `cr0` в регистре `cr`

`ORI`

- Синтаксис операндов: `rD, rA, UIMM`
- Размер операндов: REG_SIZE
- Операция: `rA | UIMM -> rD`

`ORIS`

- Синтаксис операндов: `rD, rA, UIMM`
- Размер операндов: REG_SIZE
- Операция: `rA | (UIMM<<16) -> rD`

`OR`
`OR.`

- Синтаксис операндов: `rD, rA, rB`
- Размер операндов: REG_SIZE
- Операция: `rA | rB -> rD`; если присутствует суффикс '.', то установить по результату поле `cr0` в регистре `cr`

`NOR`
`NOR.`

- Синтаксис операндов: `rD, rA, rB`
- Размер операндов: REG_SIZE
- Операция: `~(rA | rB) -> rD`; если присутствует суффикс '.', то установить по результату поле `cr0` в регистре `cr`

`ORC`
`ORC.`

- Синтаксис операндов: `rD, rA, rB`
- Размер операндов: REG_SIZE
- Операция: `rA | ~rB -> rD`; если присутствует суффикс '.', то установить по результату поле `cr0` в регистре `cr`

`XORI`

- Синтаксис операндов: `rD, rA, UIMM`
- Размер операндов: REG_SIZE
- Операция: `rA ^ UIMM -> rD`

`XORIS.`

- Синтаксис операндов: `rD, rA, UIMM`
- Размер операндов: REG_SIZE
- Операция: `rA ^ (UIMM<<16) -> rD`

`XOR`
`XOR.`

- Синтаксис операндов: `rD, rA, rB`
- Размер операндов: REG_SIZE
- Операция: `rA ^ rB -> rD`; если присутствует суффикс '.', то установить по результату поле `cr0` в регистре `cr`

`EQV`
`EQV.`

- Синтаксис операндов: `rD, rA, rB`
- Размер операндов: REG_SIZE
- Операция: `~(rA ^ rB) -> rD`; если присутствует суффикс '.', то установить по результату поле `cr0` в регистре `cr`

Логические инструкции, определенные с `cr` (condition register):

`CRAND`

- Синтаксис операндов: `crbD, crbA, crbB`
- Размер операндов: нет
- Операция: `(бит crbA) & (бит crbB) -> (бит crbD)`

`CRNAND`

- Синтаксис операндов: `crbD, crbA, crbB`
- Размер операндов: нет
- Операция: `~((бит crbA) & (бит crbB)) -> (бит crbD)`

`CRANDC`

- Синтаксис операндов: `crbD, crbA, crbB`
- Размер операндов: нет
- Операция: `(бит crbA) & ~(бит crbB) -> (бит crbD)`

`CROR`

- Синтаксис операндов: `crbD, crbA, crbB`
- Размер операндов: нет

– Операция: `(бит crbA) | (бит crbB) -> (бит crbD)`

`CRNOR`

– Синтаксис операндов: `crbD, crbA, crbB`
– Размер операндов: нет
– Операция: `~((бит crbA) | (бит crbB)) -> (бит crbD)`

`CRORC`

– Синтаксис операндов: `crbD, crbA, crbB`
– Размер операндов: нет
– Операция: `(бит crbA) | ~(бит crbB) -> (бит crbD)`

`CRXOR`

– Синтаксис операндов: `crbD, crbA, crbB`
– Размер операндов: нет
– Операция: `(бит crbA) ^ (бит crbB) -> (бит crbD)`

`CREQV`

– Синтаксис операндов: `crbD, crbA, crbB`
– Размер операндов: нет
– Операция: `~((бит crbA) ^ (бит crbB)) -> (бит crbD)`

4.3.3.4 Инструкции сдвига процессоров PowerPC

Во всех инструкциях можно установить код условия `cr0` по результату.

`SLW`
`SLW.`

– Синтаксис операндов: `rD, rA, rB`
– Размер операндов: REG_SIZE
– Операция: `rA << rB -> rD` (вдвигаются нули); если присутствует суффикс '.', то установить по результату поле `cr0` в регистре `cr`

`SRW`
`SRW.`

– Синтаксис операндов: `rD, rA, rB`
– Размер операндов: REG_SIZE
– Операция: `rA >> rB -> rD` (вдвигаются нули); если присутствует суффикс '.', то установить по результату поле `cr0` в регистре `cr`

`SRAW`
`SRAW.`

– Синтаксис операндов: `rD, rA, rB`
– Размер операндов: REG_SIZE

– Операция: `rA >> rB -> rD` (вдвигается знаковый разряд `rA`); если присутствует суффикс '.', то установить по результату поле `cr0` в регистре `cr`

`SRAWI`
`SRAWI.`

– Синтаксис операндов: `rD, rA, SH`
– Размер операндов: REG_SIZE
– Операция: `rA >> SH -> rD` (вдвигается знаковый разряд `rA`); если присутствует суффикс '.', то установить по результату поле `cr0` в регистре `cr`

`RLWINM`
`RLWINM.`

– Синтаксис операндов: `rD, rA, SH, MB, ME`
– Размер операндов: REG_SIZE
– Операция: `(rA <rotate left> SH) & MASK(MB, ME) -> rD`; если присутствует суффикс '.', то установить по результату поле `cr0` в регистре `cr`; (здесь `MASK(MB, ME)` - целое число, в двоичном представлении которого 1 стоят в битах с номерами от `MB` до `ME` (включительно))

`RLWNM`
`RLWNM.`

– Синтаксис операндов: `rD, rA, rB, MB, ME`
– Размер операндов: REG_SIZE
– Операция: `(rA <rotate left> rB) & MASK(MB, ME) -> rD`; если присутствует суффикс '.', то установить по результату поле `cr0` в регистре `cr`

`RLWIMI`
`RLWIMI.`

– Синтаксис операндов: `rD, rA, SH, MB, ME`
– Размер операндов: REG_SIZE
– Операция: `((rA <rotate left> SH) & MASK(MB,ME)) | (rD & ~MASK(MB, ME)) -> rD`; если присутствует суффикс '.', то установить по результату поле `cr0` в регистре `cr`

4.3.3.5 Инструкции управления процессоров PowerPC

Безусловные переходы используют следующие режимы адресации

– относительный (в инструкции задается смещение относительно текущей инструкции)
– абсолютный (в инструкции задается адрес перехода)

Нужный режим выделяется значением бита (AA, бит 30) в инструкции.

Условные переходы используют следующие режимы адресации

- относительный
- абсолютный
- адрес перехода содержится в регистре `lr` (link register)
- адрес перехода содержится в регистре `ctr` (count register)

Во всех инструкциях перехода можно установить бит (LK, бит 31), который означает, что адрес следующей инструкции будет помещен в `lr` (link register).

Во всех условных переходах используется операнд `BO`, задающий код условия. Это 5-ти битовое поле, принимающее следующие значения

`0000y` `ctr=ctr-1`, затем переход, если `ctr` не 0 и условие ЛОЖНО

`0001y` `ctr=ctr-1`, затем переход, если `ctr` равен 0 и условие ЛОЖНО

`001zy` переход, если условие ЛОЖНО

`0100y` `ctr=ctr-1`, затем переход, если `ctr` не 0 и условие ИСТИННО

`0101y` `ctr=ctr-1`, затем переход, если `ctr` равен 0 и условие ИСТИННО

`011zy` переход, если условие ИСТИННО

`1z00y` `ctr=ctr-1`, затем переход, если `ctr` не 0

`1z01y` `ctr=ctr-1`, затем переход, если `ctr` равен 0

`1z1zz` переход всегда

Здесь `z` означает, что данный бит игнорируется (должен быть 0 для совместимости с будущими моделями), `y` бит задает информацию о том, будет или нет (наиболее вероятно) выполнен этот переход.

Наиболее часто используются значения `BO=4` (переход, если условие не выполнено) и `BO=12` (переход, если условие выполнено).

`B`
`BA`
`BL`
`BLA`

- Синтаксис операндов: `address`
- Размер операндов: `address` - 24 бита (умножается на 4)
- Операция: переход; если нет суффикса '`a`', то `address` задает смещение относительно текущей инструкции, если присутствует суффикс '`a`', то `address` задает адрес перехода; если присутствует суффикс '`l`', то положить адрес следующей инструкции в регистр `lr`

`BC`
`BCA`
`BCL`
`BCLA`

- Синтаксис операндов: `BO, BI, address`
- Размер операндов: `address` - 16 бит (умножается на 4)

– Операция: условный переход, BI задает номер бита в регистре cr, который используется в качестве условия; операнд BO описан выше; если нет суффикса 'a', то address задает смещение относительно текущей инструкции, если присутствует суффикс 'a', то address задает адрес перехода; если присутствует суффикс 'l', то положить адрес следующей инструкции в регистр lr

BCLR
BCLRL

– Синтаксис операндов: BO, BI
– Размер операндов: нет (фиксированный)
– Операция: условный переход по адресу, содержащемуся в регистре lr, BI задает номер бита в регистре cr, который используется в качестве условия; операнд BO описан выше; если присутствует суффикс 'l', то положить адрес следующей инструкции в регистр lr

BCCTR
BCCTRL

– Синтаксис операндов: BO, BI
– Размер операндов: нет (фиксированный)
– Операция: условный переход по адресу, содержащемуся в регистре ctr, BI задает номер бита в регистре cr, который используется в качестве условия; операнд BO описан выше; если присутствует суффикс 'l', то положить адрес следующей инструкции в регистр lr

4.3.3.6 Мультипроцессорные инструкции процессоров PowerPC

Работа с семафорами осуществляется с помощью инструкций lwarx и stwcx.. После чтения слова инструкцией lwarx адрес этого слова запоминается процессором и активизируется bus snooping механизм. Запись по этому адресу инструкцией stwcx. будет возможна лишь в том случае, если по этому адресу не было записи после инструкции lwarx.

LWARX

– Синтаксис операндов: rD, d(rA)
– Размер операндов: REG_SIZE
– Операция: mem((rA|0)+d) -> rD, адрес (rA|0)+d запоминается и включается bus snooping механизм

STWCX.

– Синтаксис операндов: rS, d(rA)
– Размер операндов: REG_SIZE
– Операция: если ранее была инструкция LWARX с тем же адресом (rA|0)+d и не было записи по этому адресу, то rS -> mem((rA|0)+d), бит EQ в cr =1 и адрес (rA|0)+d удаляется из списка слежения bus snooping механизма; иначе запись не производится, бит EQ в cr =0 и адрес (rA|0)+d удаляется из списка слежения bus snooping механизма

4.3.3.7 Упрощенная ассемблерная мнемоника процессоров PowerPC

Приведем некоторые упрощенные ассемблерные инструкции, которые используются для удобства чтения ассемблерных текстов. Эти инструкции являются сокращениями для описанных ранее инструкций PowerPC.

`SUBI` `SUBI rD, rA, value` эквивалентна `ADDI rD, rA, -value`

`SUBIS` `SUBIS rD, rA, value` эквивалентна `ADDIS rD, rA, -value`

`SUBIC` `SUBIC rD, rA, value` эквивалентна `ADDIC rD, rA, -value`

`SUBIC.` `SUBIC. rD, rA, value` эквивалентна `ADDIC. rD, rA, -value`

`SUB` `SUB rD, rA, rB` эквивалентна `SUBF rD, rB, rA`

`SUBC` `SUBC rD, rA, rB` эквивалентна `SUBFC rD, rB, rA`

`NOP` `NOP` эквивалентна `ORI 0, 0, 0`

`LI` `LI rD, value` эквивалентна `ADDI rD, 0, value`

`LIS` `LIS rD, value` эквивалентна `ADDIS rD, 0, value`

`LA` `LA rD, d(rA)` эквивалентна `ADDI rD, rA, d`

`MR` `MR rD, rS` эквивалентна `OR rD, rS, rS`

`NOT` `NOT rD, rS` эквивалентна `NOR rD, rS, rS`

`MFXER` `MFXER rD` эквивалентна `MFSPR rD, xer`

`MFLR` `MFLR rD` эквивалентна `MFSPR rD, lr`

`MFCTR` `MFCTR rD` эквивалентна `MFSPR rD, ctr`

`MTXER` `MTXER rS` эквивалентна `MFSPR xer, rS`

`MTLR` `MTLR rS` эквивалентна `MFSPR lr, rS`

`MTCTR` `MTCTR rS` эквивалентна `MFSPR ctr, rS`

4.4 Примеры кода для семейства PowerPC

Мы используем ассемблерный синтаксис и соглашения о вызовах операционной системы AIX. Основные соглашения:

`Stack Pointer (SP)`
= регистр `r1`

`Frame Pointer (FP)`
= регистр `r31`

`Адрес возврата из функции` **(диктуется системой команд)**
= регистр `lr`

`Возвращаемое значение функции`
= регистр `r3`

`Аргументы функции`
: = регистры `r3`, ..., `r10`; при этом под каждый из этих регистров вызвавшая функция резервирует место в стеке: `mem(SP+24)`, ..., `mem(SP+52)`; если аргументов больше, чем 8, то последующие размещаются в `mem(SP+56)`, `mem(SP+60)`, ..., здесь `SP` - в точке входа в функцию

`Регистры, не сохраняемые при вызове функции`
: = `r0`, `r3`, ..., `r12`

`Регистры, сохраняемые при вызове функции`
: = `r1`, `r2`, `r13`, ..., `r31`

`Указатель на TOC (Table of Contents)`
: = регистр `r2`

Вызвавшая функция резервирует 24 байта в стеке для использования их вызванной функцией, это ячейки `mem(SP)`, ..., `mem(SP+20)`. В ячейке `mem(SP+8)` вызванная функция сохраняет регистр `lr`. Остальные ячейки используются обработчиками прерываний для сохранения состояния текущего процесса (например, регистра `ccr` и т.д.). Таким образом, минимальным размером стекового кадра у функции, вызывающей другие функции, будет 24 (эта область сохранения) + 32 (8x4 - для регистров с аргументами) = 56 байт.

Мы используем те же примеры исходного кода, что и для процессоров Motorola 68xxx.

1.
```
.f:
        li 3,0
        blr
```
2.
```
.toc
LC..0:
        .tc a
.f:
        lwz 9,LC..0(2)
        lwz 3,0(9)
        blr
```
3.
```
.toc
LC..0:
        .tc a
.f:
        lwz 3,LC..0(2)
        blr
```
4.
```
.f:
        blr
.g:
        mflr 0
```

```
        stw 0,8(1)
        stwu 1,-56(1)
        li 3,1
        bl .f
        cror 31,31,31
        addi 1,1,56
        lwz 0,8(1)
        mtlr 0
        blr
```

5.

```
.f:
        add 3,3,4
        blr
.toc
LC..0:
        .tc a
.g:
        mflr 0
        stw 0,8(1)
        stwu 1,-56(1)
        lwz 9,LC..0(2)
        lwz 4,0(9)
        li 3,1
        bl .f
        cror 31,31,31
        addi 1,1,56
        lwz 0,8(1)
        mtlr 0
        blr
```

6.

```
.f:
        mr 10,3
        li 3,0
        cmpw 1,3,4
        li 11,0
        bclr 4,4
        li 9,0
L..5:
        addi 11,11,1
        cmpw 1,11,4
        lwzx 0,9,10
        addi 9,9,4
        add 3,3,0
        bc 12,4,L..5
        blr
.toc
LC..0:
        .tc n
```

```
LC..1:
        .tc a
.g:
        mflr 0
        stw 0,8(1)
        stwu 1,-56(1)
        lwz 9,LC..0(2)
        lwz 3,LC..1(2)
        lwz 4,0(9)
        bl .f
        cror 31,31,31
        addi 1,1,56
        lwz 0,8(1)
        mtlr 0
        blr
```

7.

```
.f:
        mr 8,3
        li 3,0
        cmpw 1,3,5
        li 10,0
        bclr 4,4
        li 11,0
L..5:
        lwzx 0,11,8
        lwzx 9,11,4
        mullw 0,0,9
        addi 10,10,1
        cmpw 1,10,5
        addi 11,11,4
        add 3,3,0
        bc 12,4,L..5
        blr
```

8.

```
.f:
        cmpw 1,3,4
        cmpw 6,3,5
        cror 7,6,5
        cror 27,26,25
        mfcr 0
        rlwinm 9,0,8,1
        rlwinm 0,0,28,1
        and. 11,9,0
        bclr 4,2
        cmpw 1,4,5
        mfcr 0
        rlwinm 0,0,5,1
        neg 0,0
```

```
and 9,5,0
andc 3,4,0
or 3,9,3
blr
```

5 Архитектура процессоров SPARC

5.1 Общий обзор процессоров SPARC

В 1987 году Sun Microsystems аннонсировала SUN-4 - первую компьютерную систему, основанную на новой RISC CPU архитектуре SPARC (Scalable Processor ARChitecture). В отличие от многих других процессоров, SPARC позиционировался как открытая архитектура, которую могут использовать все.

SPARC процессоры используют Берклевскую архитектуру, основанную на регистровых окнах. Внутренние регистры образуют блоки с частичным перекрытием. Исследователи Беркли предложили использовать внутренний стек для того, чтобы избежать сохранения и восстановления регистров во внешней памяти. Основной целью было ускорение вызовов процедур за счет минимизации числа обращений к памяти, требуемых для передачи параметров и получения результата.

В каждый момент времени функция может иметь доступ к 32 регистрам: 8 global registers (r0 - r7, общие для всех функций) и окну из 24 регистров (r8 - r31). Регистровые окна перекрываются по 8-ми регистрам. В каждой функции выделяют 8 in registers (r24 - r31 или i0 - i7) (совпадают с out registers в процедуре, вызвавшей данную), 8 local registers (r16 - r23 или l0 - l7) (доступны только данной процедуре) и 8 out registers (r8 - r15 или o0 - o7) (совпадают с in registers в любой процедуре, вызванной данной). При вызове функции происходит переключение окон, так, что вызванная функция получает новый набор регистров l0 - l7, o0 - o7 и разделяет регистры i0 - i7 с вызвавшей функцией (где они адресовались как o0 - o7). Поэтому в регистрах o0 - o7 удобно размещать параметры для вызванной процедуры, в регистрах l0 - l7 - локальные переменные.

Размер каждого окна - 16 регистров (8 out + 8 local регистров). Фирма SUN специфицировала, что число окон может быть от 2 до 32 (в зависимости от реализации). Тем самым общее число регистров в окнах - от 40 (2x16+8) до 520 (32x16+8).

Если глубина вложенности функций превышает число окон, то процессор генерирует прерывание и операционная система должна сохранить часть окон в памяти.

Существует возможность вызвать функцию без переключения окон.

5.2 Основные члены семейства SPARC

Реализация всех SPARC процессоров удовлетворяет версии архитектуры SPARC с тем или иным номером. Важнейшей функцией SPARC International Compatibility and Compliance Committee является выработка и публикация SPARC Compliance Definitions, а также инструкций по переходу от одного определения к другому.

Каждый процессор в приведенных ниже технических данных обладает всеми возможностями предыдущих процессоров для обеспечения совместимости.

SPARC (1987, 55000 транзисторов, 33 MHz, 20 MIPS)

- 136 32-бит регистров
- 32-бит шина адреса и данных

- 4-х стадийный конвейер
- 2 исполняющих устройства: 1 IU (data + address) + 1 shift unit
- внешний CMU (Cache controller and MMU) и FPC (Floating Point Controller)
- всего 50 инструкций, каждая исполняется за 1 цикл

MICROSPARC-II (1994, 85 MHz, 85 MIPS, 64 Specint 92, 55 Specfp 92)

- 32-бит SPARC V8 архитектура
- 16Кб кэш инструкций + 8Кб кэш данных (32-бит Гарвардская архитектура)
- 3 исполняющих устройства: 1 IU + 1 FPU + 1 MMU
- интерфейс сопроцессора
- встроенная логика для поддержки DRAM и ввода/вывода

SuperSPARC-II (1995, 3100000 транзисторов, 90 MHz, 148 Specint 92, 143 Specfp 92)

- 32-бит SPARC V8 архитектура
- 3 инструкции за цикл
- 8-ми портовый 32x32 файл регистров
- 20Кб кэш инструкций + 16Кб кэш данных (32-бит Гарвардская архитектура)
- 64-входовый TLB (для обоих кэшей)
- 64-бит шина данных
- 4-х стадийный конвейер
- 7 исполняющих устройств: 3 IU + 1 FPU multiply + 1 FPU divide + 1 BU + 1 load/store unit
- биты предсказания переходов

UltraSPARC-II (1996, 250 MHz, 400 Specint 92, 450 Specfp 92)

- 64-бит SPARC V9 архитектура
- 4 реально исполняемых инструкции за цикл
- суперскалярный процессор: 6 исполняющих устройств: 2 IU + 2 FPU + 1 BU + 1 load/store unit
- 9-ти стадийный конвейер
- предсказание переходов
- register scoreboarding
- bi-endian (big- и little-endian порядок байтов)
- 64-бит виртуальный адрес и целочисленные данные
- спекулятивное исполнение
- многоуровневая обработка прерываний, организованных в стек
- встроенная мультимедийная поддержка для 2-х и 3-х мерной графики и новый оптимизированный набор мультимедиа инструкций

- поддержка сильно связанных процессоров (до 4-х процессоров могут разделять одну шину адреса)

5.3 Программная модель семейства SPARC

В этом разделе мы рассмотрим процессоры семейства SPARC (версии V8) с точки зрения программиста (или компилятора).

5.3.1 Набор регистров процессоров SPARC

Прикладной программе доступны

32 целочисленных регистра общего назначения (РОН) r0 - r31
: содержат слово (32 бит). Любая пара `rN`, `r(N+1)` при четном `N` может содержать данные длиной 64 бит. Регистры организованы в виде окон по 24 регистра с перекрытием по 8-ми регистрам. Выделяют

  `g0 - g7` — это регистры `r0 - r7`, являются общими для всех функций (т.е. не участвуют в переключении регистровых окон). Регистр `g0` - специальный: при чтении из него всегда читается 0, при записи в него ничего не происходит (запись не производится, из регистра всегда читается 0)

  `o0 - o7` — это регистры `r8 - r15` (`out registers`); являются `in registers` для любой функции, вызванной данной (т.е. доступны вызванной функции); регистр `o7` - специальный, инструкция `CALL` записывает в него свой собственный адрес (т.е. адрес возврата - 8)

  `l0 - l7` — это регистры `r16 - r23` (`local registers`); не доступны ни функции, вызвавшей данную, ни функциям, вызванным данной

  `i0 - i7` — это регистры `r24 - r31` (`in registers`); являются `out registers` для функции, вызвавшей данную (т.е. доступны вызвавшей функции и не доступны любой функции, вызванной данной)

32 регистра с плавающей точкой f0 - f31
: содержат значение с плавающей точкой (32 бит). Любая пара `fN`, `f(N+1)` при четном `N` может содержать данные длиной 64 бит (floating point double), любая четверка `fN`, `f(N+1)`, `f(N+2)`, `f(N+3)` при кратном 4-м `N` может содержать данные длиной 128 бит (floating point quadre).

y (multiply/divide register)
: содержит старшую часть произведения (в инструкциях умножения) или старшую часть делимого (в инструкциях деления); читается и записывается инструкциями `RDY` и `WRY`

Целочисленные коды условия icc (integer condition code)
: являются частью `PSR` (Processor State Register), который не доступен пользовательской программе как регистр. Коды условия:

`N - negative`
: устанавливается в 1, если результат меньше 0, иначе - 0

`Z - zero`
: устанавливается в 1, если результат равен 0, иначе - 0

`V - overflow`
: устанавливается в 1, если результат не входит в диапазон представимых значений, иначе - 0

`C - carry`
: устанавливается в 1, если в результате произошел перенос в самом старшем разряде, иначе - 0

Коды условия устанавливаются по результату арифметических операций (если это указано в инструкции) или специальными инструкциями и используются в командах условного перехода.

5.3.2 Режимы адресации памяти процессоров SPARC

Поддерживаются следующие два режима адресации

- `<ea> = rA+offset` (включая `offset`=0)
- `<ea> = rA+rB`

где обозначено

– `rA`, `rA` - РОН `r0...r31`
– `offset` - 13-бит смещение (знаково расширяемое)

5.3.3 Основные инструкции процессоров SPARC

Все инструкции (за исключением load/store) имеют операнды в регистрах и потому размер всех операндов равен размеру слова (32 бит).

5.3.3.1 Инструкции пересылки данных процессоров SPARC

`LDSB`
`LDSH`

– Синтаксис операндов: `[address], rD`
– Размер операндов: соответственно 8, 16 `->` 32
– Операция: `mem([address]) -> rD` со знаковым расширением

`LDUB`
`LDUH`

– Синтаксис операндов: `[address], rD`
– Размер операндов: соответственно 8, 16 `->` 32
– Операция: `mem([address]) -> rD` с расширением нулем

`LD`

– Синтаксис операндов: `[address], rD`

- Размер операндов: 32
- Операция: `mem([address]) -> rD`

`LDD`

- Синтаксис операндов: `[address], rD`
- Размер операндов: 64
- Операция: `mem([address]) -> (rD, r(D+1))`

`STB`
`STH`
`ST`

- Синтаксис операндов: `rS, [address]`
- Размер операндов: соответственно 32 -> 8, 16, 32
- Операция: `rS -> mem([address])`

`STD`

- Синтаксис операндов: `rS, [address]`
- Размер операндов: соответственно 64
- Операция: `(rS, r(S+1) -> mem([address])`

`RD`

- Синтаксис операндов: `y, rD`
- Размер операндов: 32
- Операция: `y -> rD`

`WR`

- Синтаксис операндов: `rA, rB/SIMM13, y`
- Размер операндов: 32
- Операция: `rA^(rB/SIMM13) -> y;`

5.3.3.2 Целочисленные арифметические инструкции процессоров SPARC

В большинстве инструкций можно установить коды условия `icc` по результату.

`SETHI`

- Синтаксис операндов: `const22, rD`
- Размер операндов: 22 -> 32
- Операция: `(const22<<10) -> rD`

`ADD`
`ADDCC`

- Синтаксис операндов: `rA, rB/SIMM13, rD`
- Размер операндов: 32

- Операция: `rA+(rB/SIMM13) -> rD`; если присутствует суффикс '`CC`', то установить по результату коды условия `icc`

`ADDX`
`ADDXCC`

- Синтаксис операндов: `rA, rB/SIMM13, rD`
- Размер операндов: 32
- Операция: `rA+(rB/SIMM13)+C -> rD`; если присутствует суффикс '`CC`', то установить по результату коды условия `icc`

`SUB`
`SUBCC`

- Синтаксис операндов: `rA, rB/SIMM13, rD`
- Размер операндов: 32
- Операция: `rA-(rB/SIMM13) -> rD`; если присутствует суффикс '`CC`', то установить по результату коды условия `icc`

`SUBX`
`SUBXCC`

- Синтаксис операндов: `rA, rB/SIMM13, rD`
- Размер операндов: 32
- Операция: `rA-(rB/SIMM13)-C -> rD`; если присутствует суффикс '`CC`', то установить по результату коды условия `icc`

`UMUL`
`UMULCC`

- Синтаксис операндов: `rA, rB/SIMM13, rD`
- Размер операндов: 32
- Операция: `rA*(rB/SIMM13) -> (rD, y)` (беззнаково, в регистре `y` получается старшая часть результата); если присутствует суффикс '`CC`', то установить по результату коды условия `icc`

`SMUL`
`SMULCC`

- Синтаксис операндов: `rA, rB/SIMM13, rD`
- Размер операндов: 32
- Операция: `rA*(rB/SIMM13) -> (rD, y)` (знаково, в регистре `y` получается старшая часть результата); если присутствует суффикс '`CC`', то установить по результату коды условия `icc`

`UDIV`
`UDIVCC`

- Синтаксис операндов: `rA, rB/SIMM13, rD`
- Размер операндов: 32

– Операция: `(y,rA)/(rB/SIMM13) -> rD` (беззнаково, в регистре `y` находится старшая часть делимого); если присутствует суффикс '`CC`', то установить по результату коды условия `icc`

`SDIV`
`SDIVCC`

– Синтаксис операндов: `rA, rB/SIMM13, rD`
– Размер операндов: 32
– Операция: `(y,rA)/(rB/SIMM13) -> rD` (знаково, в регистре `y` находится старшая часть делимого); если присутствует суффикс '`CC`', то установить по результату коды условия `icc`

`SAVE`

– Синтаксис операндов: `rA, rB/SIMM13, rD`
– Размер операндов: 32
– Операция: `rA+(rB/SIMM13) -> rD` и создать новое регистровое окно; при этом `rA` и `rB` - из старого (до этой инструкции) регистрового окна, `rD` - из нового (после этой инструкции) регистрового окна

`RESTORE`

– Синтаксис операндов: `rA, rB/SIMM13, rD`
– Размер операндов: 32
– Операция: `rA+(rB/SIMM13) -> rD` и вернутся к предыдущему регистровому окну; при этом `rA` и `rB` - из старого (до этой инструкции) регистрового окна, `rD` - из нового (после этой инструкции) регистрового окна

5.3.3.3 Логические инструкции процессоров SPARC

Во всех инструкциях можно установить коды условия `icc` по результату.

`AND`
`ANDCC`

– Синтаксис операндов: `rA, rB/SIMM13, rD`
– Размер операндов: 32
– Операция: `rA & (rB/SIMM13) -> rD`; если присутствует суффикс '`CC`', то установить по результату коды условия `icc`

`ANDN`
`ANDNCC`

– Синтаксис операндов: `rA, rB/SIMM13, rD`
– Размер операндов: 32
– Операция: `rA & ~(rB/SIMM13) -> rD`; если присутствует суффикс '`CC`', то установить по результату коды условия `icc`

`OR`
`ORCC`

- Синтаксис операндов: `rA, rB/SIMM13, rD`
- Размер операндов: 32
- Операция: `rA | (rB/SIMM13) -> rD`; если присутствует суффикс ‘`CC`’, то установить по результату коды условия `icc`

`ORN`
`ORNCC`

- Синтаксис операндов: `rA, rB/SIMM13, rD`
- Размер операндов: 32
- Операция: `rA | ~(rB/SIMM13) -> rD`; если присутствует суффикс ‘`CC`’, то установить по результату коды условия `icc`

`XOR`
`XORCC`

- Синтаксис операндов: `rA, rB/SIMM13, rD`
- Размер операндов: 32
- Операция: `rA ^ (rB/SIMM13) -> rD`; если присутствует суффикс ‘`CC`’, то установить по результату коды условия `icc`

`XNOR`
`XNORCC`

- Синтаксис операндов: `rA, rB/SIMM13, rD`
- Размер операндов: 32
- Операция: `rA ^ ~(rB/SIMM13) -> rD`; если присутствует суффикс ‘`CC`’, то установить по результату коды условия `icc`

5.3.3.4 Инструкции сдвига процессоров SPARC

`SLL`

- Синтаксис операндов: `rA, rB/SIMM13, rD`
- Размер операндов: 32
- Операция: `rA << (rB/SIMM13) -> rD` (вдвигаются нули)

`SRL`

- Синтаксис операндов: `rA, rB/SIMM13, rD`
- Размер операндов: 32
- Операция: `rA >> (rB/SIMM13) -> rD` (вдвигаются нули)

`SRA`

- Синтаксис операндов: `rA, rB/SIMM13, rD`
- Размер операндов: 32
- Операция: `rA >> (rB/SIMM13) -> rD` (вдвигается знаковый разряд `rA`)

5.3.3.5 Инструкции управления процессоров SPARC

Все инструкции передачи управления имеют `delayed slot`: в случае, если переход произойдет, следующая за инструкцией команда будет выполнена до перехода. Это сделано для повышения производительности процессора.

Условные переходы:

`Bcc`

- Синтаксис операндов: `<label>`
- Размер операндов: `<label>` - 22 бит (умножается на 4)
- Операция: если условие истинно, то `PC + offset(<label>) -> PC`

Коды условия `cc` здесь:

- A - always
- N - never
- NE - not equal
- E - equal
- G - greater
- LE - less or equal
- GE - greater or equal
- L - less
- GU - greater unsigned
- LE - less or equal unsigned
- CC - carry clear (greater or equal unsigned)
- CS - carry set (less unsigned)
- POS - positive
- NEG - negative
- VC - overflow clear
- VS - overflow set

Безусловные переходы (к ним относятся также описанные выше инструкции `BA` и `BN`):

`CALL`

- Синтаксис операндов: `<label>`
- Размер операндов: `<label>` - 30 бит (умножается на 4)
- Операция: `PC -> r15`, `PC + offset(<label>) -> PC`

`JMPL`

- Синтаксис операндов: `<address>, rD`
- Размер операндов: 32
- Операция: `PC -> rD`, `<address> -> PC`

`NOP`

- Синтаксис операндов: нет
- Размер операндов: нет
- Операция: `PC -> PC+4`

5.3.3.6 Мультипроцессорные инструкции процессоров SPARC

`LDSTUB`

- Синтаксис операндов: `[address], rD`
- Размер операндов: 8 `->` 32
- Операция: `mem([address]) -> rD` с расширением нулем, все единицы `-> mem([address])`; операция выполняется неразрывно (не может быть прервана)

`SWAP`

- Синтаксис операндов: `[address], rD`
- Размер операндов: 32
- Операция: `mem([address]) <-> rD`; операция выполняется неразрывно (не может быть прервана)

5.3.3.7 Упрощенная ассемблерная мнемоника процессоров SPARC

Приведем некоторые упрощенные ассемблерные инструкции, которые используются для удобства чтения ассемблерных текстов. Эти инструкции являются сокращениями для описанных ранее инструкций SPARC.

`CMP` `CMP rA, rB/SIMM13` эквивалентна `SUBCC rA, rB/SIMM13, r0`

`JMP` `JMP <address>` эквивалентна `JMPL <address>, r0`

`CALL` `CALL <address>` эквивалентна `JMPL <address>, r15`

`TST` `TST rA` эквивалентна `ORCC r0, rA, r0`

`RET` `RET` эквивалентна `JMPL r31+8, r0`

`RETL` `RETL` эквивалентна `JMPL r15+8, r0`

`NOT` `NOT rA, rD` эквивалентна `XNOR rA, r0, rD`

`NEG` `NEG rA, rD` эквивалентна `SUB r0, rA, rD`

`CLR` `CLR rD` эквивалентна `OR r0, r0, rD`

`MOV`

`MOV rA/SIMM13, rD` эквивалентна `OR r0, rA/SIMM13, rD`

`MOV y, rD` эквивалентна `RD y, rD`

`MOV rA/SIMM13, y` эквивалентна `WR r0, rA/SIMM13, y`

5.4 Примеры кода для семейства SPARC

Мы используем ассемблерный синтаксис и соглашения о вызовах операционной системы SunOS для SPARC.

Основные соглашения для функции, использующей переключение регистровых окон с помощью команды `SAVE` (это все функции, вызывающие другие функции):

`Stack Pointer (SP)`
- для функции, вызвавшей данную = регистр `i6` (`r30`)
- для текущей функции = регистр `o6` (`r14`)

`Frame Pointer (FP)`
- для функции, вызвавшей данную - не доступен
- для текущей функции = регистр `i6` (`r30`)

`Адрес возврата из функции` **(диктуется системой команд)**
= регистр `i7` (`r31`)

`Возвращаемое значение функции`
- для функции, вызвавшей данную = регистр `o0` (`r8`)
- для текущей функции = регистр `i0` (`r24`)

`Аргументы функции`
- для функции, вызвавшей данную: регистры `o0`, ..., `o5` (`r8`, ..., `r13`); при этом под каждый из этих регистров эта функция резервирует место в стеке: `mem(SP+68)`, ..., `mem(SP+88)`; если аргументов больше, чем 6, то последующие размещаются в `mem(SP+92)`, `mem(SP+96)`, ..., здесь `SP` - значение `SP` в этой (вызвавшей) функции
- для текущей функции: регистры `i0`, ..., `i5` (`r24`, ..., `r29`); если аргументов больше, чем 6, то последующие размещаются в `mem(FP+92)`, `mem(FP+96)`, ..., здесь `FP` - в точке входа в функцию (равно `SP` в вызвавшей функции)

`Регистры, не сохраняемые при вызове функции`
= `g0`, ..., `g4`, `o0`, ..., `o5`, `o7` (`r0`, ..., `r4`, `r8`, ..., `r13`, `r15`)

`Регистры, сохраняемые при вызове функции`
= `g5`, `g6`, `g7`, `l0`, ..., `l7`, `i0`, ..., `i7` (`r5`, `r6`, `r7`, `r16`, ..., `r31`)

Основные соглашения для функции, не использующей переключение регистровых окон с помощью команды `SAVE` (обычно это "достаточно простая" функция, не вызывающая другие функции):

`Stack Pointer (SP)`
= регистр `o6` (`r14`)

`Frame Pointer (FP)`
= регистр `i6` (`r30`)

`Адрес возврата из функции` **(диктуется системой команд)**
= регистр `o7` (`r15`)

`Возвращаемое значение функции`
: = регистр `o0` (`r8`)

`Аргументы функции`
: регистры `o0`, ..., `o5` (`r8`, ..., `r13`); при этом под каждый из этих регистров вызвавшая функция резервирует место в стеке: `mem(SP+68)`, ..., `mem(SP+88)`; если аргументов больше, чем 6, то последующие размещаются в `mem(SP+92)`, `mem(SP+96)`, ..., здесь `SP` - значение `SP` в вызвавшей функции

`Регистры, не сохраняемые при вызове функции`
: = `g0`, ..., `g4`, `o0`, ..., `o5`, `o7` (`r0`, ..., `r4`, `r8`, ..., `r13`, `r15`)

`Регистры, сохраняемые при вызове функции`
: = `g5`, `g6`, `g7`, `l0`, ..., `l7`, `i0`, ..., `i7` (`r5`, `r6`, `r7`, `r16`, ..., `r31`)

Вызвавшая функция резервирует 16x4=64 байта в стеке для использования их операционной системой для сохранения регистрового окна (16-ти регистров `l0`, ..., `l7`, `o0`, ..., `o7`) в случае, если аппаратные регистровые окна закончились (ячейки `mem(SP)`, ..., `mem(SP+60)`). Еще одна ячейка (`mem(SP+64)`) резервируется вызвавшей функцией для возврата указателя на возвращаемое значение структурного типа (даже если возвращаемое значение является простым). Также 8 байт в стеке резервируются для внутренних нужд программы (например, невозможно напрямую переслать данные между целочисленными регистрами и регистрами с плавающей точкой, для этого надо использовать временную ячейку памяти). Таким образом, минимальным размером стекового кадра у функции, вызывающей другие функции, будет 64 (область сохранения регистров) + 4 (для указателя на возвращаемое значение структурного типа) + 24 (6x4 - для регистров с аргументами) + 8 (резерв) = 100 байт. Это значение округляется до ближайшего, делящегося на 8, т.е. 104.

Отметим, что в ассемблере для SunOS/SPARC инструкция вызова функции "call" имеет второй операнд, который при чтении кода ниже можно игнорировать.

Мы используем те же примеры исходного кода, что и для процессоров Motorola 68xxx.

1.
```
_f:
        retl
        mov 0,%o0
```
2.
```
_f:
        sethi %hi(_a),%g2
        retl
        ld [%g2+%lo(_a)],%o0
```
3.
```
_f:
        sethi %hi(_a),%o0
        retl
        or %o0,%lo(_a),%o0
```
4.

```
_f:
        retl
        nop
_g:
        save %sp,-104,%sp
        call _f,0
        mov 1,%o0
        ret
        restore %g0,%o0,%o0
```

5.

```
_f:
        retl
        add %o0,%o1,%o0
_g:
        save %sp,-104,%sp
        sethi %hi(_a),%o0
        ld [%o0+%lo(_a)],%o1
        call _f,0
        mov 1,%o0
        ret
        restore %g0,%o0,%o0
```

6.

```
_f:
        mov %o0,%o2
        mov 0,%o0
        cmp %o0,%o1
        bge L3
        mov 0,%g3
        sll %o1,2,%o1
L5:
        ld [%g3+%o2],%g2
        add %g3,4,%g3
        cmp %g3,%o1
        bl L5
        add %o0,%g2,%o0
L3:
        retl
        nop
_g:
        save %sp,-104,%sp
        sethi %hi(_a),%o0
        sethi %hi(_n),%o1
        ld [%o1+%lo(_n)],%o1
        call _f,0
        or %o0,%lo(_a),%o0
        ret
        restore %g0,%o0,%o0
```

7.

```
_f:
        mov %o0,%o4
        mov 0,%o0
        cmp %o0,%o2
        bge L3
        mov 0,%o3
        sll %o2,2,%o2
L5:
        ld [%o3+%o4],%g2
        ld [%o3+%o1],%g3
        smul %g2,%g3,%g2
        add %o3,4,%o3
        cmp %o3,%o2
        bl L5
        add %o0,%g2,%o0
L3:
        retl
        nop
```

8.

```
_f:
        cmp %o0,%o1
        bl L2
        cmp %o0,%o2
        bge L3
        nop
L2:
        cmp %o1,%o2
        bl L3
        mov %o2,%o0
        mov %o1,%o0
L3:
        retl
        nop
```

6 Архитектура процессоров ARM

6.1 Общий обзор процессоров ARM

Компания ARM (Advansed RISC Machines) была основана в ноябре 1990 года фирмами

- Acorn Computers (информационные технологии для образования, Великобритания)
- Apple Computers
- VLSI Technology

Основной целью компании является разработка микропроцессорных ядер и их лицензирование широкому кругу производителей. В силу малости процессорного ядра ARM (всего 35000 транзисторов в базовом ядре ARM7) оно идеально подходит для интеграции в специализированные микросхемы потребителей. ARM Design Service Group постоянно работает с партнерами, обеспечивая ARM экспертизу OEM потребителям, желающим иметь встроенные в микросхемы решения на основе ядер ARM.

В настоящее время следующие компании лицензировали ARM и производят микросхемы на его основе:

1. VLSI Technology
2. Texas Instruments (TI)
3. Samsung Corporation
4. NEC Corporation
5. GEC Plessey Semiconductors (GPS)
6. Cirrus Logic
7. Digital Equipment Corporation
8. Symbios Logic
9. Sharp Corporation
10. Asahi Kasai Microsystems (AKM)
11. European Silicon Structures (ES2)
12. Lucky Goldstar Corporation
13. Intel Corporation
14. IBM Corporation

На основе ARM ядра разработано более 30 микропроцессоров и специализированных микросхем. Они находят применение в сотовых телефонах, органайзерах, модемах, графических ускорителях, видеофонах, камерах, телефонных коммутаторах, игровых приставках, дисковых накопителях, высокопроизводительных рабочих станциях, автомобильных навигационных системах, цифровых декодерах, smart картах, лазерных принтерах.

Основные отличительные черты архитектуры ARM:

1. Все инструкции являются условными (т.е. выполняются, только если код условия совпадает с кодом, указанным в инструкции). Это позволяет увеличить плотность кода и уменьшить потребность в инструкциях близкого перехода. Как следствие, нет отдельных команд условного перехода.
2. Все целочисленные арифметические инструкции могут выполнять операцию сдвига над операндами за тот же цикл, что выполняется и сама инструкция. Как следствие, нет отдельных команд сдвига.
3. Нет целочисленной инструкции деления.
4. Возможность выполнять DSP-подобные функции:
   1. присутствуют инструкции умножения и умножения со сложением (multiply-accumulate (MLA))
   2. присутствуют инструкции блочного чтения из памяти и блочной записи в память, позволяющие переслать любое подмножество из 16-ти регистров общего назначения.
5. Некоторые модели могут работать в так называемом `THUMB` режиме: инструкции кодируются 16-ю битами вместо 32-х. Это значительно увеличивает плотность кода, но накладывает ряд ограничений на систему команд:
   1. полноценно доступны только 8 регистров из 16-ти, остальные могут ограниченно использоваться только в некоторых инструкциях (`MOV`, `ADD` и `CMP`);
   2. не поддерживается условное исполнение инструкций, как следствие, появилась новая инструкция условного перехода;
   3. не поддерживается операция сдвига над операндами в целочисленных арифметических инструкциях, как следствие, появились новые инструкции сдвига;
   4. все инструкции двухоперандные (а не трехоперандные как в обычном режиме).

6.2 Основные члены семейства ARM

Каждый процессор в приведенных ниже технических данных в-основном обладает всеми возможностями предыдущих процессоров для обеспечения совместимости.

`ARM1` Прототип, использовался только в тестовых системах

`ARM2 (8 MHz, 4.7 MIPS)`
64Кб адресное пространство

`ARM3 (33 MHz, 18 MIPS)`
- ARM2 ядро
- 4Кб единый кэш
- интерфейс сопроцессора
- добавлена новая инструкция `SWP` для работы с семафорами

`ARM6 (36000 транзисторов, 33 MHz, 28 MIPS)`
- 4Гб адресное пространство
- bi-endian (big- и little-endian порядок байтов)

- интерфейс сопроцессора

`ARM600` ARM6 со встроенным MMU

`ARM7 (35000 транзисторов)`
- ARM6 ядро, способное работать на повышенной частоте
- 3-х стадийный конвейер
- улучшенная инструкция аппаратного умножения (нужна для работы DSP)

`ARM7D` ARM7 с поддержкой отладки

`ARM7DM` ARM7D с улучшенным умножением

`ARM7DMI (40 MIPS)`
ARM7DM с ICEbreaker (встроенная поддержка In-Circuit-Emulation)

`ARM70DM` ARM7DMI (как отдельная микросхема)

`ARM700 (40 MHz, 36 MIPS)`
- ARM7 ядро
- 4Кб единый кэш
- writeback buffer
- встроенное MMU

`ARM7500`
- ARM7 ядро
- 8Кб единый кэш
- writeback buffer
- встроенное MMU
- встроенный IOMD
- встроенный видеопроцессор

`ARM7Txx` ARM7xx (xx - одно из приведенных выше сочетаний) с поддержкой `THUMB` режима

`ARM8 (80 MHz, 80 MIPS)`
- совместим с ARM6 и ARM7
- 5-ти стадийный конвейер
- спекулятивное исполнение

`StrongARM (SA110: 100 MHz, 115 MIPS; 200 MHz, 230 MIPS)`
- высокоскоростной вариант ARM ядра, разработан совместно ARM ltd и Digital
- 16Кб кэш инструкций + 16Кб кэш данных (Гарвардская архитектура)
- глубокий конвейер
- полная совместимость кода не гарантируется в силу появления глубокого конвейера и раздельного кэша

AMULET2e (40 MIPS)

- это асинхронная версия ARM6, более быстрая, чем ARM7, но более медленная, чем ARM8
- 150 mW в активном состоянии, 0.1 mW в состоянии ожидания
- малое потребление мощности и механизм использования энергии делают AMULET2e идеальным процессором для приложений, где периоды высокой вычислительной нагрузки сочетаются с длительными периодами ожидания ввода

6.3 Программная модель семейства ARM

В этом разделе мы рассмотрим процессоры семейства ARM с точки зрения программиста (или компилятора).

6.3.1 Набор регистров процессоров ARM

Прикладной программе доступны

16 целочисленных регистров общего назначения (РОН) r0 - r15
: содержат слово (32 бит). Некоторые из этих регистров имеют специальное назначение:

  r15 - program counter (pc)
  : содержит адрес инструкции, находящейся через две инструкции от исполняемой в данный момент (т.е. адрес текущей инструкции + 8; при записи в этот регистр происходит переход по записанному адресу

  r14 - link register (lr)
  : после инструкции Branch and Link (BL) (вызов функции) содержит адрес следующей инструкции (адрес возврата); во всех остальных инструкциях это обычный РОН

Коды условия cc (condition code)
: являются частью CPSR (Current Program Status Register), который не доступен пользовательской программе как регистр. Коды условия:

  N - negative
  : устанавливается в 1, если результат меньше 0, иначе - 0

  Z - zero
  : устанавливается в 1, если результат равен 0, иначе - 0

  V - overflow
  : устанавливается в 1, если результат не входит в диапазон представимых значений, иначе - 0

  C - carry
  : устанавливается в 1, если в результате произошел перенос в самом старшем разряде, иначе - 0

Коды условия устанавливаются по результату арифметических операций (если это указано в инструкции) или специальными инструкциями и используются для определения того, нужно ли исполнять текущую инструкцию (напомним, все инструкции являются условными).

6.3.2 Режимы адресации памяти процессоров ARM

Поддерживаются следующие режимы адресации.

Для чтения/записи слова (32 бит) или беззнакового байта
имеются три режима адресации, каждый из которых имеет три варианта:

- **обычный**, адрес есть сумма базового регистра и смещения (которое может быть константой, другим регистром или регистром, сдвинутым на константу)
- **с преиндекированием**, адрес есть сумма базового регистра и смещения, если инструкция чтения/записи выполнена (удовлетворен ее код условия), то адрес записывается в базовый регистр
- **с постиндексированием**, адрес есть базовый регистр, если инструкция чтения/записи выполнена, то сумма базового регистра и смещения записывается в базовый регистр

Режимы адресации:

1. immediate offset
   - immediate
     Обозначение: [rA, #+/-<12_bit_offset>]
     Значение адреса: <ea> = rA +/- <12_bit_offset>
     Значение rA после выполнения: не изменяется
   - immediate pre-indexed
     Обозначение: [rA, #+/-<12_bit_offset>]!
     Значение адреса: <ea> = rA +/- <12_bit_offset>
     Значение rA после выполнения: = <ea>
   - immediate post-indexed
     Обозначение: [rA], #+/-<12_bit_offset>
     Значение адреса: <ea> = rA
     Значение rA после выполнения: = rA +/- <12_bit_offset>
2. register offset
   - register
     Обозначение: [rA, +/-rB]
     Значение адреса: <ea> = rA +/- rB
     Значение rA после выполнения: не изменяется
   - register pre-indexed
     Обозначение: [rA, +/-rB]!

Значение адреса: <ea> = rA +/- rB

Значение rA после выполнения: = <ea>

- register post-indexed

  Обозначение: [rA], +/-rB

  Значение адреса: <ea> = rA

  Значение rA после выполнения: = rA +/- rB

3. scaled register offset
   - scaled register

     Обозначение: [rA, +/-rB, <shift> #<shift_imm>]

     Значение адреса: <ea> = rA +/- (rB <shift> <shift_imm>)

     Значение rA после выполнения: не изменяется
   - scaled register pre-indexed

     Обозначение: [rA, +/-rB, <shift> #<shift_imm>]!

     Значение адреса: <ea> = rA +/- (rB <shift> <shift_imm>)

     Значение rA после выполнения: = <ea>
   - scaled register post-indexed

     Обозначение: [rA], +/-rB, <shift> #<shift_imm>

     Значение адреса: <ea> = rA

     Значение rA после выполнения: = rA +/- (rB <shift> <shift_imm>)

     Здесь
     - <shift> - одна из операций LSL, LSR, ASR, ROR, RRX (для последней <shift_imm> должен равняться 0, она вызывает сдвиг вправо на 1, в знаковый разряд вдвигается значение бита C (carry bit))
     - <shift_imm> - 5-бит константа, задающая число сдвигов

Для чтения/записи полуслова (16 бит) или чтения знакового байта

(есть только в архитектуре ARM4 и выше) имеется один режим адресации, имеющий три варианта:

- immediate offset

  Обозначение: [rA, #+/-<8_bit_offset>]

  Значение адреса: <ea> = rA +/- <8_bit_offset>

  Значение rA после выполнения: не изменяется
- immediate pre-indexed

  Обозначение: [rA, #+/-<8_bit_offset>]!

  Значение адреса: <ea> = rA +/- <8_bit_offset>

  Значение rA после выполнения: = <ea>
- immediate post-indexed

  Обозначение: [rA], #+/-<8_bit_offset>

Значение адреса: `<ea> = rA`

Значение `rA` после выполнения: `= rA +/- <8_bit_offset>`

6.3.3 Основные инструкции процессоров ARM

Все инструкции имеют поле кода условия (4 бита с номерами 28...31). Если текущее состояние флагов `N`, `Z`, `C`, `V` в регистре `CPSR` совпадает с указанным в поле кода текущей инструкции, то она будет выполнена, иначе - пропущена. Ассемблерная мнемоника для 16 возможных кодов условия в инструкции:

`EQ (Equal)`
: состояние флагов: `Z=1`

`NE (Not Equal)`
: состояние флагов: `Z=0`

`CS/HS (Carry Set/Unsigned Higher or Same)`
: состояние флагов: `C=1`

`CC/LO (Carry Clear/Unsigned Lower)`
: состояние флагов: `C=0`

`MI (Minus/Negative)`
: состояние флагов: `N=1`

`PL (Plus/Positive or Zero)`
: состояние флагов: `N=0`

`VS (Overflow)`
: состояние флагов: `V=1`

`VC (No Overflow)`
: состояние флагов: `V=0`

`HI (Unsigned Higher)`
: состояние флагов: `C=1 & Z=0`

`LS (Unsigned Lower or Same)`
: состояние флагов: `C=0 | Z=1`

`GE (Signed Greater Than or Equal)`
: состояние флагов: `N=V`

`LT (Signed Less Than)`
: состояние флагов: `N!=V`

`GT (Signed Greater Than)`
: состояние флагов: `Z=0 & N=V`

`LE (Signed Less Than or Equal)`
: состояние флагов: `Z=1 | N!=V`

`AL (Always)`
: состояние флагов: не имеет значения

`NV (Never)`
: состояние флагов: не имеет значения

Если инструкция имеет код условия `AL`, то она будет исполнена в любом случае. Код условия `NV` дает неопределенную инструкцию. Отсутствие явного указания кода условия в ассемблерной мнемонике означает код `AL`.

Все инструкции (за исключением load/store) имеют операнды в регистрах и потому размер всех операндов равен размеру слова (32 бит).

6.3.3.1 Инструкции пересылки данных процессоров ARM

Обычные инструкции пересылки:

`LDRccB`
`LDRccH (ARM4 и выше)`
`LDRcc`

- Синтаксис операндов: `rD, [address]`
- Размер операндов: соответственно 8, 16, 32 `->` 32
- Операция: `mem([address]) -> rD` с расширением нулем

`LDRccSB (ARM4 и выше)`
`LDRccSH (ARM4 и выше)`

- Синтаксис операндов: `rD, [address]`
- Размер операндов: соответственно 8, 16 `->` 32
- Операция: `mem([address]) -> rD` со знаковым расширением

`STRccB`
`STRccH (ARM4 и выше)`
`STRcc`

- Синтаксис операндов: `rS, [address]`
- Размер операндов: соответственно 32 `->` 8, 16, 32
- Операция: `rS -> mem([address])`

Блочные инструкции пересылки позволяют переслать в память (из памяти) любое подмножество РОН (возможно все). Ассемблерная мнемоника для списка регистров `<registers_list>`:

`<registers>`
: заключенный в фигурные скобки список имен регистров через запятую

`<registers>^`
: то же, что предыдущий, но в инструкциях чтения `LDM`, которые загружают `pc` (`r15`), завершающий `^` означает, что регистр `CPSR` загружается из регистра `SPSR` (Saved Program Status Register) (в этот регистр копируется `CPSR` в момент возникновения прерывания)

В инструкциях блочной пересылки режимы адресации памяти отличны от описанных выше и задаются в поле кода инструкции:

LDMccIA (Increment After)
STMccIA (Increment After)

- Синтаксис операндов: rA, <registers_list> или rA!, <registers_list>
- Размер операндов в байтах: <block_size> = 4 * (число регистров в списке <registers_list>)
- Операция:
  - <start_address> = rA
  - <end_address> = rA + <block_size> - 4
  - для LDM: mem[<start_address>] -> <registers_list>; для STM: <registers_list> -> mem[<start_address>]
  - если присутствует суффикс '!' после rA и инструкция выполнена, то rA = rA + <block_size>

LDMccIBB(Increment Before)
STMccIB (Increment Before)

- Синтаксис операндов: rA, <registers_list> или rA!, <registers_list>
- Размер операндов в байтах: <block_size> = 4 * (число регистров в списке <registers_list>)
- Операция:
  - <start_address> = rA + 4
  - <end_address> = rA + <block_size>
  - для LDM: mem[<start_address>] -> <registers_list>; для STM: <registers_list> -> mem[<start_address>]
  - если присутствует суффикс '!' после rA и инструкция выполнена, то rA = rA + <block_size>

LDMccDA (Decrement After)
STMccDA (Decrement After)

- Синтаксис операндов: rA, <registers_list> или rA!, <registers_list>
- Размер операндов в байтах: <block_size> = 4 * (число регистров в списке <registers_list>)
- Операция:
  - <start_address> = rA - <block_size> + 4
  - <end_address> = rA
  - для LDM: mem[<start_address>] -> <registers_list>; для STM: <registers_list> -> mem[<start_address>]
  - если присутствует суффикс '!' после rA и инструкция выполнена, то rA = rA - <block_size>

LDMccDB (Decrement Before)
STMccDB (Decrement Before)

- Синтаксис операндов: `rA, <registers_list>` или `rA!, <registers_list>`
- Размер операндов в байтах: `<block_size> = 4 * (число регистров в списке <registers_list>)`
- Операция:
  - `<start_address> = rA - <block_size>`
  - `<end_address> = rA - 4`
  - для `LDM`: `mem[<start_address>] -> <registers_list>`; для `STM`: `<registers_list> -> mem[<start_address>]`
  - если присутствует суффикс '`!`' после `rA` и инструкция выполнена, то `rA = rA - <block_size>`

6.3.3.2 Целочисленные арифметические инструкции процессоров ARM

Все арифметические инструкции (включая логические и сдвига) имеют бит (бит 20), при установке которого по результату операции будут выставлены коды условия (биты `N`, `Z`, `C`, `V` в регистре `CPSR`). В ассемблерной мнемонике это отражается добавлением суффикса `S` к имени инструкции.

Все арифметические инструкции (включая логические и сдвига) используют понятие `<shifter_operand>`, который может иметь одну из следующих форм.

`#<immediate>`
: `<shifter_operand> = <immediate>`, где `<immediate>` – 32 бит константа, в которой только в каких-то 8-ми подряд идущих позициях могут быть не нули, номер первой позиции должен быть четным; кодируется в инструкции как 8-ми битная константа `<8_bit_immediate>` и 4-х битный сдвиг `<rotate_imm>`, при этом `<shifter_operand> = <8_bit_immediate> Rotate_Right (<rotate_imm> * 2)`

`rA`
: `<shifter_operand> = rA`

`rA, <shift> #<shift_imm>`
: `<shifter_operand> = rA <shift> #<shift_imm>`, где `<shift>` есть одна из операций

  `LSL (Logical Shift Left)`
  : сдвиг влево, вдвигаются нули

  `LSR (Logical Shift Right)`
  : сдвиг вправо, вдвигаются нули

  `ASR (Arithmetic Shift Right)`
  : сдвиг вправо, вдвигается знаковый разряд `rA`

  `ROR (ROtate Right)`
  : циклический сдвиг вправо

  `RRX (ROtate Right with eXtend)`
  : циклический сдвиг вправо 33-х битной величины `C, rA` на 1, правый операнд (т.е. `<shift_imm>`) должен отсутствовать

`#<shift_imm>` - 5-ти битная константа

`rA, <shift> rB`
: `<shifter_operand> = rA <shift> rB`, где `<shift>` есть одна из описанных выше операций, `#<shift_imm>` - 5-ти битная константа

Во всех инструкциях ниже, если присутствует суффикс `S`, то по результату операции выставляются коды условия.

`ADDcc`
`ADDccS`

- Синтаксис операндов: `rD, rS, <shifter_operand>`
- Размер операндов: 32
- Операция: `rS + <shifter_operand> -> rD`

`ADCcc`
`ADCccS`

- Синтаксис операндов: `rD, rS, <shifter_operand>`
- Размер операндов: 32
- Операция: `rS + <shifter_operand> + C -> rD`

`SUBcc`
`SUBccS`

- Синтаксис операндов: `rD, rS, <shifter_operand>`
- Размер операндов: 32
- Операция: `rS - <shifter_operand> -> rD`

`SBCcc`
`SBCccS`

- Синтаксис операндов: `rD, rS, <shifter_operand>`
- Размер операндов: 32
- Операция: `rS - <shifter_operand> - (~C) -> rD`

`RSBcc`
`RSBccS`

- Синтаксис операндов: `rD, rS, <shifter_operand>`
- Размер операндов: 32
- Операция: `<shifter_operand> - rS -> rD`

`RSCcc`
`RSCccS`

- Синтаксис операндов: `rD, rS, <shifter_operand>`
- Размер операндов: 32
- Операция: `<shifter_operand> - rS - (~C) -> rD`

`MULcc (ARM2 и выше)`
`MULccS (ARM2 и выше)`

- Синтаксис операндов: `rD, rA, rB`
- Размер операндов: 32
- Операция: `rA * rB -> rD` (получается младшая часть (32 бит) результата, знаковое и беззнаковое умножение)

`MLAcc (ARM2 и выше)`
`MLAccS (ARM2 и выше)`

- Синтаксис операндов: `rD, rA, rB, rC`
- Размер операндов: 32
- Операция: `rA * rB + rC -> rD` (получается младшая часть (32 бит) результата, знаковое и беззнаковое умножение)

`UMULLcc (ARM3M, ARM4 и выше)`
`UMULLccS (ARM3M, ARM4 и выше)`

- Синтаксис операндов: `rDlo, rDhi, rA, rB`
- Размер операндов: 32 -> 64
- Операция: `rA * rB -> (rDlo, rDhi)` (беззнаковое умножение)

`UMLALcc (ARM3M, ARM4 и выше)`
`UMLALccS (ARM3M, ARM4 и выше)`

- Синтаксис операндов: `rDlo, rDhi, rA, rB`
- Размер операндов: 32 -> 64
- Операция: `(rDlo, rDhi) + rA * rB -> (rDlo, rDhi)` (беззнаковое умножение)

`SMULLcc (ARM3M, ARM4 и выше)`
`SMULLccS (ARM3M, ARM4 и выше)`

- Синтаксис операндов: `rDlo, rDhi, rA, rB`
- Размер операндов: 32 -> 64
- Операция: `rA * rB -> (rDlo, rDhi)` (знаковое умножение)

`SMLALcc (ARM3M, ARM4 и выше)`
`SMLALccS (ARM3M, ARM4 и выше)`

- Синтаксис операндов: `rDlo, rDhi, rA, rB`
- Размер операндов: 32 -> 64
- Операция: `(rDlo, rDhi) + rA * rB -> (rDlo, rDhi)` (беззнаковое умножение)

`MOVcc`
`MOVccS`

- Синтаксис операндов: `rD, <shifter_operand>`
- Размер операндов: 32
- Операция: `<shifter_operand> -> rD`

`MVNcc`
`MVNccS`

- Синтаксис операндов: `rD, <shifter_operand>`
- Размер операндов: 32
- Операция: `~<shifter_operand> -> rD`

`CMPcc`

- Синтаксис операндов: `rD, <shifter_operand>`
- Размер операндов: 32
- Операция: `rD - <shifter_operand>` -> коды условия

`CMNcc`

- Синтаксис операндов: `rD, <shifter_operand>`
- Размер операндов: 32
- Операция: `rD + <shifter_operand>` -> коды условия

`TSTcc`

- Синтаксис операндов: `rD, <shifter_operand>`
- Размер операндов: 32
- Операция: `rD & <shifter_operand>` -> коды условия

`TEQcc`

- Синтаксис операндов: `rD, <shifter_operand>`
- Размер операндов: 32
- Операция: `rD ^ <shifter_operand>` -> коды условия

6.3.3.3 Логические инструкции процессоров ARM

Во всех инструкциях ниже, если присутствует суффикс `S`, то по результату операции выставляются коды условия.

`ANDcc`
`ANDccS`

- Синтаксис операндов: `rD, rS, <shifter_operand>`
- Размер операндов: 32
- Операция: `rS & <shifter_operand> -> rD`

`EORcc`
`EORccS`

- Синтаксис операндов: `rD, rS, <shifter_operand>`
- Размер операндов: 32
- Операция: `rS ^ <shifter_operand> -> rD`

`ORRcc`
`ORRccS`

- Синтаксис операндов: `rD, rS, <shifter_operand>`
- Размер операндов: 32
- Операция: `rS | <shifter_operand> -> rD`

`BICcc`
`BICccS`

- Синтаксис операндов: `rD, rS, <shifter_operand>`
- Размер операндов: 32
- Операция: `rS & ~<shifter_operand> -> rD`

6.3.3.4 Инструкции сдвига процессоров ARM

Инструкции сдвига как отдельные команды отсутствуют. Вместо этого используются возможности формирования `<shifter_operand>` и описанная выше инструкция `MOV`.

6.3.3.5 Инструкции управления процессоров ARM

Поскольку запись в регистр `pc` (`r15`) вызывает переход по записанному адресу, то любая инструкция, результатом которой является регистр `pc` (т.е. `rD=r15` в описанных выше командах), является командой условного (так как все инструкции условные) перехода. Например, инструкция `MOVcc r15, rS` является командой условного перехода по абсолютному адресу в регистре `rS`, а инструкция `ADDcc r15, r15, rS` – командой условного перехода по относительному адресу в регистре `rS`.

`Bcc`

- Синтаксис операндов: `<target_address>`
- Размер операндов: `<target_address>` - 24 бит (умножается на 4)
- Операция: `PC + offset(<target_address>) -> PC`

`BLcc`

- Синтаксис операндов: `<target_address>`
- Размер операндов: `<target_address>` - 24 бит (умножается на 4)
- Операция: `PC -> r14`, `PC + offset(<target_address>) -> PC`

`BXcc (ARM4 и выше)`

- Синтаксис операндов: `rS`
- Размер операндов: 32
- Операция: `rS -> PC`, по младшему биту `rS` определяется, надо ли переключать процессор в `THUMB` режим

6.3.3.6 Мультипроцессорные инструкции процессоров ARM

Для работы с семафорами предназначены следующие инструкции:

`SWPccB (ARM3 и выше)`
`SWPcc (ARM3 и выше)`

- Синтаксис операндов: `rD, rS, [rA]`
- Размер операндов: соответственно 8, 32
- Операция: `mem(rA) -> rD`, `rS -> mem(rA)`

6.3.3.7 Упрощенная ассемблерная мнемоника процессоров ARM

Приведем некоторые упрощенные ассемблерные инструкции, которые используются для удобства чтения ассемблерных текстов. Эти инструкции являются сокращениями для описанных ранее инструкций ARM.

Инструкции блочной пересылки имеют синонимы по имени типа стека.

`LDMccFD (Full Descending)`
: синоним для `LDMccIA`

`STMccEA (Empty Ascending)`
: синоним для `STMccIA`

`LDMccED (Empty Descending)`
: синоним для `LDMccIB`

`STMccFA (Full Ascending)`
: синоним для `STMccIB`

`LDMccFA (Full Ascending)`
: синоним для `LDMccDA`

`STMccED (Empty Descending)`
: синоним для `STMccIB`

`LDMccEA (Empty Ascending)`
: синоним для `LDMccIA`

`STMccFD (Full Descending)`
: синоним для `STMccIA`

6.4 Примеры кода для семейства ARM

Мы используем ассемблерный синтаксис и соглашения о вызовах операционной системы RISCiX. Основные соглашения:

`Stack Pointer (SP)`
: = регистр `r13`

`Frame Pointer (FP)`
: = регистр `r11`

`Адрес возврата из функции` **(диктуется системой команд)**
: = регистр `r14`

`Возвращаемое значение функции`
: = регистр `r0`

`Аргументы функции`
: = регистры `r0`, ..., `r3`; если аргументов больше, чем 4, то последующие размещаются в `mem(SP)`, `mem(SP+4)`, ..., здесь `SP` - в точке входа в функцию

`Регистры, не сохраняемые при вызове функции`
: = `r0`, ..., `r3`, `r12`, `r14`

`Регистры, сохраняемые при вызове функции`
: = `r4`, ..., `r11`, `r13`

Используются следующие синонимы для регистров:

`fp` = `r11`

`ip` = `r12`

`sp` = `r13`

`lr` = `r14`

`pc` = `r15`

Мы используем те же примеры исходного кода, что и для процессоров Motorola 68xxx.

1.
```
        _f:
                mov     ip, sp
                stmfd   sp!, {fp, ip, lr, pc}
                mov     r0, #0
                sub     fp, ip, #4
                ldmea   fp, {fp, sp, pc}^
```
2.
```
        LC0:
                .word   _a
        _f:
                mov     ip, sp
                stmfd   sp!, {fp, ip, lr, pc}
                ldr     r3, [pc, #LC0 - . - 8]
                sub     fp, ip, #4
                ldr     r0, [r3, #0]
                ldmea   fp, {fp, sp, pc}^
```
3.
```
        LC0:
                .word   _a
        _f:
                mov     ip, sp
                stmfd   sp!, {fp, ip, lr, pc}
                ldr     r0, [pc, #LC0 - . - 8]
                sub     fp, ip, #4
                ldmea   fp, {fp, sp, pc}^
```

4.

```
_f:
        mov     ip, sp
        stmfd   sp!, {fp, ip, lr, pc}
        sub     fp, ip, #4
        ldmea   fp, {fp, sp, pc}^
_g:
        mov     ip, sp
        stmfd   sp!, {fp, ip, lr, pc}
        mov     r0, #1
        sub     fp, ip, #4
        bl      _f
        ldmea   fp, {fp, sp, pc}^
```

5.

```
_f:
        mov     ip, sp
        stmfd   sp!, {fp, ip, lr, pc}
        add     r0, r0, r1
        sub     fp, ip, #4
        ldmea   fp, {fp, sp, pc}^
LC0:
        .word   _a
_g:
        mov     ip, sp
        stmfd   sp!, {fp, ip, lr, pc}
        mov     r0, #1
        ldr     r3, [pc, #LC0 - . - 8]
        sub     fp, ip, #4
        ldr     r1, [r3, #0]
        bl      _f
        ldmea   fp, {fp, sp, pc}^
```

6.

```
_f:
        mov     ip, sp
        stmfd   sp!, {fp, ip, lr, pc}
        sub     fp, ip, #4
        mov     ip, r0
        mov     r0, #0
        mov     r2, r0
        cmp     r0, r1
        ldmgeea fp, {fp, sp, pc}^
L5:
        ldr     r3, [ip, r2, asl #2]
        add     r0, r0, r3
        add     r2, r2, #1
        cmp     r2, r1
        blt     L5
        ldmea   fp, {fp, sp, pc}^
```

```
LC0:
        .word   _n
LC1:
        .word   _a
_g:
        mov     ip, sp
        stmfd   sp!, {fp, ip, lr, pc}
        ldr     r0, [pc, #LC1 - . - 8]
        ldr     r3, [pc, #LC0 - . - 8]
        sub     fp, ip, #4
        ldr     r1, [r3, #0]
        bl      _f
        ldmea   fp, {fp, sp, pc}^
```

7.

```
_f:
        mov     ip, sp
        stmfd   sp!, {r4, fp, ip, lr, pc}
        sub     fp, ip, #4
        mov     r4, r0
        mov     lr, r2
        mov     r0, #0
        mov     ip, r0
        cmp     r0, lr
        ldmgeea fp, {r4, fp, sp, pc}^
L5:
        ldr     r2, [r4, ip, asl #2]
        ldr     r3, [r1, ip, asl #2]
        mla     r0, r3, r2, r0
        add     ip, ip, #1
        cmp     ip, lr
        blt     L5
        ldmea   fp, {r4, fp, sp, pc}^
```

8.

```
_f:
        mov     ip, sp
        stmfd   sp!, {fp, ip, lr, pc}
        cmp     r0, r2
        cmpge   r0, r1
        sub     fp, ip, #4
        ldmgeea fp, {fp, sp, pc}^
        cmp     r1, r2
        movge   r0, r1
        movlt   r0, r2
        ldmea   fp, {fp, sp, pc}^
```

Список описанных понятий

О

П

Р

С

Т

У

Ц

Список описанных терминов

A

B

C

D

E

F

G

I

J

L

M

N

O

P

Q

R

S

T

U

V

W

X

Y

Краткое содержание

Содержание