ВВЕДЕНИЕ

Термин «архитектура» трактуется самым различным образом и вообще–то не имеет общепризнанного определения. Воспользуемся определением, данным в «Толковом словаре по вычислительным системам»:

Архитектура – это описание цифровой вычислительной системы на некотором общем уровне, включая описание пользовательских возможностей программирования, системы команд и средств пользовательского интерфейса, организации памяти и системы адресации, операций ввода–вывода и управления и т.д.

Вообще говоря, можно предложить несколько уровней описания системы и способов деления на уровни, причем различной степени детализации.

Цифровой компьютер – это машина, которая может решать задачи, выполняя данные ей команды. Последовательность команд, описывающих решение определенной задачи, называется программой. Электронные схемы каждого компьютера могут распознавать и выполнять ограниченный набор простых команд. Поэтому все программы перед выполнением должны быть превращены в последовательность таких команд, которые обычно не сложнее чем

- Сложить два числа;
- Проверить, не является ли число нулем;
- Скопировать кусок данных из одной части памяти в другую.

Это примитивные команды и являются тем языком, которым человек общается с компьютером. Такой язык называется машинным. Обычно разработчик старается сделать машинные команды как можно проще, что бы избежать дополнительных сложностей при конструировании компьютера, снизить затраты на электронику. Поэтому большинство машинных языков очень примитивны и их использование трудно и утомительно.

Это привело к тому, что с течением времени появились ряд уровней абстракций, каждая из которых настраивается над абстракцией более низкого уровня. Такой поход называется многоуровневой компьютерной организацией. Такой подход в рассмотрении архитектуры компьютера принят в классической книге «Архитектура компьютера» Эндрью Таненбаума, ведущего специалиста в области разработки компьютеров из МТИ.

В.1 МНОГОУРОВНЕВАЯ КОМПЬЮТЕРНАЯ ОРГАНИЗАЦИЯ

Существует разница между тем, что удобно людям и тем, что удобно для компьютера (или что удобно программисту и что удобно разработчику электроники). В литературе это получило название семантического разрыва. Семантический разрыв определяет различие принципов, лежащих в основе языков программирования высокого уровня и тех принципов, которые определяют архитектуру ЭВМ. В этом курсе мы рассмотрим, каким образом можно решить эти проблемы.

Языки, уровни и виртуальные машины

Обозначим через Я1 – язык программирования, удобный для человека. Машинные команды также образуют некоторый язык, понятный компьютеру, который обозначим как Я0. Для того, чтобы перевести программы с языка Я1 на понятный компьютеру язык Я0 можно использовать следующие подходы.

Первый способ – замена каждой команды, написанной на языке Я1 эквивалентным набором команд, написанных на языке Я0. В этом случае компьютер выполняет новую программу, написанную на языке Я0 вместо старой, записанной на языке Я1. Такая технология называется трансляцией.

Второй способ – написание программы на языке Я0, которая берет команды, написанные на языке Я1 в качестве исходных данных, рассматривает каждую команду по очереди и сразу выполняет эквивалентный набор команд языка Я0. Эта технология не требует составления новой программы на языке Я0, Эта технология называется интерпретацией, а программа, осуществляющая интерпретацию называется интерпретатором.

Различия между трансляцией и интерпретацией заключается в том, что при трансляции все программа Я1 переделывается в программу Я0 и программа Я1 может быть отброшена, а в память компьютера загружается программа Я0. При интерпретации каждая команда перекодируется и тут же выполняется.

На практике проще представить себе существование некоторого гипотетического компьютера, или виртуальной машины, для которой машинным языком является удобный для нас язык Я1, чем думать о трансляции и интерпретации. Назовет такую машину М1, а машину, понимающую язык Я0, машиной М0.

Хотелось бы сконструировать машину, для которой машинным языком был бы язык Я1, однако эта машина была бы слишком дорогой.

Чтобы трансляция и интерпретация были бы целесообразны, необходимо, чтобы языки Я0 и Я1 не очень отличались друг от друга. Казалось бы, это требование находится в некотором противоречии с целью создания языка Я1, удобного для человека. Однако это противоречие разрешается следующим очевидным образом.

Создается следующий набор команд Я2, который в большей степени ориентирован на человека. Этому языку соответствует виртуальная машина

М2, соответствующие трансляторы или интерпретаторы Я2-Я1 и т.д. Изобретение ряда языков, каждый из которых более удобен для человека, чем предыдущий может продолжаться до тех пор, пока мы не дойдем до подходящего. Каждый последующий язык рассматривает предыдущий как основу и поэтому можно представить компьютер в виде уровней (см. рис. В1).

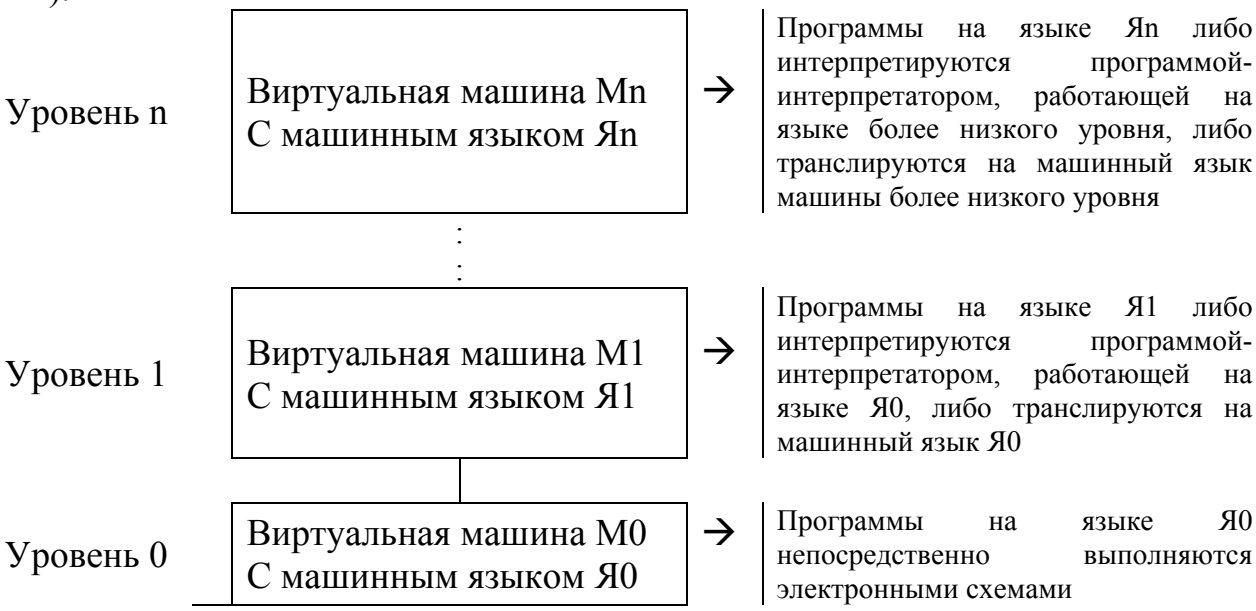
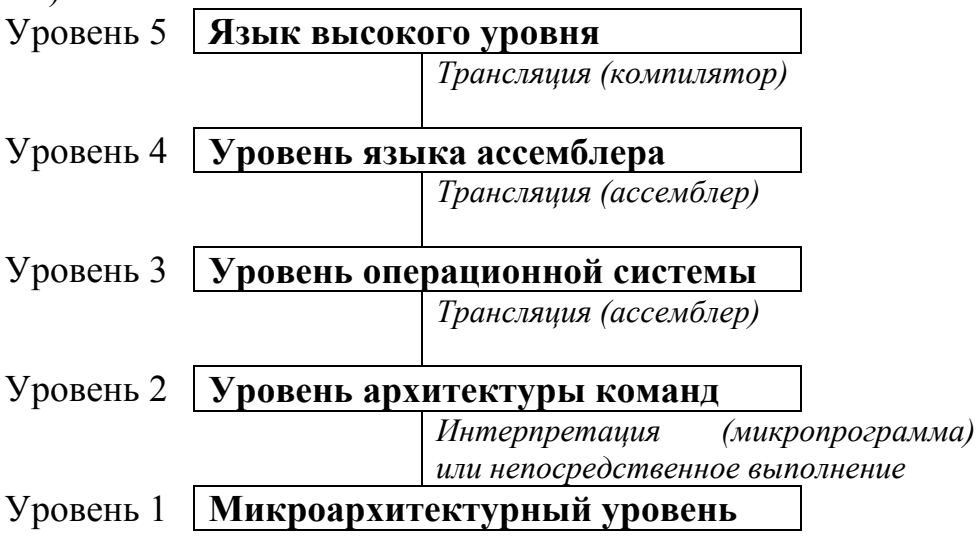


Рис. В1.

Человеку, пишущему программы на языке Яn не обязательно знать об интерпретаторах и трансляторах более низких уровней. Большинство пользователей, использующих машину уровня n интересуются только этим уровнем. Однако для изучения работы компьютера необходимо изучить все уровни.

Современные многоуровневые машины

Деление на уровни является в некотором роде условным. Можно говорить о шести уровнях представления современного компьютера. (рис. В2).



Аппаратное обеспечение

Уровень 0 **Цифровой логический уровень**

На рисунке отсутствует уровень физических устройств, расположенный ниже уровня 0. Это электронные схемы и не являются предметом рассмотрения курса.

На самом нижнем уровне – *цифровом логическом* – объекты состоят из вентилей и хотя вентили являются электронными схемами, их работа может быть описана как цифровые средства. Вентили образуют бит памяти, которые в свою очередь, образуют регистры.

Следующий уровень – *микроархитектурный.* На этом уровне рассматриваются совокупности регистров, которые образуют локальную (или регистровую) память и арифметическо-логическое устройство (АЛУ), предназначенное для выполнения простых операций. Регистры вместе с АЛУ образуют *тракт данных*. Основная операция состоит в следующем: выбирается один или два регистра, АЛУ производит над ними некоторую операцию (например, сложение), результат помещается в один из регистров. В некоторых машинах тракт данных контролируется *микропрограммой*. На других контроль осуществляется аппаратными средствами.

Второй уровень – *уровень архитектуры системы команд.* Руководства по машинному языку, выпускаемые фирмами-производителями содержат информацию именно об этом уровне. Когда там описывается система команд, то описываются команды, выполняемые программой-интерпретатором или аппаратными средствами.

Следующий третий уровень – *уровень операционной системы*-- обычно гибридный. Большинство команд этого уровня имеется также и на уровне архитектуры системы команд (команды, имеющиеся на одном уровне, также могут иметься и на других). Особенности уровня: новые команды, иная организация памяти, способность выполнять две и более программ одновременно и некоторые другие. Команды третьего уровня, идентичные командам второго, выполняются микропрограммой или аппаратными средствами, а не операционной системой. Часть команд (не имеющихся на втором уровне), выполняются операционной системой.

Нижние уровни конструируются не для того, что бы с ними работал обычный программист. Они предназначены для работы интерпретаторов и трансляторов и поддерживаются системными программистами. Уровни с четвертого и выше предназначены для работы прикладных программистов. Следующая особенность: уровни 2 и 3 обычно интерпретируются, а уровни 4 и выше как правило, поддерживаются транслятором. Уровни 1, 2 и 3 – цифровые, т.е. программы, написанные на языках этих уровней, состоят из наборов цифр. Начиная с четвертого уровня, языки содержат слова и сокращения, понятные человеку.

Четвертый уровень представляет собой символьную запись языка более низкого уровня. Программа, которая выполняет трансляцию называется *ассемблером .*

Пятый уровень обычно состоит из языков, разработанных для прикладных программистов. Такие языки называются языками высокого уровня.

Выводы: компьютер проектируется как иерархическая структура уровней, каждый из которых надстраиваbется над предыдущим. Каждый уровень представляет собой абстракцию с различными объектами и операциями.

Набор типов данных, операций и особенностей каждого уровня называется ***архитектурой.*** Архитектура связана с аспектами, которые видны программисту. Аспекты разработки, технологии и т.д. не являются частью архитектуры. Термины компьютерная архитектура и компьютерная организация в сущности означают одно и то же.

Развитие многоуровневых машин

Аппаратное обеспечение состоит из электронных схем, памяти, устройств ввода-вывода, т.е. из осязаемых объектов.

Программное обеспечение состоит из алгоритмов и программ. В первых вычислительных машинах разница между программными и аппаратными средствами была очевидной. Со временем эта грань стала размываться. Сейчас можно говорить о том, что аппаратное и программное обеспечения логически эквивалентны. Карен Панетта Ленц говорил: «Аппаратное обеспечение – это всего лишь окаменевшее программное обеспечение». Разделение функций программного и аппаратного обеспечения определяется такими факторами как стоимость, скорость, надежность, а также частота ожидаемых изменений.

Изобретение микропрограммирования

У первых компьютеров было только два уровня уровень архитектуры набора команд и цифровой логический уровень.

В 1951 году Морис Уилкс (Кембриджский университет) предложил идею трехуровневого компьютера. Такой компьютер должен иметь встроенный неизменяемый интерпретатор (микропрограмму), функция которого заключалась в выполнении программ посредством интерпретатора. Таким образом, аппаратное обеспечение должно было выполнять только микропрограммы с ограниченным набором команд. Электронные схемы существенно упростились, цена уменьшилась, а надежность возросла. К 70-м годам идея микропрограммирования стала преобладающей.

Изобретение операционной системы

Первые операционные системы появились в 60-е годы. Придумана она была для того, чтобы автоматизировать работу оператора. Однако создание операционной системы было первым шагом на пути в развитии новой виртуальной машины. К уровню архитектуры команд добавлялись новые команды и в итоге сформировался новый уровень. Некоторые команды нового уровня были идентичны командам предыдущего, но появились и новые команды, которые полностью отличались. Эти команды тогда назывались *макросами ОС* или *вызовами супервизора*. Сейчас используют термин *системный вызов*. Первые операционные системы были ориентированы для работы в *пакетном режиме*. В начале 60-х годов в МТИ разработали операционную систему, которая позволяла одновременно работать нескольким пользователям. В такой системе ресурсы центрального процессора разделялись между несколькими пользователями и такие системы назывались и сейчас называются системами с *разделением времени*.

Перемещение функциональности системы на уровень микрокоманд

С 1970 г. микропрограммирование стало обычным, производители вводили новые команды путем расширения микропрограммы, т.е. программными методами. Многие новые команды не представляли особой ценности, т.к. эти же действия можно было сделать уже имеющимися средствами. Однако новые команды могли выполнять операции быстрее или предоставляли дополнительные удобства пользователю. Например:

- Ускорение работы с массивами (индексная и косвенная адресация)
- Системы прерывания, которые дают команду процессору, как только закончилась операция ввода или вывода;
- Способность приостановить одну программу и начать другую, используя небольшое количество команд (переключение процесса).

Устранение микропрограммирования

В 60-х – 70-х годах количество микропрограмм увеличивалось, но они работали все медленнее и медленнее, т.к. требовали значительного объема памяти. Пришло понимание того, что с устранением микропрограмм резко сократиться количество команд и компьютеры станут работать быстрее. Таким образом, компьютеры вернулись к тому состоянию, в котором они были до изобретения микропрограммирования.

Вывод: граница между аппаратным и программным обеспечением постоянно перемещается. Так же обстоит дело с уровнями – между ними нет четких границ.

В.2 РАЗВИТИЕ КОМПЬЮТЕРНОЙ АРХИТЕКТУРЫ

Нулевое поколение – механические компьютеры (1642-1945)

Первую счетную машину создал французский ученый Блез Паскаль в 1642 году. Ему тогда было 19 лет, он создал машину для своего отца, сборщика налогов. Машина могла выполнять только операции сложения и вычитания.

Тридцать лет назад великий немецкий математик Готфрид Вильгейм Лейбниц (1646-1716) построил счетную машину, которая помимо операций сложения и вычитания могла выполнять операции умножения и деления.

Еще через 150 лет профессор математики Кембриджского университета Чарльз Бебидж (1792-1871) (изобретатель спидометра) разработал и сконструировал разностную машину, предназначенную для подсчета таблиц чисел морской навигации. Машина могла выполнять только один алгоритм – метод конечных разностей с использованием полиномов. Машина, выполнявшая только один алгоритм, Бэбиджу вскоре наскучила и он начал разрабатывать (и потратил на это очень много средств) аналитическую машину. У аналитической машины было запоминающее устройство, вычислительное устройство, устройство ввода (для перфокарт), устройство вывода (перфоратор и печатающее устройство). Машина могла выполнять разные задачи. Она считывала команды с перфокарт и выполняла их. Поскольку аналитическая машина программировалась на ассемблере, то ей было необходимо программное обеспечение. Первый программист – дочь поэта Байрона Ада Ловлейс. В ее честь назван язык программирования АДА. Аналитическая машина была механической. Идеи Бэбиджа опередили эпоху в том смысле, что технологическая база не позволяла создавать устройства такой сложности с приемлемой надежностью.

В конце 30-х -- начале 40-х годов XX века счетные машины были сконструированы в Германии и Америке, в которых были использованы электромагнитные реле. Машина Атанасова была очень развита для своего времени. В ней использовалась бинарная арифметика, информационные емкости, которые периодически обновлялись. К сожалению, эта машина так и не заработала.

Айкен, опираясь на исследования Бэбиджа, решил создать такой же компьютер, но на основе реле. Работа над первым компьютером была закончена в 1944 году. Называлась машина «MARK 1». Затем началась эра электроники.

Первое поколение – электронные лампы (1945-1955)

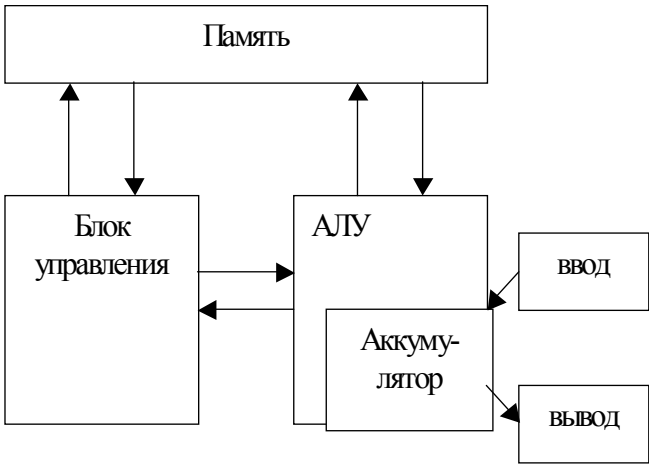
Стимулом для разработки электронного компьютера стала Вторая мировая война. Машина создавалась для шифровки и дешифровки в Великобритании. Одним из создателей этой машины был Алан Тьюринг.

В Америке Джон Моушли со своим студентом Дж. Преспером Экертом начали конструировать компьютер, предназначенный в первую очередь для

составления таблиц для нацеливания тяжелой артиллерии. К моменту завершения разработки война закончилась, машина стала не нужна для военных целей, и разработчикам было разрешено организовать школу, где они рассказывали о своей работе. Эта машина – ENIAC. Патент на ЦВМ они не получили, т.к. приоритет был отдан Атанасову.

В это же время в Институт специальных исследований в Принстоне приехал один из участников проекта ENIAC Джон фон Нейман, чтобы сконструировать свою версию компьютера.

Он предложил размещать программу вместе с данными в оперативной памяти и использовать бинарную арифметику. Основной проект известен теперь как фон-неймановская вычислительная машина. Он был использован в машине EDSAC. Практически все современные компьютеры являются фон-неймановскими машинами. Схема архитектуры этой машины приведена на рисунке В3. Машина не имела операций с плавающей точкой. Нейман полагал, что любой сведущий математик способен держать плавающую точку в голове.



Приблизительно в это же время в МТИ был создан компьютер Whirlwind-1. Особенности компьютера: слова небольшой длины (16 бит) и работа в РМВ. Он является прототипом мини-компьютера.

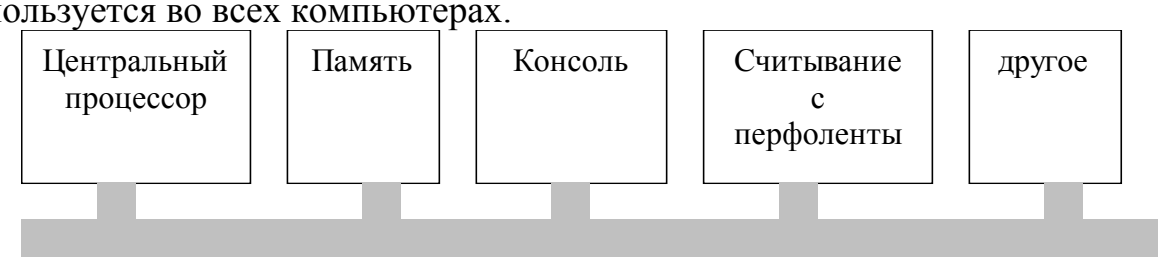
В 1953 году IBM создала свой первый компьютер IBM-704.

Второе поколение – транзисторы (1955-1965)

Транзисторы были изобретены в лаборатории Bell Джоном Бардином, Уолтером Браттейном и Уильямом Шокли, за что в 1956 году им была присуждена Нобелевская премия. Первый компьютер на транзисторах был построен в МТИ и назывался TX-1, а затем TX-2. Практического значения эти компьютеры не имели, но один из разработчиков, Кеннет Ольсен в 1957 году основал фирму DEC и произвели первую серийную машину на транзисторах PDP-1 (1961 г.). Эта была самая быстродействующая машина того времени. Время цикла – 5 микросекунд. Это в два раза меньше, чем у IBM-7090 (транзисторного аналога IBM-709). Стоил PDP-1 $120 000, а IBM – миллионы. Компания DEC продала десятки компьютеров PDP и так возникла компьютерная промышленность. Один из компьютеров был отдан в МТИ, где был создан первый графический дисплей, а студенты написали первую компьютерную игру – «Война миров».

Затем была создана машина PDP-8, которая была 12-разрядной, стоила $16 000, а главное нововведение – одна шина. *Шина* – это набор параллельно соединенных проводов для связи компонентов компьютера. Структура

компьютера с общей шиной приведена на рис. В4. Такая структура с тех пор используется во всех компьютерах.



В 964 году компания CDC выпустила машину CDC-6600, которая имела производительность на порядок выше, чем IBM-7090 и ее более дешевый аналог IBM-1401. Высокая производительность обеспечивалась за счет того, что внутри центрального процессора находилась машина с высокой степенью параллелизма. Разработчиком этого компьютера был Сеймур Крей. Он посвятил свою жизнь созданию мощных компьютеров, которые сейчас называются *суперкомпьютерами.* Это компьютеры CDC-6600, CDC-7600, Crey-1.

Разработчики упомянутых выше компьютеров занимались в первую очередь аппаратным обеспечением, стремясь повысить его надежность, быстродействие и снизить стоимость.

Следует отметить еще один проект – Burroughs B50000. Разработчики создавали компьютер с намерением программировать ее на языке Algol 60 (предшественник языка Pascal), сконструировав аппаратное обеспечение так,, что бы упростить работу компилятору. Так появилась идея, что программное обеспечение тоже надо учитывать при разработке компьютера.

Третье поколение – интегральные схемы (1965-1980)

В 1958 году была изобретена кремниевая технология (изобретатель – Роберт Нойс). Компьютеры на интегральных схемах были меньшего размера, работали быстрее, стоили дешевле. Наиболее значительные следующие.

К 1964 г. Фирма IBM лидировала на рынке, но выпускаемые ей компьютеры были программно несовместимы. Компания сделала решительный шаг. Она выпустила серию компьютеров на транзисторах System 360, которые были предназначены как для научных, так и для коммерческих расчетов. System 360 содержала много нововведений. Это было семейство компьютеров с одним и тем де ассемблером. Каждая новая модель была больше и мощнее предыдущей. Идея создания семейств компьютеров вскоре стала популярной и в течении нескольких лет большинство компьютерных компаний выпустило целые серии сходных машин.

Еще одно нововведение – *мультипрограммирование*. В памяти располагалось несколько программ и пока одна программа ждала окончания ввода-вывода, другая выполнялась.

Мир микрокомпьютеров сделал также большой шаг вперед вместе с производством компьютеров PDP-11. Во многих отношениях PDP-11 была младшим братом IBM 360 по организации компьютера и наличию в семействе машин разной стоимости и производительности.

Четвертое поколение – сверхбольшие интегральные схемы (1980-?)

Появление СБИС в 80-х годах позволило помещать на одну плату сначала десятки тысяч, а затем и миллионы транзисторов. К 80-м годам цены на компьютеры упали на столько, что приобретать компьютеры смогли не только организации, но и отдельные люди. Началась эра персональных компьютеров. Первые персональные компьютеры продавались в виде комплектов, как правило, на базе Intel 8080. Программное обеспечение пользователь писал сам. Затем появилась операционная система CP/M. Эта ОС помещалась на дискету, включала систему управления файлами и интерпретатор для выполнения пользовательских команд, которые набирались на клавиатуре.

Компания IBM, лидирующая в то время на рынке компьютеров, тоже решила заняться производством персоналок. Для ускорения процесса разработки копания IBM предоставила одному своему сотруднику, Филипу Эстриджу крупную сумму денег и отправила куда-нибудь подальше и не возвращаться, пока не будет создан персональный компьютер. Компьютер IBM PC появился в 1981 году и стал самым покупаемым в истории.

Но IBM вместо того, чтобы держать проект в секрете или защитить его патентами, она опубликовала полные проекты, включая электронные схемы. Многие компания тут же начали делать *клоны*, которые продавали дешевле. Из других компаний, производивших персональные компьютеры на базе своих процессоров, выжить удалось только некоторым, и то только потому, что они работали в узких областях.

Первая версия IBM PC была оснащена операционной системой MS-DOS, которую выпускала крошечная компания Microsoft. Эта компания разработала также собственную ОС Windows, которая работала на базе MS-DOS.

B.3 СЕМЕЙСТВА КОМПЬЮТЕРОВ

Pentium II

В 1968 году Роберт Нойс, изобретатель кремниевой интегральной схемы, Гордон Мур и Артур Рок, капиталист из Сан-Франциско, основали корпорацию Intel для производства микросхем. Сначала дела шли не очень хорошо. В 60-х годах калькуляторы были размером с принтер и весили 20 кГ.

В 1969 году японская фирма Buscom обратилась к компании Intel с просьбой выпустьть 12 несерийных микросхем. Інженер Тэд Хофф решил, что можно поместить 4-х битный универсальный процессор на одну

миросхему. Так в 1970 году появился первый процессор на одной микросхеме, процессор 4004. В 1972 году Intel выпустила 8-битный процессор 8008.

Новая микросхема вызвала большой интерес и Intel начала разработку новой микросхемы, у которой предел обращения к памяти в 16 Кбайт был преодолен. Так появился 8080, выпущенный в 1974 году. Этот процессор произвел революцию.

1978 год – процессор 8086. Он имел 16-разрядные внутренние регистры, 16-разрядные внутренние и внешние шины данных и мог адресовать до 1 Мб физической памяти. Быстродействие – 0,8 MIPS (миллион оп/с).

Затем появился 8088, с такой же архитектурой, как у 8086. Он имел шину не 16, а 8 бит, работал медленнее, но был дешевле. Когда фирма IBM выбрала 8088 для IBM PC, эта микросхема стала эталоном в производстве персональных компьютеров.

В начале 80-х годов Intel разработала 80286, совместимый с 8086. Он уже мог адресовать до 16 Мб физической памяти, имел специальный защищенный режим работы, который обеспечивал гибкий механизм адресации, управление доступом к памяти, управление привилегиями и т.п. Система команд также была расширена, а быстродействие достигало 2,7 MIPS.

Intel386 (1985) – это уже полностью 32-разрядный микропроцессор с 32-разряднымивнутренними регистрами и шинами данных. Его адресное пространство – 4 Гб. Добавлены страничный механизм и добавлен новый режим работы V86, который обеспечивал совместимость данного микропроцессора с программами, написанными для предыдущих устройств серии. Внутренняя архитектура микропроцессора Intel386 позволила организовать параллельное выполнение нескольких операций (выборка команд, декодирование, исполнение команд) что позволило увеличить быстродействие до 6 MIPS. Приблизительно в это же время была разработана новая операционная система Windows.

В 1989 г. выпущен микропроцессор Intel486, который содержал интегрированное устройство вычислений с плавающей точкой (FPU) и внутреннюю кэш-память для данных и команд.

В 1993 году был создан первый микропроцессор семейства P5 – Pentium, характеризующийся значительно более совершенной архитектурой. В этих процессорах устройство целочисленных вычислений имело два практически идентичных блока, в которых различные команды могли выполняться параллельно. Были доработаны модули, отвечающие за выборку и декодирование команд: была предусмотрена возможность прогнозировать действия процессора, путем осуществления предвыборку и предкодирование команд еще до получения результатов предыдущих команд. Переработано FPU с целью повышения его быстродействия. Были внесены некоторые изменения в архитектуру процессов: появилась поддержка объемных страниц, более гибким стал режим V86, введены дополнительные средства

внутренней самодиагностики. Быстродействие таких процессоров могло достигать 100 MIPS.

В 1993-1997 Intel продолжал работу по совершенствованию архитектуры своих процессоров. Был выпущен Pentium Pro – микропроцессор архитектуры P6, которая стала основной для более поздних микропроцессоров, вплоть до Pentium III. В этих процессорах: четыре модуля параллельно осуществляли выполнение команд, предвыборка дополнилась возможностями предсказания ветвлений, кэш-память стала двухуровневой, введена расширенная система обработки мультимедийной информации (MMX).

В настоящее время наиболее распространены Pentium II/III. Их быстродействие может достигать 1000 MIPS, объем оперативной памяти до 64 Гб, внутренней кэш-памяти – 2 Мб. Система обработки мультимедийной информации была усовершенствована: в микропроцессор Pentium III были введены 70 новых так называемых SIMD-команд.

Однако все эти возможности базируются на стандартной архитектуре x86.

UltraSPARC II

В 70-х годах в университетах очень популярна была ОС UNIX, но персональные компьютеры не подходили для этой системы. Энди Бехтольсхайм (аспирант из Стенфорда) разрешил эту проблему, самостоятельно построив рабочую станцию UNIX из стандартных компонентов, имеющихся в продаже и назвал ее SUN (сеть Стенфордского университета).

27-летний индиец Винод Косла предложил Энди Бехтольсхайму организовать компанию по производству рабочих станций SUN. Он нанял также аспиранта Скотта Мак-Нили и программиста Билла Джоя, главного создателя системы UNIX. Они организовали компанию Sun Microsystems. Первый компьютер был оснащен процессором Motorola 68020 и имел большой успех, так же, как и последующие модели. Они были мощнее персональных компьютеров и имели аппаратные и программные средства работы в сети ARPANET (предшественник Internet). В 1987 году компания решила разработать свой собственный процессор, основанном на революционном проекте Калифорнийского университета (RISC II). Этот процессор назывался SPARC (Scalable Processor ARCitecture – наращиваемая архитектура процессора).

В отличие от других компаний, Sun решила не производить процессоры, а предоставила патент нескольким компаниям. Так появились MicroSPARC, HyperSPARC, SuperSPARC, TurboSPARC.

Первый SPARC был 32-разрядным, а в 1995 году была разработана 64-разрядная версия. UltraSPARC с самого начала был предназначен для работы с изображениями, аудио, видео и мультимедиа вообще.

PicoJava II

Язык программирования С был придуман в лаборатории Bell Деннисом Ритчи и предназначался для UNIX. Затем Бьярн Строуструп добавил к С некоторые особенности из ООП и появился С++.

В 90-х годах решали задачу: как сделать так, чтобы пользователь могли вызывать программы через Internet и загружать их как часть web-страниц. Им нравился С++, но он не был надежным в том смысле, что программа, посланная на некоторый компьютер, могла причинить ему вред. Для решения задачи был придуман язык программирования Java – объектно-ориентированный язык программирования. Этот язык был создан для того, чтобы пересылать программы между компьютерами, и чтобы пользователю не надо было изменять их. Но если программа компилировалась на SPARC, а посылалась на Pentium, то запустить ее было нельзя.

Чтобы разрешить эту проблему, компания Sun придумала новую виртуальную машину – JVM. Был разработан компилятор, преобразующий программы на языке Java на язык JVM и интерпретаторы JVM для выполнения этих программ. Однако интерпретатор работает медленно, а использование соответствующего компилятора (JIT-компилятор) вызывает некоторую задержку между получением программы и ее выполнением.

Кроме программного обеспечения компания Sun разработала микросхемы JVM-процессоры, которые сами выполняют программы без какой-либо интерпретации. Это и есть PicoJava. Следует учесть, что PicoJava – это не микросхема, а проект, который является основой для ряда микросхем (например, Sun MicroJava 701), производящихся по патенту Sun. Этот процессор интересен тем, что он сильно отличается от Pentium, SPARC и имеет совершенно другую область применения.

*Актуальность архитектурных исследований. На одном из первых компьютеров мира EDSAC (1949 г., Кембридж) время такта 2 микросекунды ($2*10^{-6}$), можно было 2*n операций выполнить за 18*n миллисекунд, т.е. 100 операций в секунду. Вычислительный узел суперкомпьютера HP V2600 время такта составляет 1.8 наносекунды ($1.8*10^{-9}$ секунд), а пиковая производительность около 77 миллиардов арифметических операций в секунду. Таким образом производительность возросла в семьсот миллионов раз, а тактовая частота около 1000 раз*

2. ЦИФРОВОЙ ЛОГИЧЕСКИЙ УРОВЕНЬ

В самом низу иерархической схемы находится цифровой логический уровень, или аппаратное обеспечение компьютера. Основные элементы, из

которых конструируется цифровые компьютеры частично были рассмотрены в курсах «Дискретная математика» и «Физические основы ЭВМ». Поэтому в этом разделе некоторые компоненты будут просто упомянуты, а другие рассмотрены более подробно.

2.1. ВЕНТИЛИ И БУЛЕВА АЛГЕБРА

В курсе «Дискретная математика» были изучены вопросы математического описания и создания комбинационных схем на базе элементов И-ИЛИ-НЕ, а также представление любой логической функции в базисе Шеффера и Пирса.

2.2.ОСНОВНЫЕ ЦИФРОВЫЕ ЛОГИЧЕСКИЕ СХЕМЫ

Основные логические схемы рассмотрены в курсе «Физические основы ЭВМ».

Комбинационные схемы

Это цифровые устройства без памяти, т.е. комбинация входных сигналов определяет состояние выхода схемы. Комбинационные элементы:

- Мультиплексоры – устройство с 2^n входами данных, одним выходом и n управляющими входами, позволяющими выбрать один из входов данных;
- Декодер – из позиционного кода формирует унитарный;
- Компараторы – производят сравнение двух слов;
- Программируемые логические матрицы;

Арифметические схемы

Эти устройства используются для выполнения арифметических операций:

- Схемы сдвига;
- Сумматоры;
- Арифметико -- логические устройства;
- Тактовые генераторы;

2.3 ПАМЯТЬ

- Защелки и триггера—это один бит памяти;
- Регистры;
- Микросхемы памяти.

- ОЗУ и ПЗУ.

2.4. МИКРОСХЕБЫ ПРОЦЕССОРОВ И ШИНЫ

Микросхемы процессоров

Каждая микросхема процессора содержит набор выводов, через которые происходит обмен информацией с внешним миром. Выводы микросхем можно подразделить на три типа: адресные, информационные, управляющие.

Число адресных и информационных выводов – два ключевых параметра, определяющих производительность процессора. Обычно процессор имеет 16, 20, 32 или 64 – разрядный адрес и 8, 16, 32, 36 или 64 – разрядный информационный код.

Кроме адресных и информационных входов каждый процессор содержит выводы управления, а также питания, земли и синхронизирующего сигнала (меандр). Состав управляющих сигналов у различных процессоров может различаться, но их можно разделить на следующие основные категории:

- Управление шиной;
- Прерывание;
- Арбитраж шины;
- Состояние;
- Разное.

Схема типичного центрального процессора приведена на рис. 2.1.

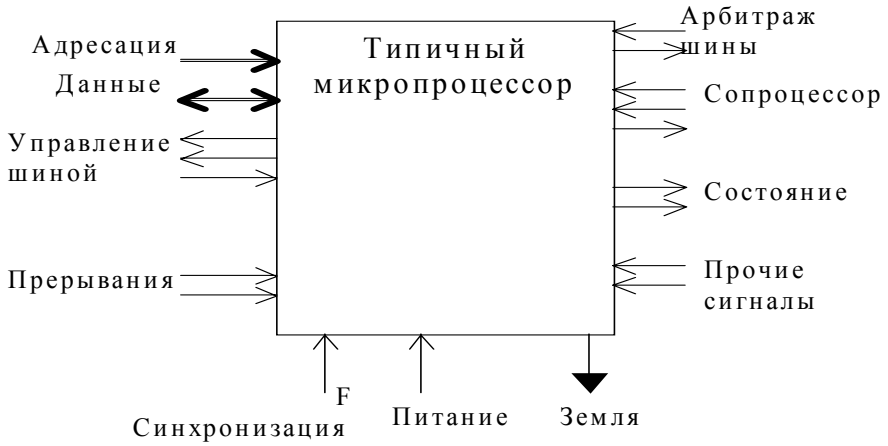


Рис. 2.1

Шины

Шина – это группа проводников, соединяющих различные устройства.

Первые персональные компьютеры имели одну внешнюю шину, которая называлась *системной.* Современные компьютеры содержат обычно содержат специальную шину между центральным процессором и памятью и

по крайней мере еще одну шину для устройств ввода вывода. Существуют четкие правила о том, как работает шина, и все устройства, связанные с шиной подчиняются этим правилам. Эти правила называются *протоколом шины.* Существуют ряд широко используемых шин.

Устройства, взаимодействующие с шиной могут быть активными и пассивными. Активное устройство может инициировать обращение к шине и называется *задающим.* Пассивное устройство ждут запроса и называются *подчиненными.* Одно и тоже устройство может выступать и как задающее и как подчиненное. Память – всегда подчиненное устройство. Разработка шин и принципы действия – достаточно сложные вопросы. Принципиальными вопросами является ширина шины, синхронизация шины, арбитраж и функционирование. Рассмотри эти параметры.

Ширина шины

Чем больше ширина шины, тем она лучше и тем она дороже. Пример увеличение ширины шины проиллюстрируем на примере процессора x86.

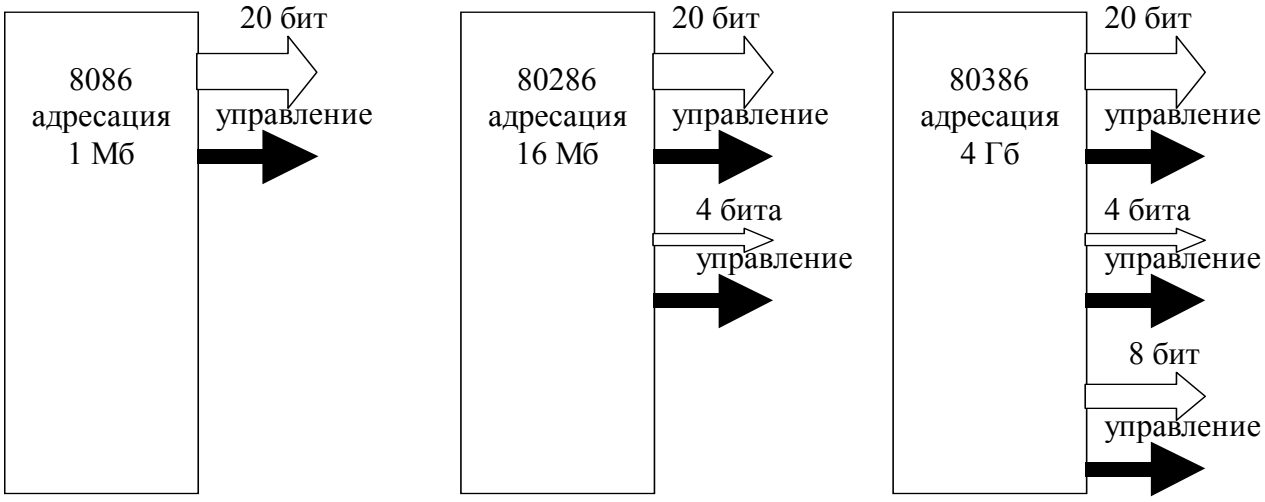


Рис. 2.2

Пропускную способность шины можно увеличить двумя способами: увеличением ширины шины и сокращением времени цикла. Увеличение скорости работы шины порождает проблему совместимости ее со старыми устройствами. Поэтому разработчики вынуждены использовать решения рис. 2.2. Иногда используют *мультиплексную* шину: в таких шинах нет разделения на адресные и информационные линии. Объединение линий дает сокращение ширины, но приводит к усложнению протокола обмена и снижению пропускной способности шины.

Синхронизация шины

Шины можно разделить на две категории: синхронные и несинхронные шины.

Синхронная шина содержит линию, которая запускается кварцевым генератором и любое действие шины занимает целое число так называемых *циклов шины*. Асинхронная шина не содержит кварцевого генератора и циклы шины могут быть любой требуемой длины.

Арбитраж шины

Механизмы арбитража шины могут быть централизованными и децентрализованными.

Пример централизованного арбитража приведен на рис. 2.3а. В данном случае один арбитр шины определяет, чья очередь следующая.

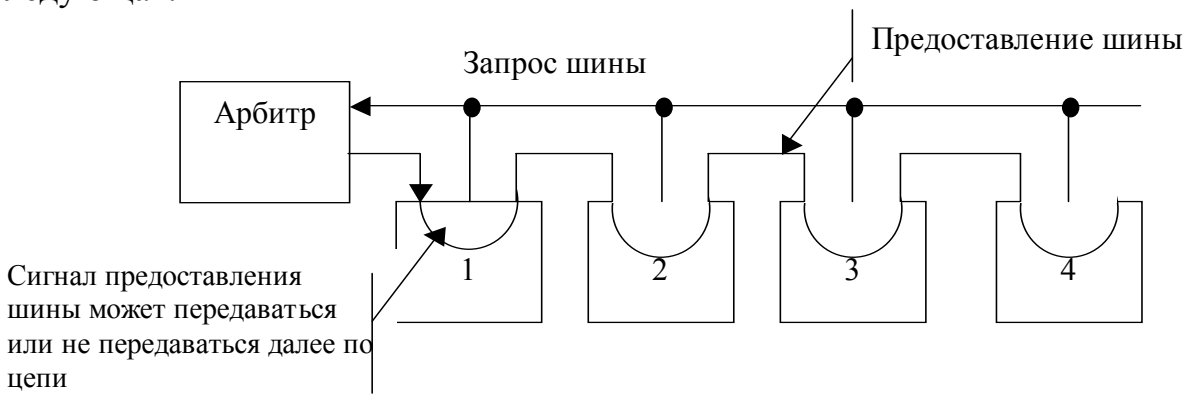


Рис. 2.3 а

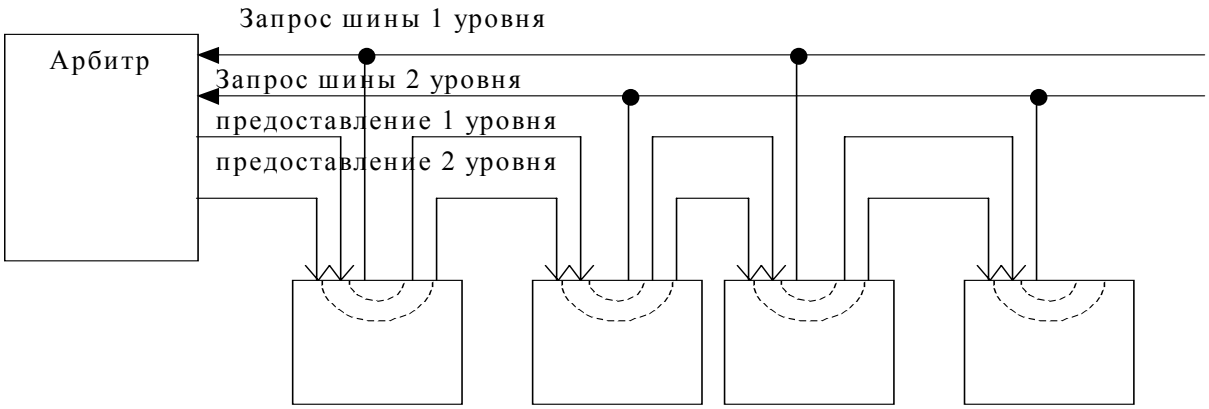


Рис. 2.3.б

Когда арбитр видит запрос шины, он запускает линию предоставления шины. Эта линия последовательно связывает все устройства ввода-вывода. Когда физически ближайшее к арбитру устройство воспринимает сигнал предоставления шины, оно проверяет, есть ли запрос шины. Если запрос существует, устройство пользуется шиной и не распространяет сигнал предоставления шиной дальше; в противном случае устройство передает сигнал предоставления шины следующему устройству. Передача сигнала предоставления шины продолжается до тех пор, пока какое-нибудь

устройство не воспользуется шиной. Такая система называется *системой последовательного опроса.* Приоритет устройства определяется тем, насколько близко к арбитру расположено устройство.

Чтобы обойти такую систему приоритетов, в некоторых шинах устанавливаются несколько уровней приоритетов. (рис. 2.3б). Среди устройств одного уровня приоритета используется система последовательного опроса.

Возможен также децентрализованный арбитраж шины. В таком случае каждое устройство перед обращением к шине анализирует ее состояние. Если шина свободна, то устройство ее монополизирует; если шина занята, то устройство сбрасывает сигнал запроса и спустя некоторое время повторяет попытку воспользоваться шиной.

2.5. ПРИМЕРЫ ЦЕНТРАЛЬНЫХ ПРОЦЕССОРОВ

Pentium II

С точки зрения программного обеспечения Pentium II представляет собой 32-разрядную машину. С точки зрения аппаратного обеспечения Pentium II представляет собой нечто большее. Он может обращаться к 64 Гбайт памяти, передавать данные блоками по 64 бита. На микроархитектурном уровне Pentium II представляет собой Pentium Pro с командами MMX. Команды вызываются из памяти заранее и разбиваются на микрооперации. Эти микрооперации хранятся в буфере, и как только одна из них получает необходимые ресурсы для выполнения, она может начаться. Если в одном цикле может начаться несколько микроопераций, Pentium II работает как суперскалярная машина.

Pentium II имеет двухуровневую КЭШ-память. КЭШ-память первого уровня имеет 16 Кбайт для команд и 16 Кбайт для данных, а смежная КЭШ-память второго уровня – 512 Кбайт.

С системах с процессором Pentium II используются две внешние шины, обе синхронные. Шина памяти используется для главному динамическому ОЗУ; шина PCI используется для общения с устройствами ввода-вывода. К шине PCI может подключаться *унаследованная* шина для подключения старых периферийных устройств.

Система Pentium II может содержать один или две центральных процессора. Существует специальная система поддержки совместного функционирования процессоров для устранения конфликтных ситуаций.

Pentium II существенно отличается от своих предшественников компоновкой. Микропроцессор Pentium II представляет собой SEC (Single Edge Cartridge – картридж с однорядным расположением контактов). Из 242 контактов картриджа 170 используются для сигналов, 27 для питания (с различной мощностью), 35 для “земли” и 10 – резерв на будущее.

Сигналы приведены на рис. 2.4.

При обозначении знак # используется для обозначения низкого уровня управления.

Рассмотрим основные типы сигналов.

Первая группа сигналов используется для запроса шины (арбитража). Сигнал BPRI# позволяет устройству с высоким приоритетом получить доступ к шине раньше других устройств. Сигнал LOCK# позволяет центральному процессору не предоставлять другим устройствам доступ к шине, пока работа не будет закончена.

Запрос на доступ к шине использует следующие сигналы. Адрес A# (36 бит, последние 3 равны 0). Все передачи состоят из 8 байт. Поэтому максимальный объем памяти 64 Гбайт.

Когда адрес передается на шину, устанавливается сигнал ADS#, который сообщает целевому объекту, что задействованы адресные линии. На линиях REQ# запускается цикл шины определенного типа (например, запись блока). Два сигнала четности нужны для проверки A#, а третий для проверки ADS# и REQ#. Подчиненное устройство использует пять специальных выводов для сообщения об ошибках.

Группа сигналов для отслеживания адресов, по которым происходило изменение данных используется в многопроцессорных системах.

Ответные сигналы передаются от подчиненного к задающему устройству. Сигнал RS# содержит код состояния. Сигнал TRDY# показывает, что подчиненное устройство готово принять данные. Эти сигналы также проверяются на четность.

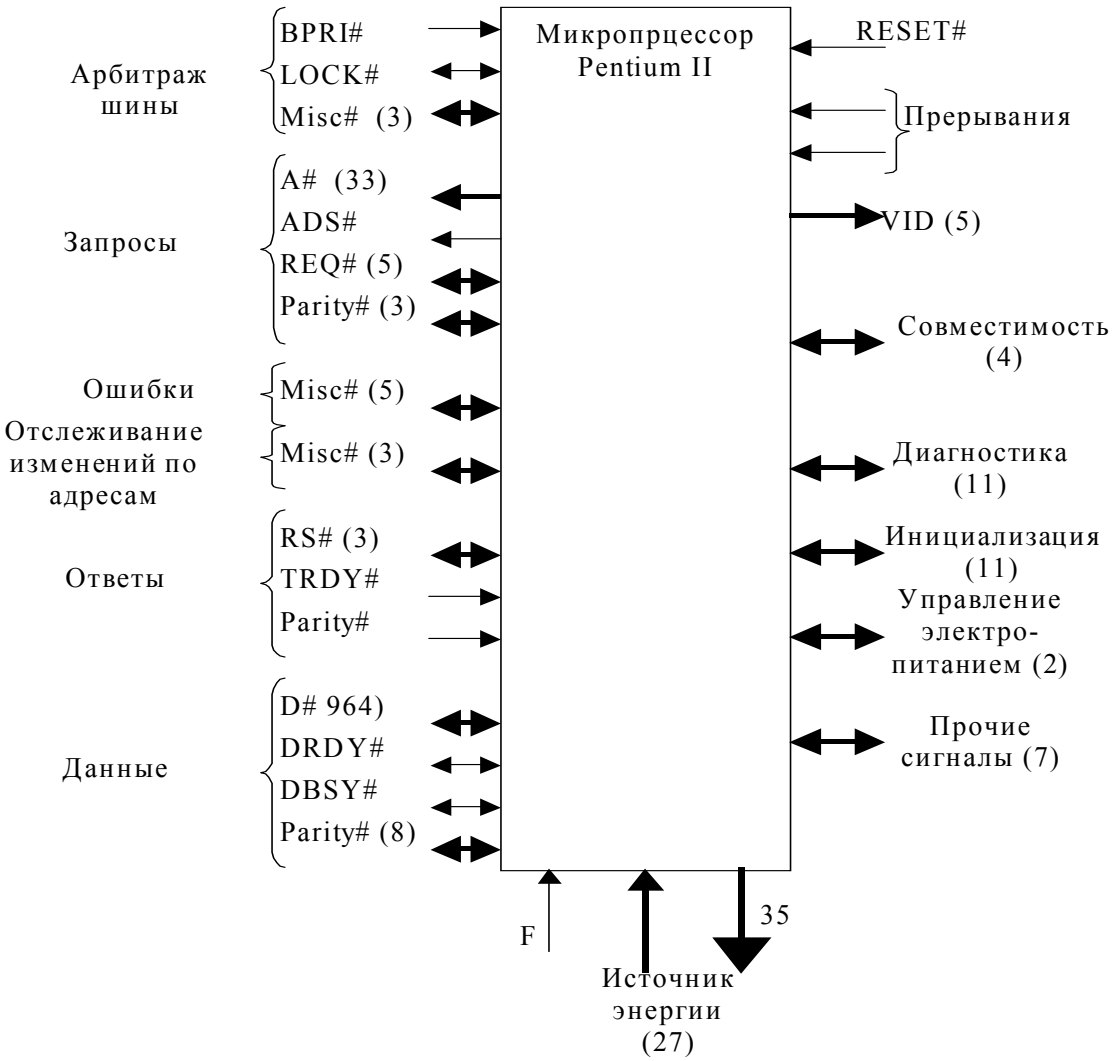


Рис. 2.4.

Последняя группа сигналов нужна для передачи данных. Сигнал D# используется, чтобы поместить 8 байт на шину. Когда они готовы, вырабатывается сигнал DRDY#; этот сигнал также сообщает, что шина занята.

Сигнал RESET# используется для перезагрузки процессора.

Сигналы VID используются для автоматического выбора напряжения источника питания.

Сигналы совместимости нужны для работы с устройствами более старых моделей.

Диагностика – для отладки; инициализация – для загрузки системы; управление электропитанием – «сон»; прочие – например, перегрев.

Конвейерный режим шины памяти процессора Pentium II. Поскольку процессор работает быстрее, чем ОЗУ, для повышения производительности системы шина памяти процессора работает в конвейерном режиме, при этом в шине одновременно 8 операций. Обращение процессора к памяти называются транзакциями и имеют 6 стадий:

1. Фаза арбитража шины -- определяется, какое из задающих устройств будет следующим.
2. Фаза запроса -- на шину передается адрес.
3. Фаза сообщения об ошибке – подчиненное устройство передает сигнал об ошибке четности в адресе или каких-либо других неполадках.
4. Фаза проверки на наличие нужного слова на другом процессоре – проверяется, нет ли конфликта с другим процессором (эта фаза нужна только в многопроцессорных системах).
5. Фаза ответа – задающее устройство узнает, где взять необходимые данные.
6. Фаза передачи данных – собственно передача данных.

Наличие всех фаз необязательно.

В системах с Pentium II на каждой стадии используются определенные сигналы, отличные от сигналов других стадий, поэтому каждая из них не зависит от остальных.

UltraSPARC II

Семейство UltraSPARC – это серия 64-разрядных процессоров SPARC, используемых в рабочих станциях и серверах Sun и некоторых других системах.
UltraSPARC II представляет собой машину типа RISC.

Процессор UltraSPARC II был разработан для создания 4-узловых мультипроцессоров с разделенной памятью без добавления внешних схем , а также для создания более крупных мультипроцессоров с минимальным дополнением схем. Поэтому в каждую микросхему UltraSPARC II были включены связующие элементы, необходимые для построения мультипроцессорных систем.

Микросхема содержит 5.4 млн. транзисторов. Микросхема имеет 787 выводов. Такое количество выводов объясняется наличием 64 разрядов для адресации 128 для данных, а также особенностями работы Кэш-памяти и наличием резерва.

Проц6ссор UltraSPARC II содержит два внутренних блока Кэш-памяти: 16 Кбайт для команд и 16 Кбайт для данных. Как и у Pentium II у UltraSPARC II вне кристалла процессора расположена Кэш-память второго уровня, но в отличии от Pentium II, она не упакована в один картридж с процессором, что дает возможность выбирать различные микросхемы Кэш-памяти второго уровня.

Большинство рабочих станций Sun содержат синхронную шину на 25 МГц, которая называется Sbus. К этой шине может подключаться память, однако, ее пропускная способность низкая, в процессоре UltraSPARC II соединение с памятью используется механизм *UPA (Ultra Port Architecture – высокоскоростной пакетный коммутатор)*. Ядро системы UltraSPARC II

приведено на рис. 2.4. и содержит: центральный процессор, интерфейс UPA и Кэш-память второго уровня (2 статических ЗУ).

На рисунке также изображена микросхема UDB II – буфер данных, функции которой мы опишем ниже. Когда процессору нужно слово из памяти, то он сначала обращается к Кэш-памяти первого уровня. Если он находит слово, то он продолжает выполнять операции с полной скоростью. Если слово отсутствует, то процессор обращается к Кэш-памяти второго уровня.

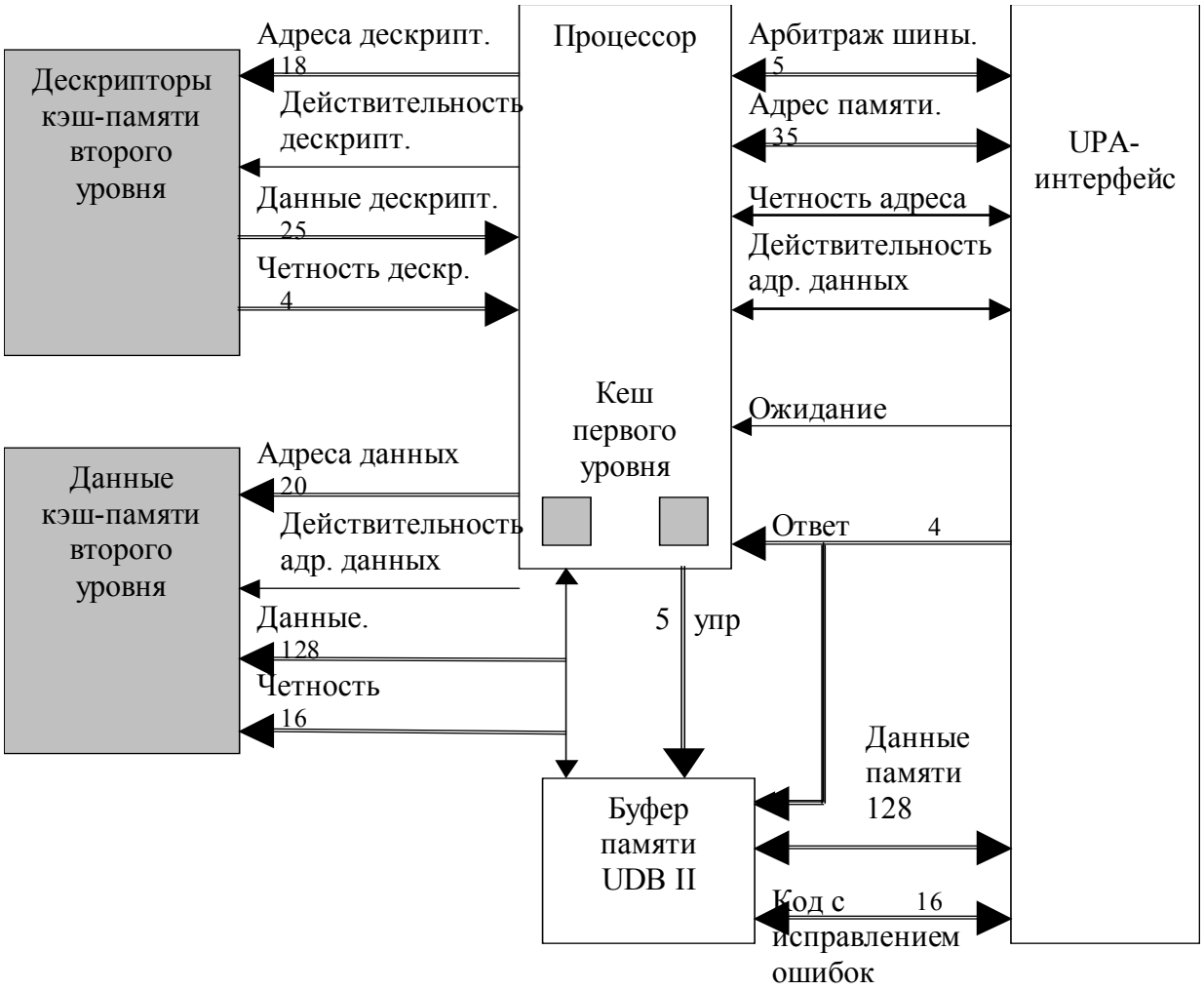


Рис. 2.5

В случае отсутствия нужной строки в Кэш-памяти первого уровня, центральный процессор посылает идентификатор строки, которую он ищет (адрес дескриптора) в Кэш-память второго уровня. Ответ (данные дескриптора) предоставляют процессору информацию о том, есть ли нужная строка в Кэш-памяти второго уровня. Если есть – центральный процессор получает ее. Передача данных осуществляется по 16 байтов, для пересылки строки требуется 4 цикла.

Если требуемой строки нет, ее нужно вызывать из основной памяти через интерфейс UPA, который управляется специальным контроллером.

Туда поступают адресные сигналы и сигналы управления от центрального процессора. Чтобы получить доступ к памяти, центральный процессор должен получить разрешение пользоваться шиной.

В ожидании результатов центральный процессор может заниматься другой работой, т.е. выполнять другие команды. Таким образом, сразу несколько транзакций к UPA могут ожидать выполнения. Система UPA может справляться с двумя независимыми потоками транзакций (обычно это чтение и запись), каждый поток проходит с несколькими задержками. Задача центрального контроллера – следить за процессами и производить обращения в память в наиболее рациональном порядке.

Данные из памяти могут поступать блоками по 8 байт. Они содержат 16-разрядный код с исправлением ошибок для повышения надежности. Все входные данные поступают в буфер UDB и хранятся там. Буфер необходим для того, чтобы центральный процессор и память работали асинхронно.

Приведенное описание работы процессора сильно упрощено, но отражает суть.

PicoJava II

Pentium II и UltraSparc II процессоры высокой производительности. Существуют и другие компьютеры – так называемые встроенные системы. Телевизоры, магнитофоны, охранные сигнализации – все это управляется компьютером. В этом случае упор делается не на высокую производительность, а на низкую стоимость.

Традиционно такие компьютеры программировались на ассемблере, но с усложнением функций появилась необходимость в использовании языка более высокого уровня. Таким языком стал Java, т.к. он достаточно прост и программы занимают мало места. Недостатки:

- Для использования языка требуется достаточно большой интерпретатор для выполнения кода;
- Процесс интерпретации занимает много времени.

Для разрешения этой проблемы Sun и другие компании разработали процессор со встроенным набором команд Java. Рассмотрим один из процессоров, который был разработан специально для встроенных систем.

Это процессор picoJava II, который составляет основу микросхемы microJava 701. Это однокристалический процессор с двумя интерфейсами шины: один из них предназначен для шины памяти шириной в 64 бита, а другой для шины PCI (рис. 2.6). Процессор может содержать Кэш-память первого уровня (до 16 Кбайт команд и 16 Кбайт данных), но не имеет Кэш-памяти второго уровня.

Особенности микросхемы:

1. используется шина PCI, разработанная Intel, для использования в системах Pentium. Преимущества PCI в том, что она стандартна.
2. Система microJava 701 обычно содержит флш-память.

3. Система microJava 701 содержит 16 программируемых линий ввода-вывода, которые можно связать с кнопками, переключателями и т.д. Наличие программируемых линий ввода-вывода на процессоре исключает необходимость использования программируемых контроллеров, что делает устройство дешевле.

В микросхему microJava 701 встроено также три программируемых тактовых генератора, которые удобно использовать для приборов, управляемых в масштабе реального времени.

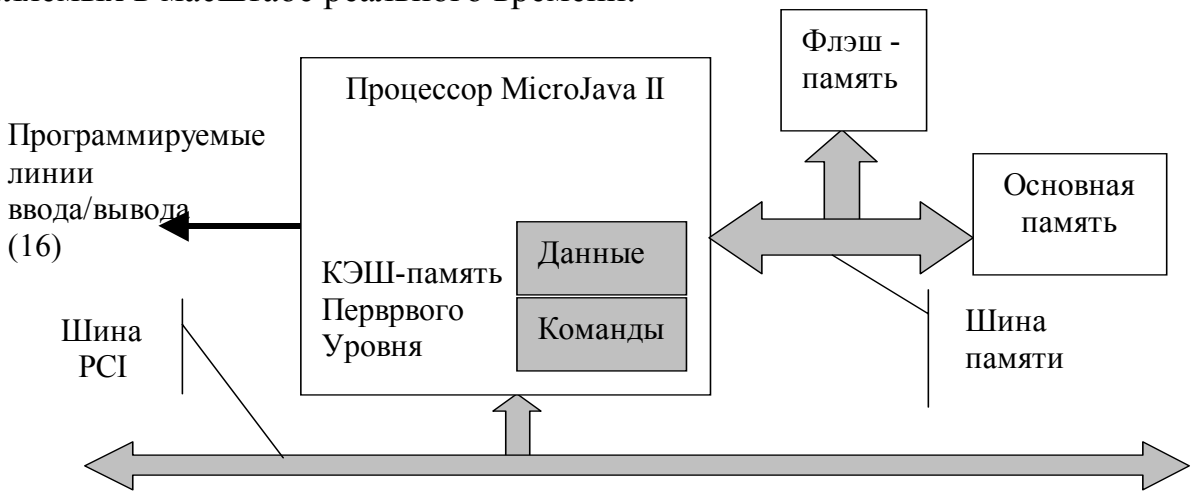


Рис. 2.5

Микросхема выпускается в стандартном корпусе, содержащем 316 выводов. Из них:

- 59 – для шины PCI;
- 123 – для шины памяти (64 двунаправленые для данных и адреса);
- 7 – управление;
- 3 – синхронизация;
- 11 – прерывания;
- 10 – проверка;
- 16 – ввод/вывод.

Имеются выводы «земля» и питание, часть выводов не используется.

Микросхема имеет режим ожидания (экономный), имеет поддержку стандартного тестирования.

2.6. ПРИМЕРЫ ШИН

Шины соединяют компьютерную систему в одно целое. Шина ISA представляет собой небольшое расширение первоначальной шины IBM PC. По соображениям совместимости она все еще используется в компьютерах Intel. Однако такие компьютеры содержат еще одну шину PCI, которая работает быстрее и шире, чем ISA. Шина USB обычно используется как шина ввода/вывода для периферийных устройств малого быстродействия.

Шин ISA

Шина ISA была неофициальным стандартом процессора 8086. Содержала 62 сигнальные линии (20 адресных, 8 данных, сигналы записи, чтения, прерывания, считывания с устройств ввода/вывода).

Когда разрабатывался 80286, то возникла необходимость создания новой шины. Однако для совместимости было решено расширить старую шину. Для этого радом со старым разъемом появился дополнительный на 36 линий.

Когда компания IBM выпустила серию компьютеров PS/2, то компьютеры со средней и высокой производительностью были оснащены новой шиной MCA, которая была защищена патентом.

В ответ компьютерная промышленность ввела свой собственный стандарт ISA (Industry Standard Architecture), которая фактически представляет собой шину PC/AT на частоте 8,33 МГц. Преимущества шины – совместимость. Позднее шина была расширена до 32 разрядов, такая шина называлась EISA (Extended Industry Standard Architecture).

Шина PCI

В 1990 году компания Intel разработала новую шину с гораздо более высокой пропускной способностью, чем у EISA: *PCI (Peripheral Component Interconnect – взаимодействие периферийных компонентов)*.

Первая шина PCI передавала 32 бита за цикл и работала с частотой 33 МГц (время цикла 32 нс); общая пропускная способность составляла 133 Мбайт/с. Сейчас шина PCI 2.2 подходит и для портативных компьютеров (энергоэкономная), работает на частоте 66 МГц, способна передать 64 бита за цикл, пропускная способность 528 Мбайт/с.

Проблемы шины PCI следующие.

1. Пропускная способность в 528 Мбайт/с достаточно высокая, но недостаточная.
2. Шина не совместима со всеми старыми картами ISA.

По этим причинам Intel решила разрабатывать компьютеры с тремя и более шинами, например, как изображено на рис. 2.6.

Ключевым компонентом данной архитектуры являются мосты между шинами. Мост PCI связывает центральный процессор, память и шину PCI. Мост ISA связывает шину PCI с шиной ISA, а также поддерживает один или два диска IDE. Практически все системы Pentium II выпускаются с одним или несколькими свободными слотами PCI и с одним или несколькими слотами ISA.

В принципе можно использовать систему с несколькими шинами каждого типа. Существуют специальные мосты, которые связывают две шины PCI, может быть также несколько мостов ISA.

Шины PCI являются синхронными. Все транзакции в шине осуществляются между задающими и подчиненными устройствами. Для сокращения числа выводов на плате адресные и информационные линии

объединяются. В этом случае достаточно 64 линии на адреса и данные, однако это усложняет протокол обмена. Каждая транзакция может осуществляться за 3 цикла.

Арбитраж шины PCI осуществляется централизованным арбитром. В большинстве случаев арбитр встраивается в один из мостов между шинами. От каждого устройства PCI к арбитру тянется две специальные линии. Одна из них (REQ#) используется для запроса шины, вторая (GNT#) для получения разрешения.

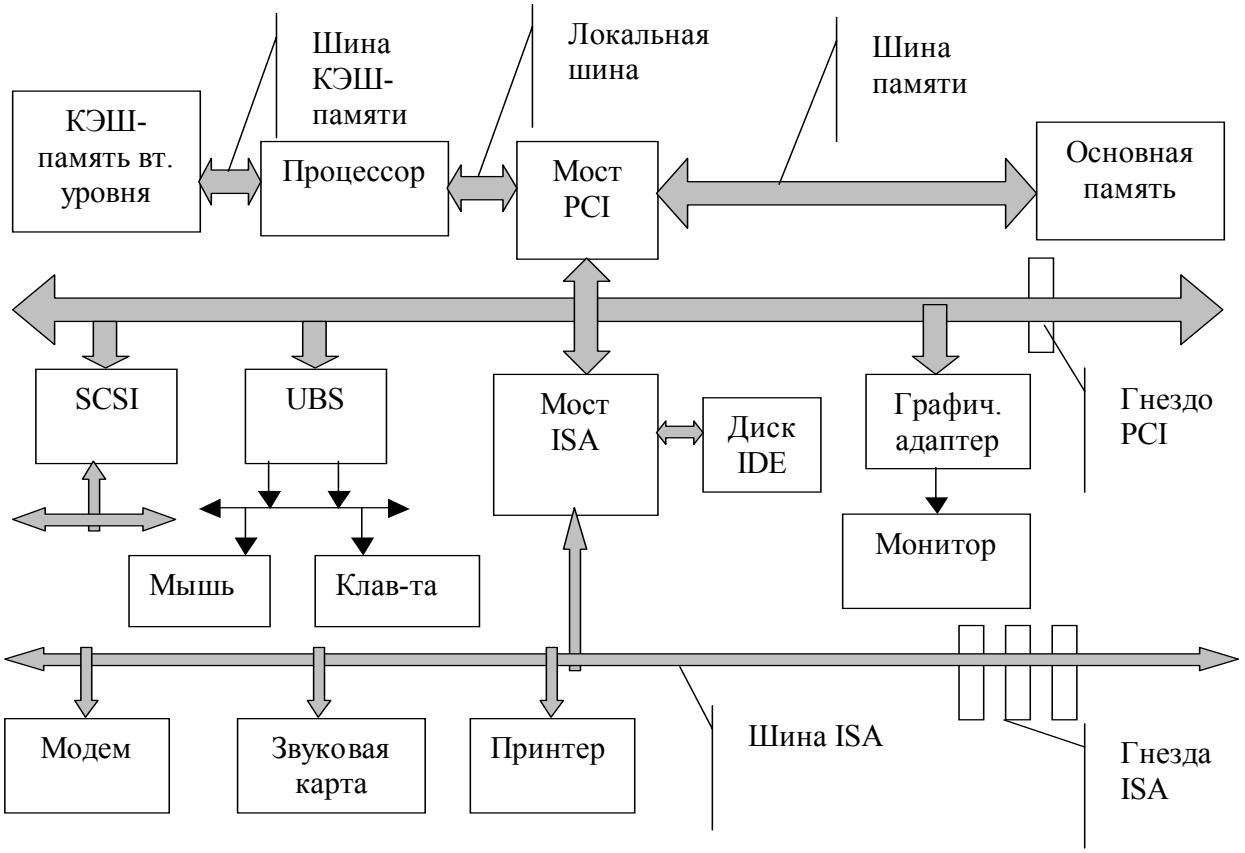


Рис. 2.6

Шина PCI содержит обязательные сигналы (32-битные) и факультативные (на 64 бита). Перечень сигналов можно посмотреть в справочнике и перечислять их не будем.

Шина USB

Шина UBS – результат совместной разработки семи ведущих компаний; она предназначена для подсоединения низкоскоростных устройств. UBS – универсальная последовательная шина.

Требования, предъявляемые шине следующие:

1. Пользователи не должны устанавливать перемычки и переключатели на платах и устройствах.
2. Пользователи не должны открывать компьютер, чтобы установить новые устройства ввода-вывода.

3. Должен существовать только один тип кабеля, подходящий для подсоединения всех устройств.
4. устройства ввода-вывода должны питаться через кабель;
5. Необходима возможность подключения к одному компьютеру до 127 устройств.
6. Система должна поддерживать устройства реального времени.
7. Должна быть возможность устанавливать устройства во время работы компьютера.
8. Должна отсутствовать необходимость перезагрузки компьютера после установки устройства.
9. Производство новой шины и устройств ввода-вывода для нее не должно требовать больших затрат.

Шина UBS полностью удовлетворяет этим условиям. Ее пропускная способность – 1,5 Мбайт/с. Это достаточно для большинства устройств (мышь, клавиатура, фотоаппаратов и т.д.), предел в 1,5 Мбайт выбран из соображений дешевизны.

Шина UBS состоит из *центрального хаба (концентратор)*, который вставляется в разъем главной шины. Этот центральный хаб содержит разъемы для кабелей, которые могут подключаться к устройствам ввода-вывода или к дополнительным хабам. Т.О. топология шины UBS представляет собой дерево с корнем в центральном хабе.

Кабель состоит из 4 проводников: земля, питание, данные, синхронизирующий сигнал. Кодировка: «0» -- изменение сигнала; «1» -- сигнал без изменений; нормальное состояние – высокий.

При подсоединении устройства центральный хаб распознает это и прерывает работу операционной системы. Операционная система запрашивает устройство, что оно собой представляет и какая пропускная способность ему требуется. Если операционная система считает, что пропускной способности достаточно, она приписывает устройству уникальный адрес.

Шина UBS представляет собой ряд каналов от центрального хаба к устройствам ввода-вывода. Каждое устройство может разбить свой канал на 16 подканалов для различных типов данных (например, аудио и видео). В каждом канале или подканале данные перемещаются от центрального хаба к устройству и обратно; между двумя устройствами ввода-вывода обмен информацией не происходит.

3. МИКРОАРХИТЕКТУРНЫЙ УРОВЕНЬ

3.1. МИКРОАРХИТЕКТУРА

Над цифровым логическим уровнем находится микроархитектурный уровень. Его задача – интерпретация уровня команд в управляющие сигналы (команды) для цифровых устройств. Строение микроархитектурного уровня

зависит от того, каков уровень архитектуры команд, от стоимости и назначения компьютера.

В настоящее время уровень архитектуры команд часто содержит простые команды, которые выполняются за один цикл (системы RISC). В других системах (например Pentium II) на этом уровне имеются более сложные команды, выполняемые за несколько циклов.

Для того, чтобы выполнить команду небходимо

- Найти операнды в памяти;
- Считать операнды;
- Выполнить операцию;
- Записать полученный результат в память.

Управление уровнем команд со сложными командами отличается от управления уровнем с простыми операциями, т.к. выполнение сложных команд требует определенной последовательностти операций.

Сейчас хотелось бы описать общие принципы разработки микроархитектурного уровня. К сожалению, таких общих принципов не существует и каждая разработка индивидуальна. Поэтому мы будем рассматривать конкретные примеры. Рассмотрм некоторые понятия присущие всем разработкам.

Микроархитектура содержит микропрограмму, располагаемую в ПЗУ. Микропрограмма должна вызывать, декодировать и выполнять команды. Поскольку аппаратное обеспечение состоит из вентилей и регистров, то микропрограммы должна выдавать команды, управляющие этими вентилями и регистрами.

Принято считать, что разработка микроархитектуры – это проблема программирования. Каждая команда интерпретируется как функция, которая реализуется микропрограммой.

Микропрограмма содержит набор переменных, к которым имеют доступ все функции. Эти переменные называются *состоянием* компьютера. Каждая ыункция изменяет некоторые переменные, формируя новое сосотяние компьютера. Например, счетчик команд указывает на местонахождение функции (команды уровня архитектуры команд), которую нужно выполнить следующей.

Каждая команда состоит из нескольких полей, каждое из которых выполняет определенную функцию. Первое поле называется *кодом операции.* Это поле определяет действие, которое не необходимо выполнить. Другие поля указывают на операнды, типы и т.д. Чем сложнее команда, тем больше полей требуется для ее описания.

3.1.1. Тракт данных

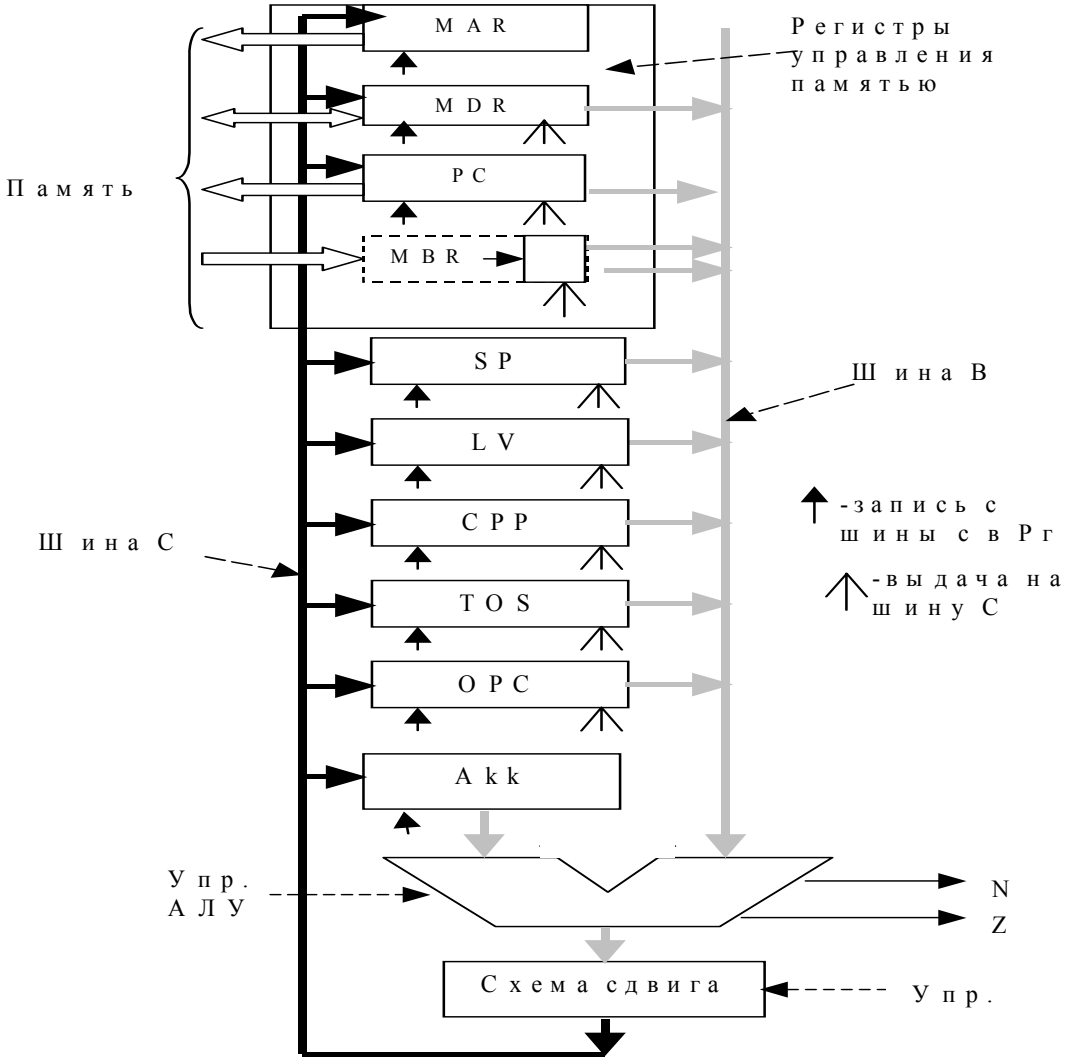


Рис. 3.1

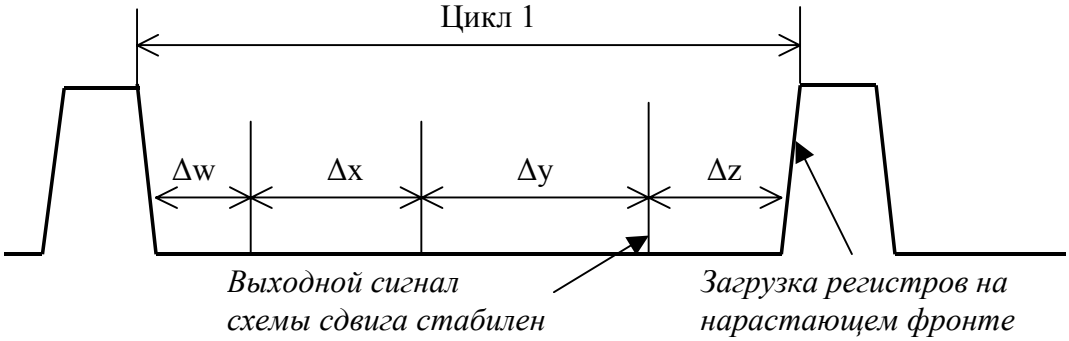
Тракт данных – это часть центрального процессора, состоящего из АЛУ и его входов и выходов. В общем тракт данных содержит ряд регистров и АЛУ. Регистры и АЛУ связаны между собой шинами. Под действием управляющих сигналов содержимое определенных регистров передается на вход АЛУ, а результат записывается в требуеый регистр. Укрупненная и упрощенная структурная схема тракта данных приведена на рис. 3.1.

Функционирование АЛУ зависит от управляющих сигналов, имеет два входа. Схема сдвига имеет свое управление.

За один цикл АЛУ можно считать и записать один и тот же регистр. Процессы считывания и записи происходят в разных частях цикла. Содержимое одного из регистров выдается на шину С в начале цикла и сохраняются там на протяжении всего цикла. Затем АЛУ выполняет операцию, результат которой через схему сдвига поступает на шину С. После стабилизации сигналов на шине С ее содержимое передается в один или несколько регистров. Одним из нескольких регистров может быть тот, от которого поступил сигнал на шину В. Для реализации этих действий должна быть выполнена синхронизация тракта данных.

3.1.2. Синхронизация тракта данных

Синхронизация тракта данных иллюстрируется рис. 3.2.



Δw—установка сигналов для запуска тракта данных
Δx—загрузка данных в Акк и шину B
Δy—АЛУ и схема сдвига
Δz—продвижение сигнала из схемы сдвига в регистры

Рис. 3.2

Хотя в трактк данных нет запоминающих устройств, для прохождения сигналов требуется некоторое время. Для правильной работы тракта данных требуется жесткая синхронизация и достаточно длительный цикл. Никаких синхронизирующих сигналов на вход тракта данных в течение цикла не поступает. Поэтому, необоснованное сокращение длительности цикла может привести к ошибкам работы тракта данных. АЛУ и схема сдвига работают постоянно, но в течение времени Δw+Δx входные сигналы не действительны, а в течении времени Δw+Δx+Δy недействительны выходные сигналы.

3.1.2. Работа памяти

Работа с памятью может быть осуществлена двумя способами: через порт с пословной адресацией (регистры MAR и MDR) и через порт с байтовой адресацией (MBR – буферный регистр). Регистры MAR и MDR используются для чтения и записи слов, а регистры MBR и PC – для считывания программы на уровне архитектуры команд в виде потока байтов.

3.1.3. Микрокоманды

Для управления трактом данных необходимы управляющие сигналы (29), которые можно разделить на 5 функциональных групп:

- Сигналы для записи данных из шины C в регистры (9);
- Сигналы для разрешения передачи содержимого регистров на шину B и в АЛУ (9);
- Сигналы управления АЛУ и схемой сдвига (8);
- Сигналы управления регистрами MAR\MDR (2);

- Сигналы управления регистрами PC\MBR (1).

Значения этих сигналов определяют операции для одного цикла тракта данных. Если был установлен сигнал обращения в память в k-ом цикле, то данные из памяти могут появиться в регистрах MDR(MBR) только в конце следующего k+1-го цикла, а использовать эти данные можно только k+2 цикле и то только в том случае, если эти данные были в КЭШ-памяти.

3.1.3. Разработка микроархитектурного уровня

При разработке микроархитектурного уровня постоянно приходится искать компромисс между желаниями и возможностями. При разработке центрального процессора очень важную роль играет выбор между высокой скоростью и низкой стоимостью, между сложностью программ и сложностью аппаратного обеспечения.

Скорость и стоимость

Существует три основных подхода , которые позволяют увеличить скорость выполнения операций:

1. Соркащение количества циклов, необходимых для выполнения кманды.
2. Упрощение организации машины таким образом, чтобы можо было сделать цикл короче.
3. выполнение нескольких операций одновременно.

Первые два подхода очевидны, но существуют различные способы их реализации. Число циклов, необходимых для выполнения операции азывается длиной пути. Длину пути можно уменьшить за счет введения дополнительного оборудования и использования параллельного выплнения команд. Основные подходы:

- в микропрограмме, реализующей команду, находят циклы, для выполнния которых не требуется работа АЛУ. Такие циклы ставить в конце последовательности микрокоманд.
- Введение дополнительной шины. Т.е. к АЛУ подвести две шины и на входы подавать сигналы с произвольных регистров. Таким образом исключается цикл передачи одного из операндов в Акк, но усложняется кодировка микрокоманды и аппратура -- *переход к трехшинной архитектуре*.
- Введение блока выборки команд: процедура выборки следующей команды передается отдельному блоку – *команды из памяти должны вызываться специализированным функциональным блоком.*

Конвейерная архитектура

Следующий рассматриваемый вариант усовершенствования архитектуры – ввести в машину больше параллелизма.

Длительность цикла определяется временем, необходимым на прохождение сигнала через тракт данных. В цикле тракта данных есть три основных компонента:

1. Время, необходимое на передачу значений выбраных регистров на входы АЛУ.
2. Время рабты АЛУ и схемы сдвига.
3. Время на передачу полученых значений в регистры и сохранение результатов.

Таким образом один цикл можно разбить на 3 цикла, более которкие. Продвижение микрокоманды осуществляется по трем устройствам. Следующая микрокоманда может быть начата до окончания предыдущей. Проблема: RAW-взаимозависимость (Read After Write – чтение после записи) – последующей операции могут понадобиться данные, которые еще не готовы. В таком случае возникает простой.

Это простейший конвейер с 3 стадиями, количество стадий (степень параллелизма может быть увеличена).

3.1.4. Увеличение производительности.

Усовершенствования компьютеров распадаются на две категории: усовершенствование реализации и усовершенствование архитектуры.

Усовершенствование реализации – такие способы построения нового процессора и памяти, после применения которых система работает быстрее, но архитектура не меняется. Это означает, что старые программы будут работать на новой машине. Это очень нравится потребителям. Например, улучшение производительности от 80386 к 80486, Pentium, Pentium Pro, Pentium II происходило без изменения архитектуры.

Однако возникаем момент, когда старая архитектура себя исчерпала и единственный способ развивать технологии дальше – начать новую разработку. Таким революционным скачком было появление RISC в 80-х годах.

КЭШ-память

Одним оз основных вопросов при рзработке компьютеров является построение такой системы памяти, которая могла бы передавать операнды процессору с той же скоростю, с которой она их обрабатывает. Однако, производительность процессора растет значительно быстрее, чем быстродействие памяти и относительно процессора память работает все медленне с каждым десятилетием и эта ситуация все ухудшается.

Одним из способов решения этой проблемы является добавление КЭШ-памяти. Основная технология – введение разделеной КЭШ-памяти – отдельно для операндов и отельно для команд. В такой КЭШ-памяти операции могут начинаться независимо, что увеличивает пропускную способность системы в целом.

Помимо этого между КЭШ-памятью иосровной паятью часто помещают КЭШ-паять второго уровня.

Существует два типа локализации адресов:

- *Пространственая локализация*—основана на вероятности того, что в скором времени потребуется обратиться к ячейкам памяти, которые расположены рядом с недавно выбранными ячейками;
- *Временная локализация* –имеет место, когда недавно запрашиваемые ячейки запрашиваются снова. Принцип временной локализации используется в тех случаях, когда решается вопрос о том, какой элемент выкинуть из КЭШ-памяти в случае промаха. Обычно отбрасываются элементы, к которым не было обращений.

Во всех типах КЭШ-памяти используется следующая модель: основная память разделяется на блоки фиксированного размера, которые называются строками КЭШ-памяти. Строка состоит из нескольких последовательных байтов (от4 до 64). При обращении к памяти сначала проверяется наличие требуемой иформации в КЭШ-памяти и в случае промаха из КЭШ-памяти удаляется строка, а на ее место помещается требуемая из основной памяти. Трока большн, чем вызывемый элемент.

Прогнозирование ветвлений

Современные компьютеры сильно конвейеризорованы, они могут содержать десять и более стадий. Но конвейеры дают высокое быстродействие только на линейном коде, а в случае наличия ветвления возникает проблема с выбором следующей команды.

Первые конвейеризированные процессоры простаивали до тех пор, пока не становилось известно, куда нужно выполнить переход.

Современные машины содержат средства, позволяющие прогнозировать переходы. Первое предположение заключается в том, что переход будет назад, т.к. если переход находится в конце цикла, то переход назад более вероятен. Если переход предсказан неправильно, то необходимо отменить выполненные команды.

Динамаческое прогнозирование ветвлений. Один из самых простых способов – хранить специальную таблицу (в аппаратном обеспечении), в которую центральный процессор записывает условные преходы, когда они встречаются и там их можно искать, если он снова появляется. В такой таблице хранится адрес перехода и бит, который указывает, был ли сделан этот переход. Прогноз состоит в том, что программа пойдет тем же путем, что и в предыдущий раз. Если прогноз не верен, то бит меняется.

Статическое прогнозирование ветвлений. Динамическое прогнозирование во время работы программы и это их положительное качество. Однако реализация динамического прогнозирования ветвлений требует специализированного и достаточно дорогого аппаратного обеспечения. Такое прогнозирование осуществляется компилятором.

В некоторых машинах (UltraSPARC) введен дополнительные команды перехода, которые содержат бит, который указывает, совершать переход или нет.

Исполнение с изменением последовательности и подмена регистров

Большинство современных компьютеров являются конвейеризированными и суперскалярными. Это означает, что там есть блок выборки команд, который заранее вызывает команды из памяти и передает их в блок декодирования. Блок декодирования передает декодированные команды в соответствующие функциональные блоки для выполнения.

При такой организации возникает следующая проблема: если выполняемой команде нужно значение, получаемое в предыдущей, то она не может начаться до тех пор, пока значение не будет получено. Существуют и другие виды взаимозависимости.

Для устранения этой проблемы используется подход, который заключается в том, что изменяется последовательность выполнения команд таким образом, чтобы зависимые команды не стояли друг за другом.

Возможна ситуация, при которой возникает конфликт при обращении к регистрам: необходимо записать информацию в регистр, а там еще хранятся не востребованные данные. Этого можно избежать, если записывать информацию не в занятый регистр, а куда-то в другое место (можно временно). Для реализации указанных возможностей необходимо вводить дополнительные аппаратные и программные средства.

Спекулятивное выполнение

Компьютерные программы можно разбить на *базовые элементы,* каждый из которых представляет собой линейную последовательность команд с точкой входа и точкой выхода в конце. Базовый элемент не содержит никаких управляющих структур. Базовые элементы связываютя между собой операторами управления. Рассмотренный выше прием переупорядочения команд эффективен внутри базового элемента.

Большинство базовых элементов очень короткие и в них недостаточно параллелизма.

Для повышения эффективности нужно сделать так, чтобы переупорядочение команд можно было применить не только в пределах одного базового элемента. Выгоднее всего передвинуть потенциально более медленный базовый элемент передвинуть вперед, чтобы его выполнение началось раньше. Такой операцией может быть операция считывания из памяти, операция с плавающей точкой и т.д. Перемещение операции вверх называется *подъемом.*

В этом случае возникает проблема: понадобится ли эта операция? Выполнение команды до того, как стало известно, понадобится ли она, называется *спекулятивным выполнением.* Чтобы использовать эту

технологию, требуется поддержка компилятора, аппаратного обеспечения и некоторое усовершенствование архитектуры.

В связи со спекулятивным выполнением команд возникают некоторые проблемы. Важно, чтобы ни одна из спекулятивных команд не имела окончательного результата, который нельзя отменить. Эта проблема решается путем подмены регистров.

Вторая проблема: что делать, если запрашиваемые спекулятивной командой данные не находятся в КЭШ-памяти? В ряде современных компьютеров в такой ситуации команда не выполняется.

3.1. МИКРОАРХИТЕКТУРА ПРОЦЕССОРА Pentium II

Pentium II – поддерживает 32-битные операнды и арифметику, 64-битные операции с плавающей точкой, а также 8-ми и 16-ти битные операции, унаследованные от предыдущих моделей. Процессор может адресовать до 64 Гбайт памяти и считывать слова по 64 бита за раз. Обычная система Pentium II изображена на рис. 3.2.

Рис. 3.2

Конструктивно в корпусе Pentium II помещены две интегральные схемы: центральный процессор (на котором находится разделенная КЭШ-память первого уровня и объединенная КЭШ-память второго уровня. На рис. 3.2 показаны основные компоненты центрального процессора: блок вызова-декодирования, блок отправки-выполнения, блок возврата, которые вместе действуют как конвейер высокого уровня. Эти три блока обмениваются данными через пул команд – место для хранения информации о частично выполненных командах. Информация в пуле команд находится в таблице, которая называется *ROB (ReOrder Buffer – буфер переупорядочных команд).*

Блок вызова/декодирования вызывает команды и разбивает их на микрооперации для хранения в ROB. Блок отправки/выполнения получает микрооперации из буфера ROB и выполняет их. Блок возврата завершает выполнение каждой операции и обновляет регистры. Команды поступают в ROB по порядку, могут выполняться в произвольном порядке, но завершаться должны по порядку.

Блок сопряжения с шиной отвечает за обмен информацией системой памяти (Кэш-память второго уровня и основная память). Кэш-память второго уровня не связана с локальной шиной, поэтому блок сопряжения с шиной отвечает за вызов данных из основной памяти через локальную шину, а также за загрузку всех блоков кэш-памяти.

Блок вызова/декодирования

Блок вызова/декодирования (рис. 3.3) отличается высокой степенью конвейеризации (содержит семь стадий). Блок отправки/выполнения и блок возврата имеют еще пять стадий – всего 12. Команды поступают в конвейер на стадии IFU0 (Instruction Fetch Unit – блок выборки команд), куда из кэш-памяти команд загружается целые 32-битные строки. Регистр NEXT IP (NEXT Instruction Point – следующий указатель команд) управляет процессом вызова команд.

Поскольку в наборе команд Intel IA-32 содержатся команды разной длины и разного формата, то IFU1 – анализ потока байтов и определение начала команды. (*Замечание. На стадии 1 предусмотрена возможность предварительного анализа до 32 команд. но, как правило, на таком отрезке команд встречаются команды перехода, которые не всегда правильно прогнозируются. Поэтому в возможности анализа такого большого числа команд нет смысла.)*

На стадии IFU2 команды выравниваются и без труда декодируются на стадии ID0. В системе Pentium II декодирование состоит из превращения каждой команды IA-32 в одну или несколько микроопераций. На стадии ID0 имеется три внутренних декодера: два предназначены для коротких команд, а третий обрабатывает остальные. На выходе получается последовательность микроопераций. Каждая микрооперация содержит код операции, два входных и один выходной регистр.

На стадии ID1 выстраивается очередь микроопераций. На этой стадии происходит прогнозирование ветвлений (статическое, на всякий случай). Затем используется динамическое прогнозирование ветвлений.

Чтобы избежать взаимозависимостей WAW и WAR (использование регистров, в которые записывается/считывается результат) используется распределитель регистров RAT.

Микрооперации копируются в буфер ROB со скоростью 3 микро-операции за цикл. В этот же буфер собираются операнды, если они имеются в наличии. Если все готово для выполнения микрооперации и ресурсы доступны, то она выполняется. В противном случае, микрооперация находится в блоке ROB, пока не появятся необходимые ресурсы.

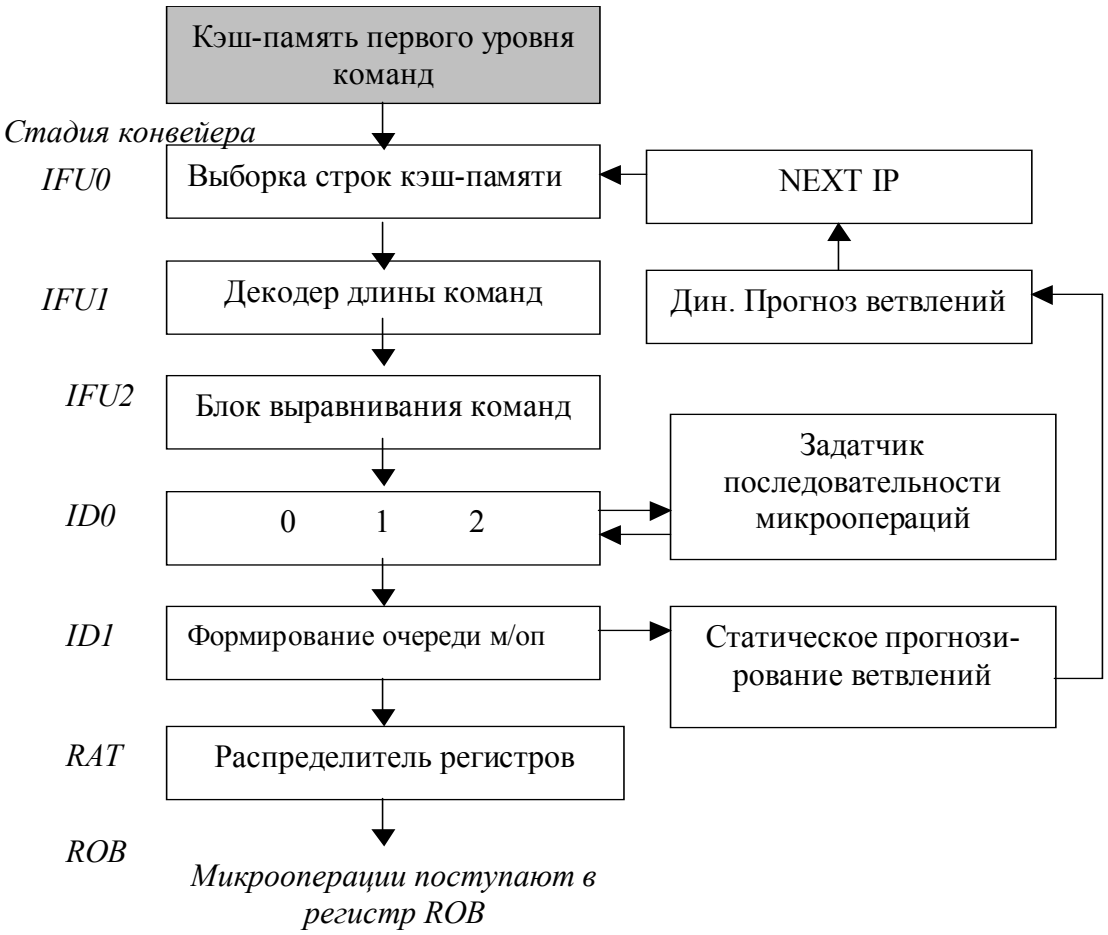


Рис. 3.3

Блок отправки/выполнения

Блок отправки/выполнения приведен на рис. 3.4. Этот блок устанавливает очередность и выполняет микрооперации, разрешает взаимозависимости и конфликты ресурсов. Хотя за один цикл можно декодировать всего три команды (стадия ID0), за один цикл можно выпустить для выполнения целых 5 микроопераций (по количеству портов). Такая скорость превышает возможности блока возврата. Чтобы следить за микрооперациями, регистрами и функциональными блоками, требуется сложный счетчик обращений. Если операция готова к выполнению, то она может начинаться, даже если другие, поступившие ранее, еще не готовы. Если несколько микроопераций пригодны для выполнения одним и тем же функциональным блоком, то выбирается важнейшая из них. Например, выполнение перехода важнее, чем выполнение арифметического действия, поскольку переход влияет на ход выполнения программы.

Блок отправки/выполнения состоит из резервации и функциональных блоков, которые связаны с пятью портами. Резервация представляет собой очередь из 20 элементов для микроопераций, которые имеют собственные операнды. Они ожидают очереди в резервации, пока не освободится нужный функциональный блок.

Между резервацией и функциональными блоками имеется пять портов. Некоторые функциональные блоки разделяют один порт. Блок загрузки и блоки запоминающих устройств выдают информацию для операций загрузки и сохранения соответственно. Для запоминающих устройств есть два порта. Через один порт за один цикл может выдаваться только одна микрооперация.

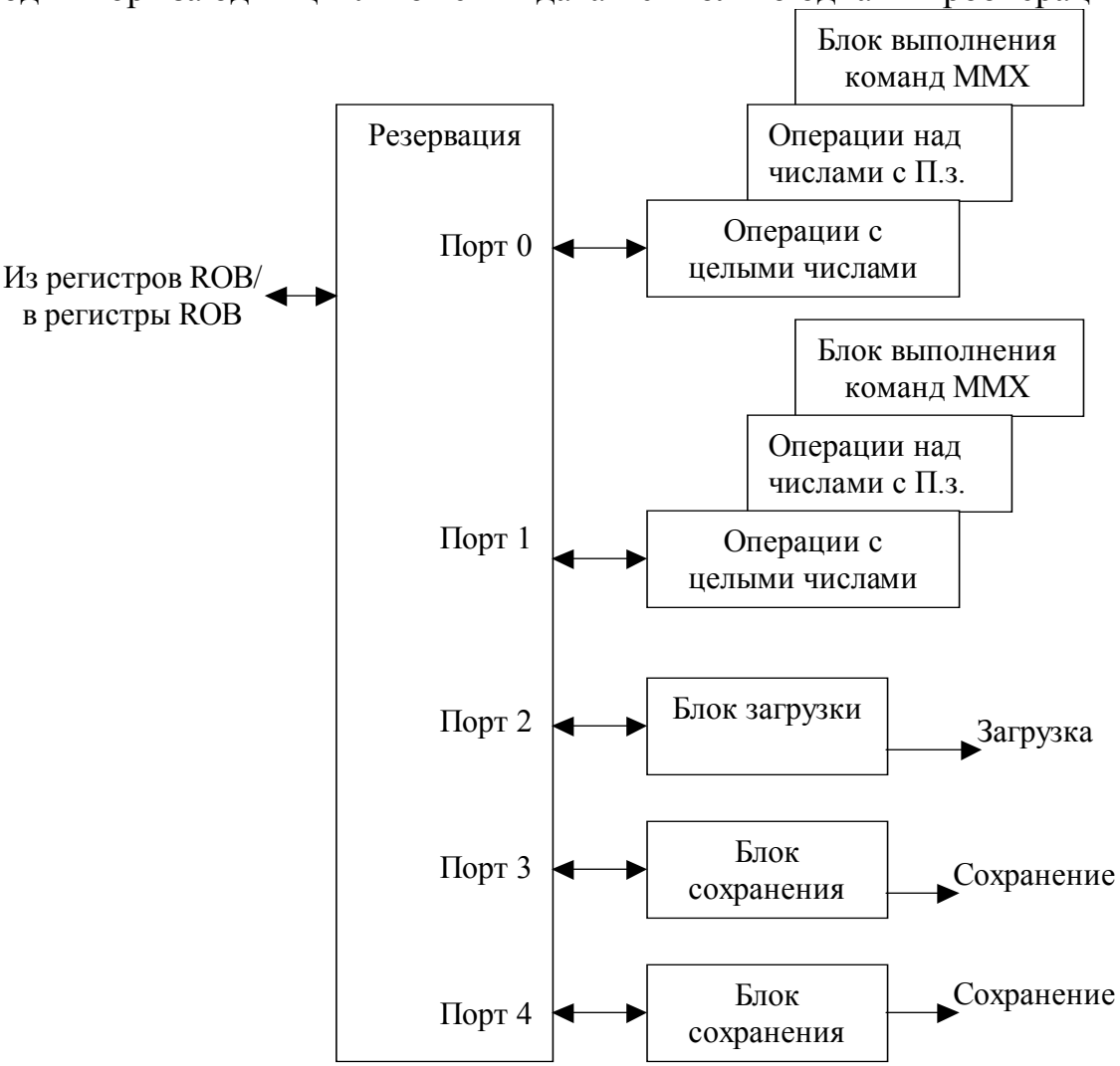


Рис. 3.4

Блок возврата

После выполнения микрооперация переходит обратно в резервацию, а затем в буфер ROB и там ожидает возврата. Блок возврата отвечает за отправку результата в требуемое место – в соответствующий регистр или другие устройства блока отправки/выполнения. Блок отправки/выполнения содержит «официальные» регистры, т.е. те, в которых хранятся результаты завершенных команд, а также ряд «промежуточных» регистров. В регистры возвращаются только результаты «официально» выполненных команд, причем это происходит в том порядке, что и в программе, даже если команды выполнялись в произвольном порядке.

3.2. МИКРОАРХИТЕКТУРА ПРОЦЕССОРА UltraSPARC II

UltraSPARC II – это 64-разрядная машина (по данным и адресам), но в целях совмещения с предыдущими версиями, она воспринимает 32-разрядные операнды, адреса и программы. Шина памяти 128-битная. Ядро системы представлено на рис. 3.5.

Вся серия SPARC изначально представляла собой систему RISC. Большинство команд – трехадресные, поэтому они очень хорошо подходят для конвейеризации и нет необходимости разбивать команды CISC на микрооперации RISC (как в Pentium II).

UltraSPARC II это суперскалярная машина, которая может выдавать 4 команды за цикл. Команды запускаются по прядку, а завершаться могут в произвольном прядке.

Общий обзор системы UltraSPARC II

Диаграмма UltraSPARC II представлена на рис. 3.4. Все указанные на рисунке компоненты расположены на микросхеме центрального процессора за исключением кэш-памяти второго уровня, которая является внешней по отношению к процессору. Кэш команд – 16 Кбайт со строками по 32 байта и возможностью возврата половины строки. Половина строки (16 байт) содержит ровно 4 команды, которые могут выдаваться за один цикл.

В случае промаха кэш-памяти первого уровня нужная строка ищется в кэш-памяти второго уровня. В случае успеха строка копируется в кэш первого уровня. В случае неудачи внешняя кэш-память (второго уровня) посылает команду вызова строки из основной памяти.

Блок выборки/отправки в целом похож на блок вызова/декодирования системы Pentium II. Однако его работа проще, т.к. нет необходимости определять длину команды, выравнивать и т.д. Блок может вызывать команды из кэш-памяти первого и второго уровня без потери времени. За один цикл вызываются 4 команды. Устройство предсказания ветвлений находится внутри блока выборки/отправки. UltraSPARC II содержит ряд команд перехода, в которых компилятор может сообщать аппаратному обеспечению способ предсказания переходов.

Схема группировки – выбирает по четыре команды за один раз из очереди для запуска. Задача состоит в том, чтобы найти 4 команды, которые можно запустить одновременно. Блоки выполнения команд содержат по два

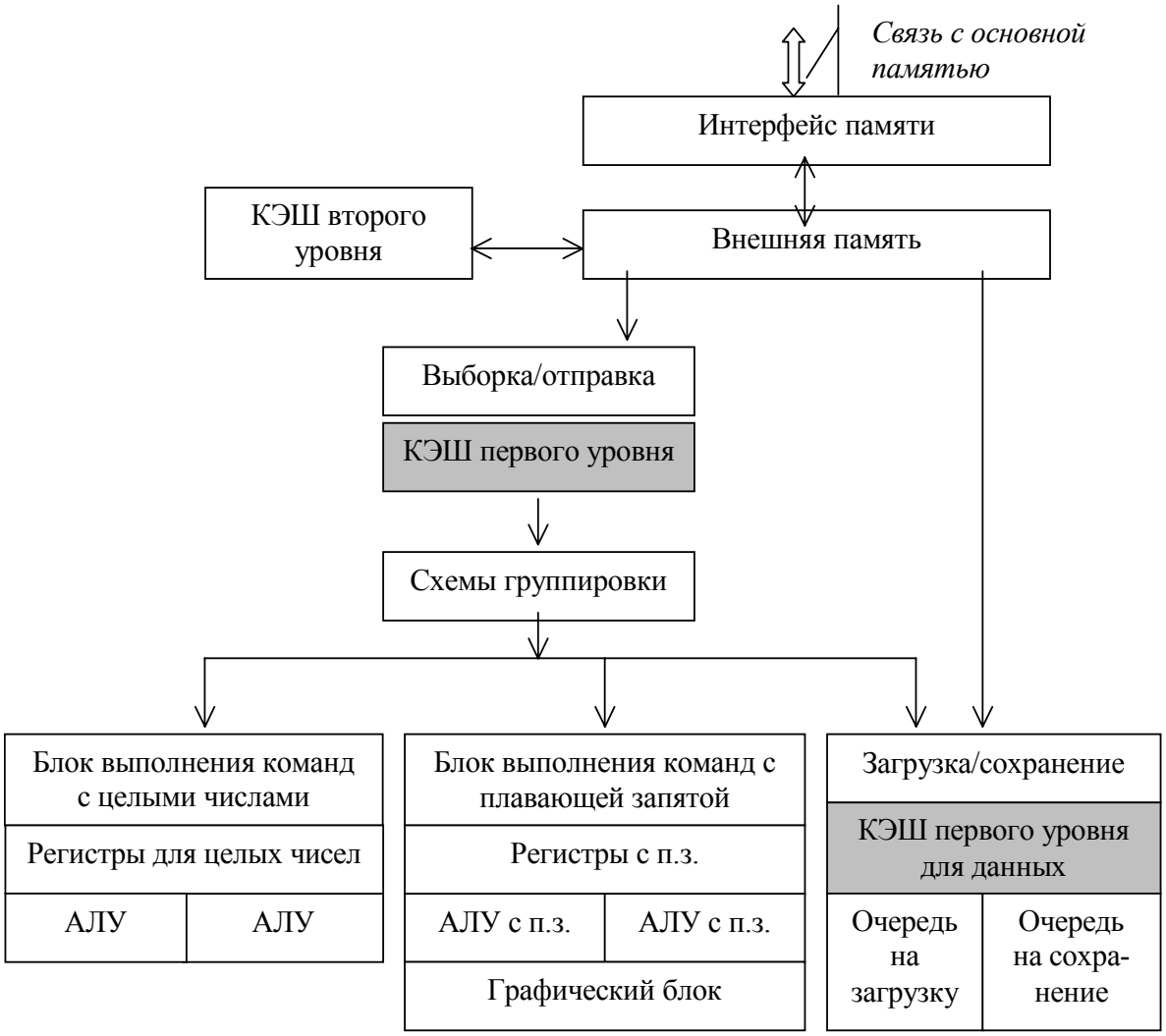


Рис.3.5

АЛУ, следовательно, в одной группе может содержаться по две команды каждого типа. Чтобы сделать загрузку оптимальной команды должны запускаться не по порядку, что допускается. Завершаются команды также в произвольном порядке.

Блоки выполнения команд имеют собственные регистры, поэтому команды берут операнды внутри блоков и там же оставляют результаты. В блоке вычислений с п.з. находится графический блок, который выполняет специальные команды двух- и трехмерных изображений, аналогичных командам MMX системы Pentium II.

Блок загрузки/сохранения управляет командами записи и считывания.

Конвейеризация системы UltraSPARC II

Система UltraSPARC II содержит конвейер с 9 стадиями. Некоторые стадии различны для команд с целыми числами и команд с п/з (рис. 3.6).

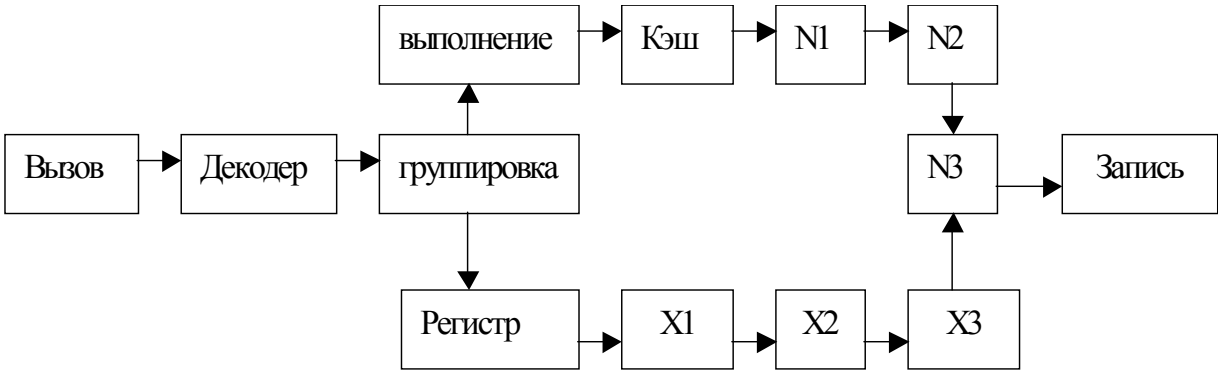


Рис. 3.6

На первой стадии вызываются команды из кэш-памяти команд (если это возможно). При благоприятных условиях (отсутствие промахов кэш-памяти, неправильного прогнозирования ветвлений, наличии правильной смеси команд и т.д.) машина может продолжать вызывать и запускать 4 команды за цикл. На стадии декодирования перед копированием команд в очередь к каждой команде прибавляются дополнительные биты. Эти биты ускоряют обработку (например, помечается функциональный блок, на котором выполняется команда).

Стадия группировки соответствует схеме группировки, рассмотренной ранее. На стадии декодирования команды объединяются в группы по 4 команды, так, чтобы они могли выполняться за один цикл.

С этого момента стадии конвейера для операций с целыми числами и п/з разделяются. Операции над целыми числами выполняются за один цикл. Однако команды записи/считывания требуют дополнительной обработки на стадии кэширования. На стадиях N1 и N2 никаких действий не выполняется, но эти этапы необходимы для синхронизации работы двух конвейеров.

Блок с п/з содержит 4 отдельные стадии. Первая нужна для доступа к регистрам с п/з. Следующие три нужны для выполнения команды. Все команды с п/з выполняются за три цикла, за исключением команды деления (12 циклов) и извлечения квадратного корня (22 цикла).

Стадия N3 общая для обоих блоков, нужна для разработки исключительных ситуаций, например, деления на нуль.

На последней стадии результаты записываются обратно в регистр. Эта стадия напоминает блок возврата системы Pentium II в том смысле, что если команда не прошла эту стадию, она считается незавершенной.

3.3. МИКРОАРХИТЕКТУРА ПРОЦЕССОРА picoJava II

Общий обзор системы PicoJava II

Диаграмма микроархитектуры PicoJava II представлена на рис. 3.7.

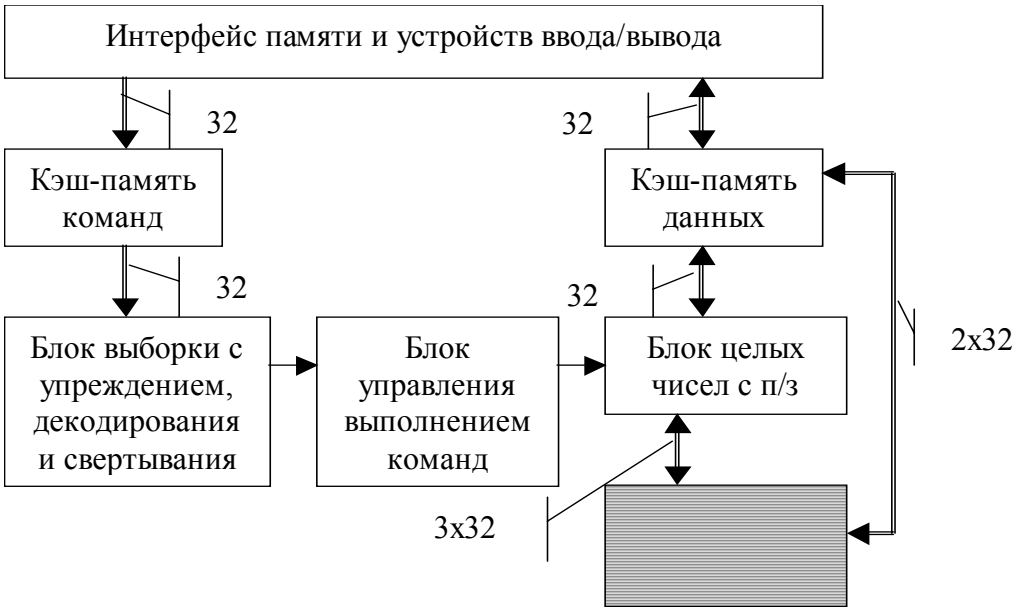


Рис. 3.7

Микросхема процессора содержит разделенную кэш-память первого уровня. Объем кэш-памяти команд может составлять 1, 2, 4, 8 или 16 Кбайт. Размер строки составляет 16 байт. Кэш-память данных также может составлять 1, 2, 4, 8 или 16 Кбайт. Это двухвходовая ассоциативная память. Размер строки составляет 16 байт.

Каждая кэш-память соединяется с шиной памяти по 32-битному каналу. Система microJava 701 имеет оба блока кэш-памяти в обязательном порядке объемом в 16 Кбайт. Блок с п/з является факультативным.

Кэш-память команд передает в блок вызова, декодирования и свертывания по 8 байт за раз. Этот блок связан контроллером выполнения и основным трактом данных (блок целых чисел и блок чисел с п/з). Ширина тракта данных –32 бита для целочисленных операций. Этот тракт также может управлять плавающей точкой с одинарной и двойной точностью.

Регистровый файл состоит из 64 32-битных регистров. В этих регистрах может содержаться верхние 64 слова стека JVM, что существенно повышает скорость доступа к словам в стеке. И стек операндов, и стек локальных переменных под ним могут находиться в регистровом файле. Ширина канала между регистровым файлом и блоком операций с целыми числами и с п/з составляет 96 бит. За один цикл происходит 2 32-битных считывания и из стека и одна запись в стек.

Если, например, стек операндов состоит из 2-х слов, то в регистровом файле может находиться до 62 слов локальных переменных. Естественно, что при помещении еще одного слова возникает проблема. Происходит *дрибблинг* – одно или несколько слов, находящиеся глубоко в стеке

записываются обратно в память. Точно также, если несколько слов выталкиваются из стека операндов, в регистровом файле освобождается место и поэтому некоторые слова, находящиеся глубоко в стеке могут перезагружаться в регистровый файл. Специальные регистры на микросхеме определяют, насколько полным должен быть регистр, чтобы слова из нижней части стека записывались в память, и насколько пустым он должен быть, чтобы перезагрузить регистровый файл из памяти. Чтобы легко произвести дрибблинг без копирования, регистровый файл действует как кольцевой буфер с указателем на самое нижнее и самое верхнее слова. Дрибблинг производится автоматически всякой раз, когда регистровый файл переполняется или пустеет.

Конвейер системы picoJava

Конвейер системы picoJava II состоит из 6 стадий (рис. 3.8).

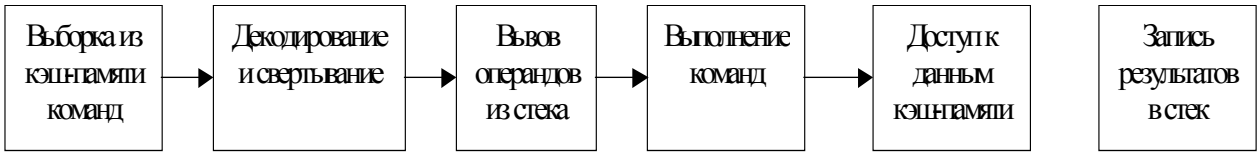


Рис. 3.8

На первой стадии из кэш-памяти команд в буфер команд вызываются команды по 8 байт за раз. Емкость буфера команд – 16 байт.

На следующей стадии команды декодируются и определенным образом объединяются. На выходе блока образуется последовательность трехадресных операций. В этом отношении просматривается сходство с Pentium II. Но, в отличии от Pentium II, машина picoJava не является суперскалярной и микрооперации начинаются и завершаются в том порядке, в котором они запускаются. В случае промаха кэш-памяти, если операнд приходится вызывать из памяти, процессор должен простаивать.

На третьей стадии вызываются операции из стека (фактически из регистрового файла).

Четвертая стадия – выполнение команд.

На пятой стадии в случае необходимости производится обращение к кэш-памяти данных (например, чтобы сохранить там результаты).

Наконец, на шестой стадии результаты записываются обратно в блок.

Свертывание команд

Блок декодирования способен *свертывать* операции. Суть свертывания заключается в том, что блок декодирования определяет условия, при которых не требуется обращения в память операнды находятся в регистровом файле и выполняет операцию над регистрами, не перемещая операнды в вершину

стека. Способность сворачивать операцию CISC с многочисленным обращением в память в простую микрооперацию является ключом к высокой производительности.

3.4. СРАВНЕНИЕ Pentium, UltraSPARC и picoJava

Машина Pentium Iicодержит старый набор команд CISC. UltraSPARC – система RISC. PicoJava – машина со стековой организацией и командами различной длины, которые совершают огромное число обращений в память.

Все три машины имеют сходные функциональные блоки. Все функциональные блоки принимают микрооперации, которые содержат код операции, два входных и один выходной регистр. Все они могут выполнять микрооперации за один такт. Все они конвейеризированы и применяют прогнозирование ветвления. Все содержат раздельную кэш-память для команд и данных.

Главное различие между Pentium II, UltraSPARC и PicoJava – переход от набора команд к функциональному блоку. Pentium разбивает команды CISC для перехода к трехрегистровому формату и делает из больших команд маленькие микрооперации. PicoJava решает обратную проблему – скомбинировать несколько команд для получения того же трехрегистрового формата. UltraSPARC ничего не надо делать, поскольку ее первоначальные команды уже представляют собой маленькие удобные микрооперации. Вот поэтому большинство новых архитектур команд – архитектуры типа RISC (если нет причин использовать другое).

Все рассмотренные машины конвейеризированы. Больше всего стадий у Pentium, т.к. приходится разбивать команды произвольной длины. UltraSPARC содержит больше стадий, чем ему требуется, поскольку конвейер для целочисленных вычислений искусственно удлинен на две стадии в целях синхронизации с конвейером для п/з.

Вывод: при современном состоянии техники оптимальным является конвейер с 6-7 стадиями, который обрабатывает трехрегистровые операции.

4. УРОВЕНЬ АРХИТЕКТУРЫ КОМАНД

Исторически уровень архитектуры команд развивался раньше других и сначала был единственным. В наши дни этот уровень называют «архитектурой» машины. Уровень архитектуры команд имеет особое значение: он является связующим звеном между аппаратным и программным обеспечением.

Все разработчики считают, что нужно транслировать программы, написанные на различных языках высокого уровня, в общую промежуточную форму – на уровень архитектуры команд – и соответственно конструировать аппаратное обеспечение, которое может непосредственно выполнять программы этого уровня (уровня архитектуры команд). Уровень архитектуры команд связывает компиляторы и аппаратное обеспечение, это язык, понятный и компиляторам и аппаратному обеспечению (рис. 4.1).

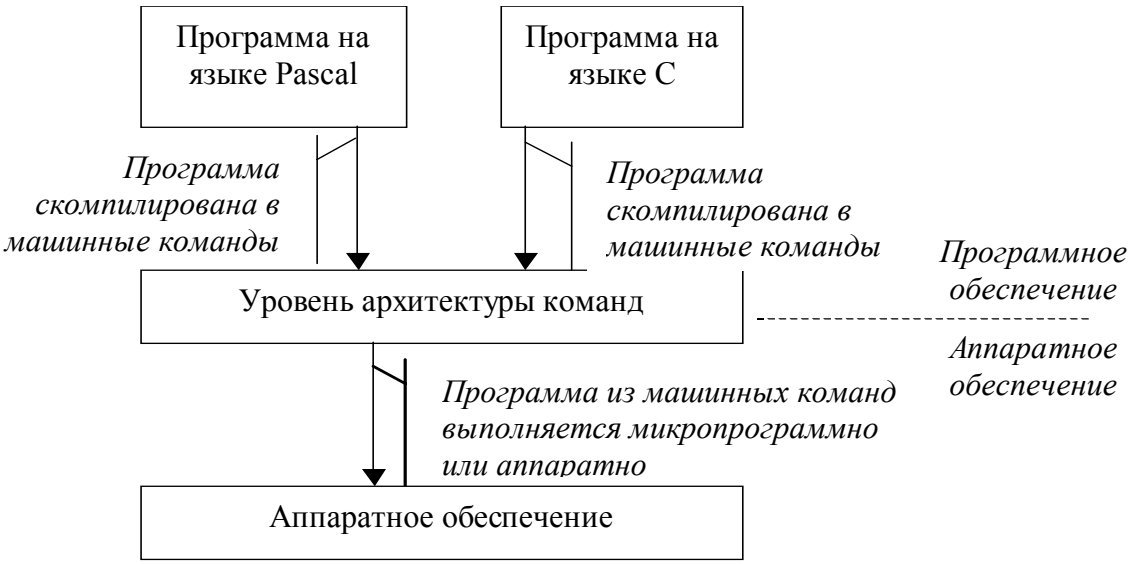


Рис. 4.1

Когда появляется новая машина, первые вопросы, который задает покупатель: «Совместима ли машина с предыдущими версиями?», «Могу ли я установить мою старую операционную систему?» и «Будут ли работать старые программы?». Покупатели редко рвутся выбросить старое ПО и заняться разработкой нового с нуля.

Этот факт заставляет разработчиков сохранять один и тот же уровень команд в разных моделях и обеспечивать совместимость ПО (по крайней мере снизу вверх). Разработчики должны обеспечить совместимость на уровне команд, но они могут делать что угодно с аппаратным обеспечением. Таким образом возникает задача построения лучших машин но совместимых с предыдущими версиями.

Какую архитектуру команд можно считать хорошей?

1. Хорошая архитектура должна определять набор команд, которые можно эффективно реализовать в современной и будущей технике, что приводит к рентабельным разработкам на несколько поколений.
2. Хорошая архитектура команд должна обеспечивать ясную цель для оттранслированной программы. Поскольку уровень архитектуры команд является промежуточным между программной и аппаратной частью, то он должен быть удобен как для разработчиков аппаратного обеспечения, так и для составителей программного обеспечения.

4.1 ОБЩИЙ ОБЗОР УРОВНЯ АРХИТЕКТУРЫ КОМАНД

Свойства уровня команд

Уровень команд – это то, каким представляется компьютер программисту машинного языка. Программа уровня архитектуры команд –

это то, что выдает компилятор. Чтобы произвести программу уровня команд, составитель компилятора должен знать, какая модель памяти используется в компьютере, какие регистры, типы данных и команды имеются в наличии и т.д.

В связи с этим определением такие вопросы как программируется ли микроархитектура или нет, конвейеризирован компьютер или нет, является ли компьютер суперскалярным не относится к уровню архитектуры команд, однако знания об этом могут повысить эффективность разрабатываемых компиляторов.

Модели памяти

Большинство машин имеет единое адресное пространство. В некоторых машинах содержатся отдельные адресные пространства для команд и данных. Такая система гораздо сложнее, чем единое адресное пространство, но зато она имеет два преимущества:

- Появляется возможность иметь 2^{32} байтов для программы и 2^{32} байтов для данных при размере регистра адреса в 32 разряда;
- Исключается сама возможность интерпретировать код как данные и наоборот.

Один из аспектов модели памяти – семантика памяти. Естественно ожидать, что если в программе идет запись по некоторому адресу, а затем считывание из этого же адреса, то мы получим толь что сохраненное значение. Но в некоторых машинах микрокоманды переупорядочиваются. Возникает опасность, что память не будет действовать так, как ожидалось (сопроцессор). Ситуация усложняется в случае с мультипроцессором, когда каждый процессор посылает свой запрос в память. Системные разработчики могут применять различные подходы к решению этих задач. В некоторых случаях запросы в память делают упорядоченными, в других – возлагают на разработчиков компиляторов (или даже программистов) заботу о правильном функционировании памяти. Все эти проблемы затрудняют работу разработчиков ПО, хотя надо отметить, что существуют модели памяти, в которых аппаратное обеспечение автоматически блокирует определенные операции с памятью связанные с зависимостью записи/считывания.

Регистры

Во всех компьютерах имеется несколько регистров, которые видны на уровне архитектуры команд. Они нужны для хранения промежуточных результатов, для контроля выполнения программы и для других целей.

Регистры уровня команд можно разделить на две категории: специальные регистры и регистры общего назначения. Специальные регистры включают счетчик команд, указатель стека, а также другие регистры с особой функцией. Регистры общего назначения содержат ключевые локальные переменные, и промежуточные результаты. Их

основная функция – обеспечить быстрый доступ к часто используемым данным (избегать обращений в память). Машины RISC с высокоскоростным процессором обычно содержат минимум 32 регистра общего назначения, и их количество постоянно растет.

В некоторых машинах все регистры взаимозаменяемы, в некоторых регистры общего назначения могут быть специализированы. Например, в Pentium II регистр EDX может использоваться в качестве регистра общего назначения, но который получает половину произведения и содержит половину делимого при делении.

Кроме регистров, доступных на уровне команд, всегда существует довольно большое количество регистров, доступных только в привилегированном режиме. Эти регистры контролируют различные блоки кэш-памяти, основную память, устройства ввода/вывода и другие элементы аппаратного обеспечения.

Общий обзор уровня команд машины Pentium II

Процессор Pentium II развивался на протяжении многих лет. Основная архитектура команд обеспечивает выполнение программ, написанных для 8086 и 8088, а в машине даже содержатся элементы 8-разрядного процессора 8080. На процессор 8080, в свою очередь сильно повлияли требования совместимости с процессором 8008, который был основан на процессоре 4004 (4-битная схема).

Pentium II имеет 3 операционных режима, в двух из которых он работает как 8086.

Real Address Mode – режим реальной адресации (или просто реальный режим), полностью совместим с 8086. В этом режиме возможна адресация до 1 Мб физической памяти.

Существенным дополнением является Virtual 8086 Mode – режим виртуального процессора 8086. Этот режим является особым состоянием задачи защищенного режима, в котором процессор функционирует как 8086. На одном процессоре в таком режиме могут параллельно выполняться несколько задач с изолированными друг от друга ресурсами. При этом использование физического адресного пространства памяти управляется механизмами сегментации и трансляции страниц. Попытки выполнения недопустимых команд, выхода за пределы отведенного пространства памяти контролируются системой защиты. Когда пользователь WINDOWS начинает работу с MS-DOS, программа, которая действует под DOS, запускается в виртуальном режиме, чтобы программа WINDOWS не могла вмешаться.

Protected Virtual Address Mode – защищенный режим виртуальной адресации (или просто защищенный режим). В этом режиме процессор позволяет адресовать до 4 Гб физической памяти. В этом режиме доступны 4 уровня привилегий.

В Pentium имеется 31 регистр; они подразделяются на 16 регистров прикладного программиста (пользовательские регистры) и 15 регистров системного программиста (системные регистры).

Пользовательские регистры процессора изображены на рис. 4.2.

Регистры общего назначения по сути являются расширением 16-ти разрядных регистров, например EAX есть расширение AX, причем доступ к старым 16-ти разрядным и 8-ми разрядным частям регистра типа (AH) сохраняется. Роль регистров общего назначения сохранилась из 8086 и подробно на них останавливаться не будем.

Сегментные регистры. Их теперь 6, однако новые сегментные регистры FS и GS не выполняют никаких специальных функций, роли регистров CS, DS, SS, ES те же, что и в 8086.

В общем, каждый сегментный регистр определяет сегмент (т.е. блок смежных ячеек) памяти. Если в процессоре 8086 содержимое сегментного регистра прямо задает физический базовый адрес сегмента и максимальный размер всех сегментов составляет 64 Кбайт, то в процессорах 286 и старше содержимое сегментного регистра определяет сегмент косвенно, через так называемую дескрипторную таблицу (descriptor table). С помощью дескрипторной таблицы для каждого сегмента задаются базовый адрес, размер (limit) и право доступа (access rights), показывающие, какие программы и в каких операциях могут обращаться к конкретному сегменту. Производить прямые обращения по физическим адресам памяти программист не может.

Записываемые в сегментный регистр слова (16 бит) называют селекторами (selector), поскольку оно выбирает или «селектирует» один из сегментов из множества возможных.

Указатель команды EIP\IP предназначен для адресации команд внутри текущего сегмента кода.

Регистр флажков EFLAGS\FLAGS содержит флажки, которые подразделяются на восемь флажков состояния и шесть управления. Формат регистра приведен на рис. 4.3 (флажки управления помечены *, а флажки состояния – ^).

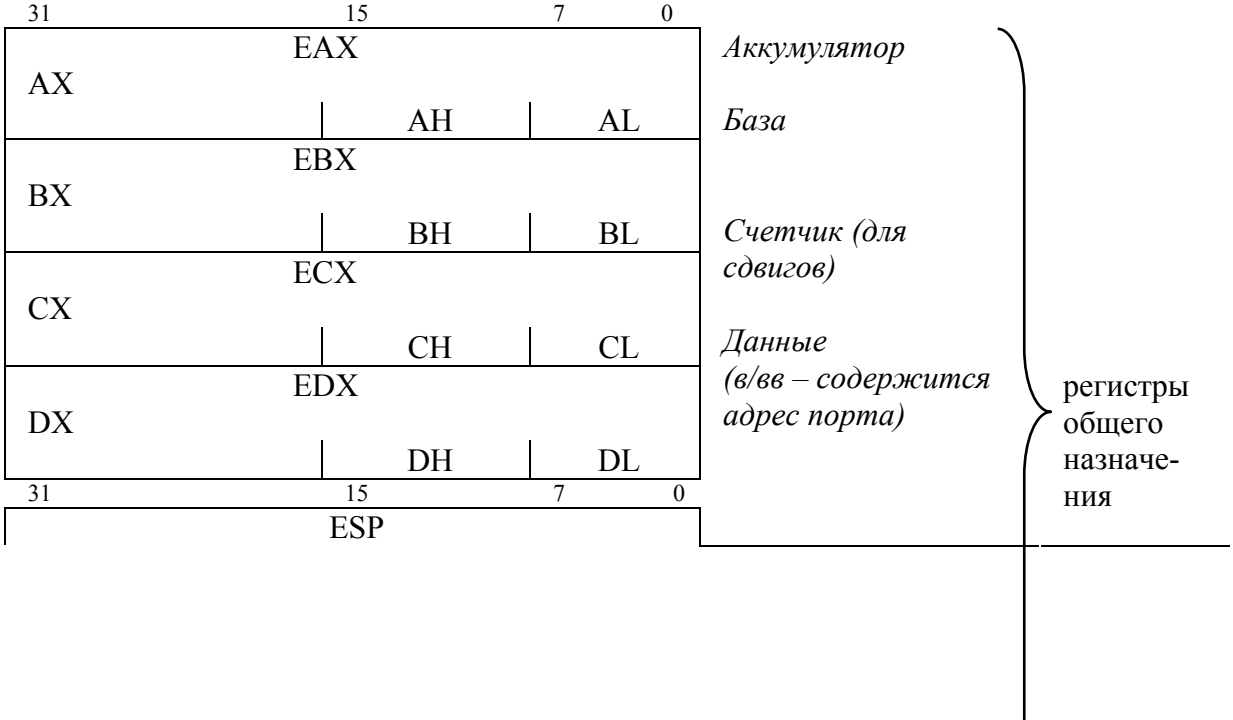


Рис. 4.2

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
0	0	0	0	0	0	0	0	0	0	ID	VIP	VIF	AC	VM	RF
										*	*	*	*	*	*

15	14	13 12	11	10	9	8	7	6	5	4	3	2	1	0
0	NT	IOPL	OF	DF	IF	TF	SF	ZF	0	AF	0	PF	1	CF
	^	^	^	*	*	*	^	^		^		^		^

Рис. 4.3

Биты 31-22, 15, 5, 3, 1 зарезервированы. При записи в эти биты должны устанавливаться в предварительно прочитанное состояние.

Флаги состояния:

- *CF (Carry Flag)* – флажок переноса (бит **0**). Этот бит устанавливается в **единицу**, если арифметическая операция вызвала перенос или заем из старшего бита результата. Программно доступен.
- *PF (Parity Flag)* – флажок четности (бит 2). Флажок устанавливается в единицу, если младшие восемь бит результата содержит четное число единичных бит.
- *AF (Auxiliary carry Flag)* – флажок вспомогательного переноса (бит 4). Устанавливается в единицу, если арифметическая операция вызвала

перенос или заем из младшей тетрады. Главное применение флажка связано с десятичной арифметикой. Программно не доступен.

- *ZF (Zero Flag)* – флажок нуля (бит 6). Устанавливается в единицу, если результат операции равен нулю.
- *SF (Sing Flag)* – флажок знака (бит 7). Большинство команд устанавливает этот флажок в тоже состояние, что и старший бит результата.
- *OF (Overflow Flag)* – флажок переполнения (бит 11). Устанавливается в единицу, если в арифметических операциях со знаковыми операндами результат превышает допустимый диапазон.
- *IOPL(Input / Output Privilege Level)* – уровень привилегий ввода\вывода (биты 12 и 13). Это поле содержит два бита и показывает минимальный уровень привилегий для данной задачи. Более подробно будет рассмотрено в теме о системе защиты.
- *NT (Nested Flag)* – флажок вложенной задачи (бит 14). Более подробно будет рассматриваться при изучении многозадачности. Используется для для контроля за прерванными задачами и при вызове процедур.

Флаги управления.

- *TF (Trap Flag или Trace Flag)* – флажок покомандной работы (бит **8**). Если он установлен в 1, то после выполнения каждой команды генерируется внутреннее прерывание. Используется для отладки программ.
- *IF (Interrupt Flag)* – флажок прерывания (бит **9**). Если он установлен в единицу, то внешние прерывания разрешены по входу INTR. Флажок не влияет на прерывания по входу NMI.
- *DF (Direction Flag)* – флажок направления (бит **10**). Задает направление обработки цепочек. Доступен программисту.
- *RF (Resume Flag)* – флажок возобновления (бит **16**). Если флажок установлен в 1 можно маскировать некоторые особые случаи в отладке программы.
- *VM (Virtual Mode)* – флажок виртуального режима (бит **17**).
- *AC (Alignment Check)* – флажок контроля выравнивания (бит **18**). Если от установлен в 1, то разрешен контроль выравнивания. Рассматривать не будем.
- V*IF (Virtual Interrupt Flag)* – флаг виртуального прерывания (бит 19, Pentium ...). Используется в специальном расширенном режиме обработки прерываний, является виртуальным подобием флага IF. Применяется совместно с флагом VIP и позволяет обеспечивать нормальное функционирование устаревшего ПО.
- *VIP ((Virtual Interrupt Pending)* – ожидание виртуального прерывания (бит **20** Pentium ..).
- *ID (Id Flag)* – флаг доступности команды идентификации CPUID (бит 21, Pentium ...). Если программа или процедура способна установить или сбросить этот флаг, значит команда CPUID поддерживается.

Системные регистры **GDTR, LDTR, IDTR** предназначены для хранения базовых адресов таблиц дескрипторов. Их еще называют р*егистрами управления памятью.* Они определяют местонахождение структур данных, управляющих сегментацией памяти. Регистр задачи **TR** применяется для слежения за тем, какая задача выполняется процессором в данный момент.

Для доступа к системным регистрам используются команды:
LGDT, SGDT, LLDT, SLDT, LIDT, SIDT.

Регистр глобальной дескрипторной таблицы GDTR имеет длину 48 байт. Содержит 32 битный (для 286 –24-битный) базовый адрес и 16-битный предел.

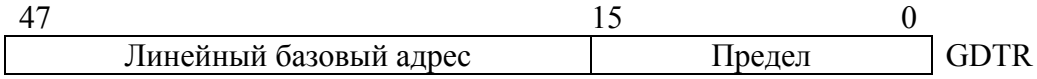


Рис. 4.4

Регистр локальной дескрипторной таблицы LDTR Видимая часть (которую можно программно читать и изменять) 10-байтного регистра содержит 16-разрядный селектор дескриптора LDT. Сам дескриптор LDT (базовый адрес, предел, атрибуты) автоматически загружается в скрытую часть LDTR из глобальной таблицы дескрипторов. Формат LDTR:

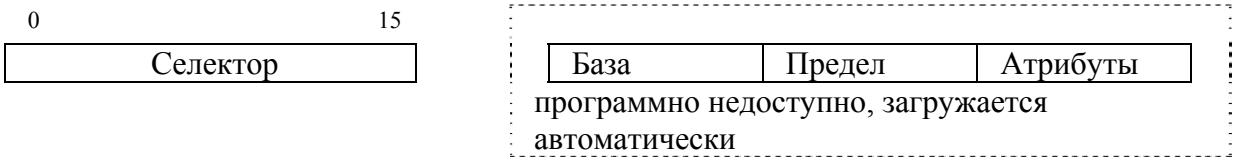


Рис. 4.4

Регистр таблицы дескрипторов прерываний содержит 32 битный (для 286 –24-битный) базовый адрес и 16-битный предел. Формат регистра:

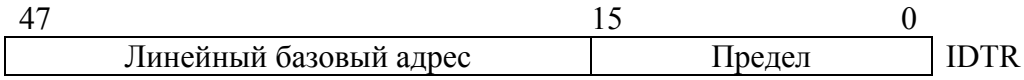


Рис. 4.5

Регистр задачи TR. Видимая часть регистра содержит 16-битный селектор дескриптора сегмента состояния задачи (TSS). Сам дескриптор TSS загружается автоматически в скрытую часть TR из глобальной таблицы дескрипторов. Формат регистра задачи совпадает с форматом регистра локальной дескрипторной таблицы.

Управляющие регистры впервые были использованы в 386 микропроцессоре. Ранее (286) было так называемое слово состояния машины, которое для сохранения совместимости полностью помещено в

управляющий регистр CR0. Практически все режимы и специальные возможности микропроцессора устанавливаются определенными флагами в управляющих регистрах.

Доступ к управляющим регистрам возможен с помощью команд

MOV CR, SMSW, LMSW, CLTS (очистить флаг переключени якоманд).

Регистр CR0. Первые 16 бит – слово состояния машины. Биты NE, WP, AM, GW, CD были введены в регистр начиная с Intel486.

31	30	29	28 19	18	17	16	15 6	5	4	3	2	1	0
PG	CD	GW	*.....*	AM	*	WP	* ... *	NE	ET	TS	EM	MP	PE
							MSW (Слово состояния машины I286)						

Рис. 4.6

PE (разрешение защиты, бит 0) – предназначен для переключения процессора (реальный/защищенный режим). PE=0 –режим реальной адресации, PE=1 –защищенный режим.

MP (слежение за сопроцессором, бит 1) – Если сопроцессор не установлен, то этот бит должен быть равен 0. Для процессоров со встроенным сопроцессором бит должен быть установлен в 1.

EM (эмуляция сопроцессора, бит 2) – предназначен для генерации особой ситуации для всех команд сопроцессора. При отсутствии сопроцессора флаг должен быть установлен. Для выполнения команд MMX – сброшен.

TS (задача переключена, бит 3) –устанавливается при каждом переключении задачи. Вызывает ситуацию исключения при выполнении команд сопроцессора, MMX/3Dnow!, SIMD. Предназначен для исключения выполнения этих команд над данными, оставшимися от других задач. Команда CLTS очищает этот бит.

ET (тип сопроцессора, Intel 386, 486) – при ET =1 используется 32-битный протокол, а при ET=0 16 битный. В процессорах Pentium считается зарезервированным.

NE (ошибка сопроцессора, бит 5, Intel486) – в зависимости от того, как установлен этот бит происходит обработка ошибок сопроцессора: либо как внешнее прерывание, либо как внутреннее.

Механизмы реакции различных моделей процессоров и сопроцессоров на возникновение исключительных прерываний могут изменяться для различных моделей. (СМ. тех. описание)

WP (защита записи, бит 16, Intel 486 ...) – защищает страницы пользовательского уровня (определенные только для чтения) от записи программой–супервизором, работающей на более высоком уровне привилегий. Если WP=0, то супервизор может осуществлять запись в пользовательские страницы, защищенные от записи.

AM (маска выравнивания, бит 18, Intel486 ...) – если бит сброшен – контроль запрещен, а если установлен (совместно с флагом AC), то генерируется особая ситуация контроля выравнивания.

NW (запрет сквозной записи, бит 29, Intel486 ...) – предназначен для управления встроенной кэш-памятью.

CD (разрешение работы кэш-памяти, бит 30, Intel486 ...) – если CB =0, то работа внутренней кэш-памяти разрешена, в противном случае – запрещена.

PG (включение страничного механизма, бит 31, Intel386...) – если PG=1, то страничный механизм включен, иначе –выключен. Используется только в защищенном режиме. Если этот бит установить в реальном режиме, то будет генерироваться ошибка общей защиты.

Регистр CR1.– зарезервирован.

Регистр CR2. если включен механизм страничной адресации и генерируется особая ситуация (страничная ошибка), то в этом регистре будет полный 32-разрядный адрес, поступление которого в блок страничной адресации вызвало ошибку.

Регистр CR3. При включенном страничном механизме (CR4.PAE=0) регистр CR3 (20 старших бит) содержат 20 старших бит физического адреса каталога страниц. Младшие 12 принимаются равными нулю. Начиная с процессора 486 используются еще два бита этого регистра (PWT и PCD), управляющие кэшированием. В процессорах (Pentium ...), которые поддерживают расширение физического адреса (CR4.PAE=1), в режиме расширенного адреса уже 27 разрядов содержат адрес таблицы указателей на каталоги страниц. Младшие 5 разрядов полагают равными 0.

PWT (сквозная запись страниц, бит 3, Intel486 ...) – используется для управления кэшированием страниц.

PCD (запрещение кэширование страниц, бит 4, Intel486 ...) – используется для управления кэшированием текущего каталога страниц.

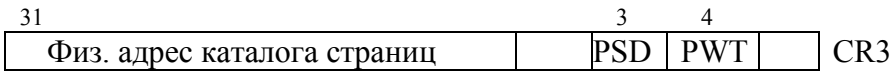


Рис. 4.7

Регистр CR4. Обеспечивает включение/выключение самых разнообразных режимов и дополнительных возможностей процессора. Впервые был использован в процессоре Pentium. Различные модификации процессора обеспечивают различные наборы возможностей. Поэтому перед программированием любых битов этого регистра следует с помощью команды CPUID проверить наличие возможностей для конкретной модификации процессора.

WME (расширение виртуального 8086 режима, бит 0, Pentium ...)

PVI (виртуальные прерывания защищенного режима, бит 1, Pentium ...)

TSD (ограничение маркера времени бит 2, Pentium ...)

DE (расширение отладки бит 3, Pentium ...)

PSE (расширение размера страниц, бит 4, Pentium ...) Если бит установлен в 1, то разрешает использование страниц расширенного размера (4 или 2 Мб) в зависимости от текущей разрядности физического адреса. В противном случае размер страниц ограничен 4 Кб.

PAE (расширение физического адреса, бит 5, Pentium Pro ...) –если бит установлен в 1, то разрешено использование 36-битной физической адресации.

MCE (расширение контроля машины, бит 6, Pentium ...)

PGE (разрешение глобальных страниц, бит 7, Pentium Pro...)

PCE (разрешение счетчика производительности, бит 8, Pentium Pro ...) – предназначен для разрешения мониторинга производительности в пользовательских программах.

OSFXSR (использование команд быстрого сохранения/восстановления, бит 9, Pentium II...) используется для быстрого восстановления/сохранения состояния FPU/MMX/SIMD.

OSXMMEXCPT (разрешение особой ситуации SIMD, бит 10, Pentium III...) – управляет реакцией процессора на SIMD-исключения.

Отладочные регистры

Регистры отладки DR0 –DR3 содержат линейные адреса четырех точек останова. Регистры DR4, DR5 – зарезервированы.

DR6 содержит информацию о том, какая из точек останова вызвала особую ситуацию отладки.

DR7 устанавливает режимы срабатывания точек отладки; служат целям отладки.

Регистры TR6, TR7 для работы с кэш–памятью.

Регистры FPU

Первые процессоры архитектуры x86 не поддерживали напрямую вычисления с плавающей точкой. Фирмой Intel выпускались отдельные устройства (микросхемы), именуемые сопроцессором (8087, 80287, 80387), реализующие такие вычисления. Сейчас FPU встраивается в сам процессор. Тем не менее, поскольку в x86 сопроцессор представляет собой отдельное устройство, необходимо представлять специфику его работы.

Сопроцессор подключается к системной шине параллельно с центральным (основным) процессором и может работать только с ним. Он не имеет своей индивидуальной программы и не может осуществлять выборку команд. Команды сопроцессора находятся в общем потоке команд. Если выбранная команда является командой центрального процессора, он выполняет ее обычным образом, а сопроцессор не привлекается. Когда выбирается команда сопроцессора, действие центрального процессора

зависит от специфики конкретной команды. Если команда не связана с обращением к памяти, ЦП ее игнорирует, и переходит к следующей команде. Но если команда требует обращения к памяти, ЦП вычисляет адрес операнда и обращается к памяти, но адрес и данные «перехватывает» сопроцессор. После этого сопроцессор реализует конкретные действия по выполнению команды. Они могут производиться параллельно с дальнейшими действиями центрального процессора. Этот параллелизм требует некоторой синхронизации действий ЦП и сопроцессора.

Синхронизация по командам. Когда ЦП выбирает для выполнения команду сопроцессора, последний может быть занят выполнением своей предыдущей команды. Следовательно, перед каждой командой сопроцессора должна находиться специальная команда ЦП, которая только проверяет состояние сопроцессора и, если он занят, переводит ЦП в состояние ожидания (эти команды имеют мнемоники WAIT, FWAIT). Обычно АССЕМБЛЕР вставляет эти команды автоматически).

Синхронизация данных. Если выполняемая сопроцессором команда записывает операнд в ячейку памяти перед последующей командой центрального процессора, которая обращается к той же ячейке, также необходима команда проверки состояния сопроцессора. АССЕМБЛЕР учесть такие ситуации не может, поэтому ответственность за их использование ложится на программиста.

FPU x86 опирается на стековую организацию, особенности и преимущества которой уже обсуждались. Программная модель FPU представлена на рис. 4.8.

Основу программной модели составляет регистровый стек из 8 80-разрядных регистров R0–R7, называемых также численными или арифметическими регистрами. В них хранятся числа в так называемом расширенном вещественном формате. В любой момент трехразрядное поле TOP (top of stack) в слове состояния SW определяет регистр, являющейся вершиной стека и обозначаемый ST(0) или ST. Пример приведен на рис. 4.9. Здесь вершина стека R3 (ST(0)), регистр R4 –это ST(1), регистр R2 в самом низу стека и обозначается ST(7). Команды в FPU рассчитаны на такую относительную адресацию.

Операция загрузки в стек декремент поля TOP и загружает данные в новую вершину, извлечение из стека действует наоборот.

Особенностями этого регистрового стека является следующее: стек имеет кольцевую структуру; контроль использования стека должен осуществлять сам программист; в командах допускается явное и неявное обращение к регистрам стека с модификацией и без модификации поля TOP.

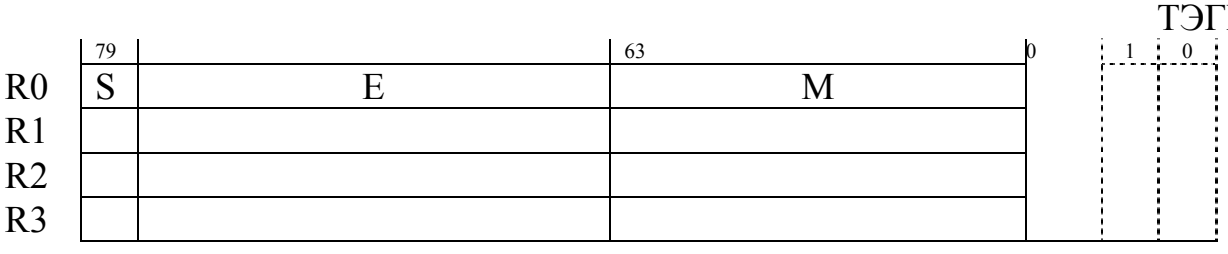


Рис. 4.8

С каждым регистром стека ассоциируется двухбитный тег (признак) совокупность которых образует слово тегов TW. Тег регистра R0 находится в младших битах этого слова, а тег регистра R7 – в старших. Тег в общем виде показывает, что содержится в соответствующем регистре:

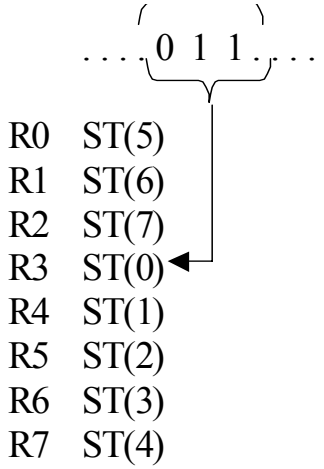


Рис. 4.9

- 00 – допустимое ненулевое число;
- 01 – нуль;
- 10 – специальное значение (о них позже);
- 11 – пустой регистр.

При своей инициализации FPU записывает 11 во все теги регистров, отмечая регистры как пустые. Попытка обращения к такому регистру вызывает особый случай (прерывание). Регистры CW, SW будут рассмотрены позже; регистры IP и DP используются в процедурах при обработке особых случаев.

Устройство FPU имеет два программно–доступных 16-разрядных регистра, содержимое которых определяет режим его работы и текущее состояние. Регистры представлены на рис. 4.10.

	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
CW	x	x	x	x	R	C	P	C	x	x	PM	UM	OM	ZM	DM	IM
SW	B	C3	T	O	P	C2	C1	C0	ES	SF	PE	UE	OE	ZE	DE	IE

Рис.4.10

*Слово управления CW.*Определяет для FPU варианты обработки численных данных. Шесть младших разрядов представляют собой индивидуальные маски особых случаев, т.е. необычных и ошибочных ситуаций, обнаруживаемых устройством FPU при выполнении программы. К битам маски относятся:

- PM – маска точности (истинный результат операции не представим в формате получателя;
- UM – маска анти переполнения;
- OM – маска переполнения:
- ZM – маска деления на ноль;
- DM – маска денормализованного операнда;
- ID – маска недействительной функции.

Если любой из этих бит установлен в состояние 1, то возникновение соответствующего особого случая не будет вызывать прерывания процессора; в противном случае – будет.

Поясним понятие недействительной операции — сюда могут относиться корень квадратный из отрицательного числа и т.д.

Управление точностью. Поле PC (2 байта) определяет точность вычислений 24 бита (PC = 00), 53 бита (10) и 64 бита (11). По умолчанию используется 64 бита.

Управление округлением. Устройство FPU имеет четыре режима округления, выбираемые полем RC слова управления:

00 – округление к ближнему;

01 – округление вниз (к $-\infty$);

10 – округление вверх (к $+\infty$);

11 – усечение (к 0).

По умолчанию принимается округление к ближнему.

Слово состояния SW. Занимает 16 бит и отражает общее состояние устройства FPU. В слове состояния SW младшие 6 бит отведены для регистрации особых случаев; назначение этих бит аналогично слову CW. При обнаружении любого из незамаскированных особых случаев устройство FPU устанавливает в состояние 1 соответствующий флажок и вызывает прерывание ЦП.

Бит SF нарушения стека предназначен для того, чтобы регистрировать ошибки в работе регистрового стека. Устройство FPU устанавливает его в состояние 1, если команда вызывает переполнение стека (особый случай недействительной операции).

Биты C0–C3 кода условия похожи на арифметические флажки центрального процессора в регистре EFLAGS. Они регистрируют особенности результатов команд сравнения. Эти биты, в основном, применяются для условных переходов. Команда FSTSW AX передает слово состояния в регистр AX. Затем командой SAHF модно переместить биты C0 –C3 и биты флажков.

Трехадресное поле вершины стека TOP показывает, какой из внутренних регистров данных является текущей вершиной стека.

Бит B занятости служит для совместимости с 8087 (совпадает с ES).

Технология MMX

Технология MMX ориентирована на приложения мультимедиа, 2D/3D-графику и коммуникации. Это расширение базовой архитектуры

появилось только после второго поколения процессоров Pentium. Основная идея MMX заключается в одновременной обработке нескольких элементов данных за одну инструкцию —технология SIMD (Single Instruction–Multiple Data). Расширение MMX использует новые типы упакованных целочисленных данных:

- упакованные байты (Packed Byte) –восемь байт;
- упакованные слова (Packed word)–четыре слова;
- упакованные двойные слова (Packed doubleword)–два двойных слова;
- учетверенное слово (Quadword)–одно слово.

Эти типы данных специальным образом могут обрабатываться в регистрах MMX0–MMX7, представляющие собой младшие биты стека 80–битных регистров FPU. Эти регистры, также как и регистры FPU не могут быть использованы для адресации памяти. Совпадение регистров MMX и FPU накладывает ограничения на чередование кодов MMX и FPU. Частое чредование кодов MMX и FPU снижает производительность за счет необходимости сохранения и восстановления весьма объемного контекста FPU – и забота об этом лежит на программисте.

Еще одна особенность технологии MMX – поддержка *арифметики с насыщением* (при переполнении фиксируется максимально/минимально допустимое число). Граничные значения определяются типом данным.

В систему команд введено дополнительно 57 инструкций для одновременной обработки нескольких единиц данных. Одновременно обрабатываемое 64-битное слово может содержать как одну единицу обработки, так и 8 однобайтных, 4 двухбайтных или 2 четырехбайтных операнда.

Новые инструкции включают следующие группы:

- арифметические (+,- в различных режимах, умножение, комбинацию умножения и сложения);
- сравнение элементов данных;
- преобразование форматив;
- логические, выполняемые на 64-разрядными кодами;
- сдвиги (арифметические и логические)
- пересылки данных между регистрами MMX и целочисленными регистрами и памятью;
- очистка MMX – установка признаков пустых регистров в слове TEГов.

Регистры MMX в отличие от регистров FPU адресуются физически, а не относительно указателя стека TOP. Любая инструкция MMX обнуляет поле TOP регистра состояния FPU. В слове TEГов свободному регистру соответствует комбинация 11, а все остальные комбинации соответствуют занятому регистру. После каждой операции MMX биты TEГов используемого регистра обнуляются. Неиспользуемые в MMX биты (79-64) регистров FPU заполняются единицами, поэтому ошибочное использование данных MMX инструкциями FPU приведет к исключению. Команда EMMS

предназначена для предназначена для очистки контекста MMX т.е. помечает все регистры как пустые.

Наличие поддержки MMX определяется по биту 23 регистра EDX после вызова инструкции CPUID(1).

Технология 3DNow! введена фирмой AMD в процессорах K6–2, расширяет возможности MMX. Она позволяет оперировать с новым типом данных – парой упакованных чисел в формате с плавающей точкой. Эти числа занимают по двойному слову в 64-разрядных регистрах MMX. Процессор K6–2 имеет два исполнительных блока, которые способны одновременно выполнять операции с плавающей точкой над своими регистрами. Каждая такая операция занимает всего два такта, операции полностью конвейеризируются. Таким образом, с конвейеров процессора за каждый такт могут сходить четыре результата операций с плавающей точкой. В систему команд добавлена 21 новая инструкция MMX, большая часть которых предназначена для обработки упакованных чисел с плавающей точкой. Имеется также новая целочисленная MMX инструкция для усреднения 8 пар 8-битных чисел. Имеется команда быстрого переключения FPU-MMX. Технология 3Dnow! программно совместима с прежними процессорами и операционными системами, поскольку регистры MMX отображаются на регистры FPU и для них работают механизмы сохранения контекста в многозадачных ОС.

Поддержка 3Dnow! определяется по биту 31 регистра EDX после вызова CPUID(8000_0001h).

Регистры SIMD

В процессорах, начиная с Pentium III, возможно использование специального архитектурного расширения, предназначенного для потоковой обработки данных. Это расширение включает дополнительный набор команд, дополнительные режимы генерации ошибок и исключений, дополнительные внутренние регистры и форматы данных, воспринимаемые процессором. Данное расширение – усовершенствование команд мультимедийной обработки MMX.

Доступ ко всем SIMD–регистрам возможен только с помощью новых SIMD–команд. Перед использованием данного расширения ПО должно с помощью команды CPUID проверить, поддерживается ли оно конкретной моделью микропроцессора.

Регистры общего назначения SIMD Восемь регистров общего назначения (XMM0 …XMM7) вводятся в процессор Pentium III … для использования командами потоковой обработки. В отличие от MMX-регистров, эти регистры не являются отображаемыми – это новые физические регистры, доступ к которым осуществляется с помощью новых команд. Поэтому при совместном использовании команд SIMD, FPU, MMX не возникает никаких проблем. Регистры XMM0 … XMM7 могут использоваться только для хранения данных, но не для косвенной адресации.

Общий обзор архитектуры уровня команд системы UltraSPARC II

Архитектура SPARC была впервые введена в 1987 г. и была первой архитектурой промышленного назначения типа RISC. Изначально она была 32-разрядной, но UltraSPARC II – это 64-разрядная машина.

Структура памяти проста: память представляет собой линейный массив из 2^{64} байтов. Память настолько велика, что в настоящее время она не реализуема. Современные реализации имеют размер адресного пространства 2^{44} (UltraSPARC II), но в будущем память будет увеличиваться.

Одна из проблем разработки архитектуры заключается в том, что архитектура команд ограничивает размер адресуемой памяти.

Архитектура команд SPARC достаточно проста, хотя организация регистров усложнена для повышения эффективности вызова процедур.

В системе UltraSPARC имеется две группы регистров: 32 64-битных регистров общего назначения и 32 регистра с плавающей точкой. Варианты названий регистров приведены в таблице 4.1.

Таблица 4.1

Регистр	**Вариант названия**	Функция
R0	GO	Связан с 0. Все, что сохраняется в этом регистре, просто игнорируется
R1- R7	G1—G7	Содержит глобальные переменные
R8 – R13	O0 – O5	Содержит параметры вызываемой процедуры
R14	SP	Указатель стека
R15	O7	Временный регистр
R16 – R23	L0 – L7	Содержит локальные переменные для текущей процедуры
R24 – R29	I0 – I5	Содержит входные параметры
R30	FP	Указатель на основу текущего стекового фрейма
R31	I7	Адрес возврата для текущей процедуры

В действительности процессор UltraSPARC имеет более 32 регистров общего назначения, но видимы для программиста только 32 в любой момент времени. Эта особенность, называемая «регистровыми окнами», предназначена для повышения эффективности вызова процедур. Идея состоит в том, что имеется несколько наборов регистров, точно также, как существует несколько фреймов в стеке. Ровно 32 регистра видны в текущий момент; регистр CWP (Current Window Pointer – указатель текущего окна) следит за тем, какой набор регистров используется в данный момент.

Команда вызова процедуры скрывает старый набор регистров и предоставляет новый набор, который может использовать вызванная процедура. Однако некоторые регистры из старого набора переносятся в новый. Система эффективна, если нет многократных вложений.

В системе UltraSPARC II также есть 32 разряда с плавающей точкой, которые могут содержать либо 32-битные (одинарная точность), либо 64-

битные (двойная точность) значения. Предусмотрена возможность использования пары регистров для поддержания 128-битных значений.

Архитектура UltraSPARC II – это архитектура загрузки/хранения. Это значит, что единственные операции, которые обращаются в память – это операции записи и чтения, служащие для передачи данных между регистрами и памятью. Все операнды для команд арифметических действий должны браться из регистров или предоставляться самой командой, а все результаты должны сохраняться в регистрах.

Общий обзор виртуальной машины Java

Уровень команд машины Java необычен, но достаточно прост.

Память содержит 4 основные области: фрейм локальных переменных, стек операндов, область процедур и набор констант.

Фрейм локальных переменных – эта область предназначена для хранения переменных во время выполнения процедур. В начале этого фрейма располагаются параметры (или аргументы) вызванной процедуры. Фрейм локальных переменных не включает в себя стек операндов. Он помещается отдельно. Существует неявный регистр, который содержит адрес первой переменной фрейма – LV (Local Variable).

Стек операндов – не должен превышать определенный размер, который заранее вычисляется компилятором Java. Пространство стека операндов располагается прямо над фреймом локальных переменных. Существует виртуальный регистр, который содержит адрес верхнего слова стека. Содержимое этого регистра меняется во время выполнения процедуры, поскольку операнды помещаются в стек и выталкиваются из него (Регистр SP).

Область процедур – область памяти, в которой содержится программа. Существует виртуальный регистр, содержащий адрес команды, которая будет вызвана следующей. Этот указатель называется счетчиком команд PC (Program Counter).Область процедур представляет собой массив байтов.

Набор констант – эта область памяти состоит из констант, цепочек и указателей на другие области памяти, на которые можно сделать ссылку. Эта область загружается в тот момент, когда программа загружается из памяти и после этого не меняется. Существует неявный регистр CPP (Constant Pool Pointer – указатель набора констант), который содержит адрес первого слова набора констант.

Доступ в память осуществляется только по смещению от одного из этих регистров. Ни одна из областей памяти не может быть очень большой. Области памяти процедур, локальных переменных и констант не превышает 64 Кбайт. Это определяется тем, что смещения для индексирования ограничивается 16 битами.

В машине нет регистров общего назначения, которые могут загружаться или сохраняться под управлением программы. Это машина со стековой организацией.

У машин со стековой организацией есть один недостаток: требуется большое количество обращений в память. Для преодоления этого недостатка используется свертывание команд (рассмотрели ранее). Кроме того, ввиду того, что архитектура проста, то для ее реализации требуется сравнительно небольшое количество вентилей. Поэтому на кристалле остается место, куда можно поместить кэш-память первого уровня, что, в перспективе, сократит количество обращений в основную память. Однако это пожелание находится в противоречии со стремлением иметь дешевую микросхему.

4.2 ТИПЫ ДАННЫХ

С точки зрения архитектуры ключевым вопросом является вопрос о том, осуществляется ли аппаратная поддержка конкретного типа данных. Под аппаратной поддержкой подразумевается, что одна или несколько команд ожидают данные в определенном формате и пользователь не может брать другой формат.

Типы данных процессора Pentium II

Тип	8 бит.	16 бит.	32 бита	64 бита	128 бита
Целые числа со знаком	*	*	*		
Целые числа без знака	*	*	*		
Двоично-десятичные целые числа	*				
Числа с плавающей точкой			*	*	

Pentium II также может манипулировать 8-ми разрядными символами ASCII: существуют специальные команды для копирования и поиска цепочки символов. Эти команды используются и для цепочек, длина которых известна заранее, и для цепочек, в конце которых стоит специальный маркер. Эти операции часто используются в библиотеках операций над строками.

Операнды не обязательно должны быть выровнены в памяти, но если адреса слов кратны 4 (выравнивание по словам), то производительность повышается.

Типы данных FPU

Устройство FPU определяет численные данные в 7-ми внешних форматах, образуя при этом три класса:

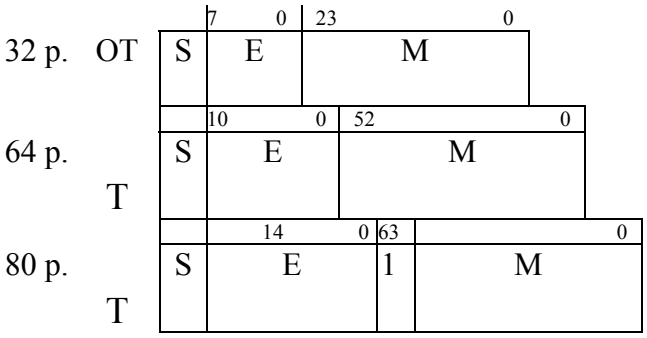
- двоичные числа – целые числа (16, 32, 64 разряда);
- упакованные десятичные целые числа;
- двоичные вещественные числа – 32, 64, 80 разрядов.

Целые числа представляются в дополнительном коде и могут занимать 16 бит (целое слово), 32 бита (короткое целое), 64 бита (длинное целое). Эти

форматы существуют только в памяти, внутри FPU они автоматически преобразуются в 86-разрядный расширяемый вещественный формат.

Упакованные десятичные целые. Они представляются в прямом коде и упакованном формате, т.е. каждый байт содержит две десятичные цифры. Старший бит старшего разряда байта отведен для знака числа, остальные биты этого байта игнорируются. Длина этого формата –10 байт (80 бит). Опять же этот формат существует только в памяти.

Вещественные числа. Имеется три формата с плавающей точкой (одинарной точности ОТ, двойной точности ДТ, расширенной точности РТ). Значащие числа находятся в поле мантиссы, поле порядка показывает фактическое положение десятичной точки в разрядах мантиссы, а бит знака определяет знак числа (рис.4.11).



		M=24	E=8	E=+127; -126	См=127
		M=53	E=11	E=+1023;-1022	См=1023
		M=64	E=15	E=+16383;-16382	См=16383

Рис. 4.11

Порядок E задается в смещенной форме, он равен истинному порядку, увеличенному на значение смещения.

E = истинный порядок + смещение.

Значение числа с плавающей точкой равно:

$$(-1)^{S} \bullet 2^{\text{E-смещение}} \bullet (F0).(F1)(F2) \ldots (Fn),$$

где n = 23, 52, 63.

Устройство FPU обычно поддерживает представление мантиссы в нормализованной форме, т.е. старший бит равен 1. Следовательно, за исключением числа нуль мантисса состоит из целой части и дроби в следующем виде: 1.(F1)(F2)(F3) … (Fn), где Fi = 0 или 1.

В форматах ОТ и ДТ бит F0 при передаче чисел и хранении их в памяти не фигурирует. Это т.н. скрытый и неявный бит, который содержит 1. Следовательно, в этих форматах невозможно представить ненормализованные числа (за некоторым исключением). Кроме того, скрытый бит не позволяет представить в этих форматах нуль, и он должен кодироваться как специальное значение.

Числа в формате РТ имеют явный бит F0. Напомним, что внутри FPU числа хранятся только в этом формате. Перевод в другие форматы из них осуществляется автоматически.

В качестве примера рассмотрим представление числа –247.375 в вещественных форматах FPU.

-247.375 =>-11110111.011

Определяем истинный порядок. Он равен –7.

Смещенный порядок:

- *для обычной точности: 7+127=134 ➔ 1000 0110*
- *для двойной точности: 7+1023=1030 ➔100 0000 0110*
- *для расширенной точности: 7+16383=16390 ➔ 100 0000 0000 0110.*

Значения чисел:

- *OT: 1 1000 0110 1110 1110 0110 00 ....00*
- *ДТ: 1 100 0000 0110 1110 1110 0110 00 ....00*
- *РТ:1 110 0000 0000 0110 1111 0111 0011 00…00.*

Специальные значения. Форматы чисел допускают представление специальных объектов и значений. Для их представления зарезервированы значения смещенного порядка 000 … 0 и 111 … 1.

Денормализованные вещественные числа – это число, которое меньше минимального нормализированного числа для каждого вещественного формата. Такие числа имеют минимальный смещенный порядок 000 … 0 и ненулевую мантиссу, бит (F0) явный или неявный считается нулевым. Введены потому, что иногда промежуточные результаты вычислений могут принимать очень малые значения.

Истинный нуль. Нуль кодируется нулевым смещенным порядком и мантиссой. Нуль является знаковым (положительные или отрицательные нули).

Бесконечность. Поддерживается знаковое представление бесконечности. Это значение кодируется со смещенным порядком из всех единиц и мантиссой 1.000… . Знаки бесконечностей учитываются и сравнение возможно. Бесконечности интерпретируются так:

$$-\infty < X < +\infty .$$

Нечисла. Обозначаются NaN (Not-a-Number). Оно имеет смещенный порядок из всех единиц, любой знак и любую мантиссу, за исключением 1.000… 0. В свою очередь могут делиться на типы. Возникают иногда как результат недействительных операций. Могут использоваться для отладки программ.

Неподдерживаемые форматы. Формат РТ имеет ряд двоичных наборов, не попадающих в рассмотренные классы. Они представляют устаревшие форматы, использовавшиеся в сопроцессоре 8087.

Типы данных машины UltraSPARC II

Тип	**8 бит.**	**16 бит.**	**32 бита**	**64 бита**	**128 бита**
Целые числа со знаком	*	*	*	*	
Целые числа без знака	*	*	*	*	
Двоично-десятичные целые числа					
Числа с плавающей точкой			*	*	*

Все операнды должны быть выровнены в памяти.

UltraSPARC представляет собой регистровую структуру, и почти все команды оперируют 64-разрядными регистрами. Символьные и строковые типы данных специальными командами аппаратного обеспечения не поддерживаются.

Типы данных виртуальной машины Java

Java – язык со строгим контролем данных. Это значит, что каждый операнд имеет особый тип и размер, который известен в период компиляции. Это отражено и в типах данных.

Тип	**8 бит.**	**16 бит.**	**32 бита**	**64 бита**	**128 бита**
Целые числа со знаком	*	*	*		
Целые числа без знака					
Двоично-десятичные целые числа					
Числа с плавающей точкой			*	*	

JVM поддерживает символы, но не традиционные 8-ббитные символы ASCII, а 16-битные символы UNICODE.

4.3 ФОРМАТЫ КОМАНД

Команда – представляет собой код, определяющий операцию вычислительной машины и данные, участвующие в операции. Команда содержит также в явной или неявной форме информацию об адресе, по которому помещается результат операции и адрес следующей команды.

Код команды можно представить состоящим из нескольких частей или полей. Команда в общем случае состоит из операционной и адресной частей.

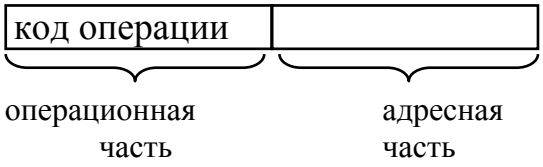


Рис. 4.12

Эти части, в свою очередь, могут состоять из нескольких полей.

Операционная часть содержит код операции (КОП), который задает вид выполняемой операции. Адресная часть команды содержит информацию об адресах операндов и результата операции, а в некоторых случаях информацию об адресе следующей команды.

Структура команды определяется составом, назначением и расположением полей в команде. Форматом команды называют ее структуру с разметкой номеров разрядов (бит), определяющих границы отдельных полей команды, или с указанием числа бит в определенных полях.

Рассмотрим некоторые классические структуры команд.

Чтобы команда содержала в явном виде всю необходимую информацию о задаваемой операции, она должна содержать код операции и четыре поля адреса для указания ячеек памяти, содержащих два операнда, ячейки, в которую помещается результат операции и адрес следующей команды. Такой порядок выполнения называется принудительным, а приведенный формат команды является труднореализуемым и неэффективным и в настоящее время не применяется.

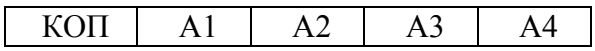


Рис. 4.13

Очевидно, что в большинстве машин и большинстве случаев за командой, хранящейся по адресу K и занимающей L ячеек, будет выполняться команда, с адресом $K+L$. Такой порядок выработки команд называется естественным, и он нарушается только специальными командами. Тогда учитывать адрес следующей команды в явном виде нет необходимости и можно использовать трехадресную команду (рис. 4.14).

Операция, описываемая трехадресной командой, символически представляется в виде:

ОП[A3] := ОП[A1] *ОП[A2].

Можно условиться, что результат операции всегда помещается на место одного из операндов. Тогда получаем двухадресную команду, задающую операцию (рис. 4.14):

ОП[A1] := ОП[A1] *ОП[A2].

В одноадресной команде подразумеваемые адреса имеют уже и результат операции, и один из операндов (рис. 4.15.).

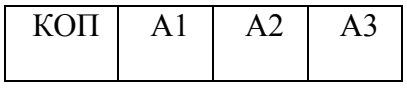


Рис. 4.13.

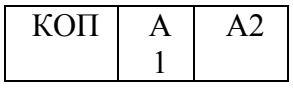


Рис. 4.14.

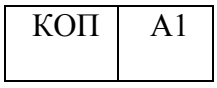


Рис. 4.15.

Подразумеваемым операндом является в данном случае внутренний регистр процессора, называемый регистром результата или аккумулятором. Результат такой операции также записывается в аккумулятор:

Акк := Акк * ОП[A1].

Наконец, в некоторых случаях возможно использование безадресных команд, когда подразумеваются адреса обоих операндов и результата. Например, при использовании стековой памяти.

С точки зрения программиста наиболее естественны и удобны трехадресные команды. Однако, такие команды длины. Кроме того, часто в качестве операндов используются результаты предыдущих команд. В этом

случае выполняемая операция принимает уже вид двух– или одноадресной команды и трехадресный формат использовать неэффективно.

Обычно в ЭВМ используется несколько форматов различной длины. Рассмотренные структуры команд достаточно схематичны. В действительности адресные поля команд большей частью содержат не сами адреса, а только информацию об этих адресах. Способы адресации рассматриваются в следующем разделе.

Критерии разработки для форматов команд

При разработке формата команд необходимо учитывать ряд факторов. Эффективность конкретной архитектуры команд зависит от технологии, которая применялась при разработке компьютера. Например, если доступ к памяти осуществляется быстро, то можно использовать стековую архитектуру, а если доступ к памяти медленный, то желательно иметь много регистров.

Если речь идет об одинаковых машинах, то лучше иметь короткие команды, чем длинные. Такие программы занимают в памяти меньше места. Но минимизация размера команд может усложнить их декодирование. Следовательно, достижение минимальной длины команды должно уравновешиваться временем, затрачиваемым на декодирование и выполнение команд.

Следующая причина минимизации длины команд связана с увеличением скорости работы процессора. Значительный рост скорости работы процессора не соответствует относительно низкой пропускной способности памяти. Трудности, связанные с пропускной способностью памяти имеют отношение не только к основной памяти, но и ко всем видам кэш-памяти.

Если пропускная способность кэш-памяти составляет t бит/с, а средняя длина команды r битов, то кэш-память способна передать самое большое (верхний предел) t/r команд/с. Очевидно, что скорость, с которой могут выполняться команды (скорость работы процессора), может ограничиваться длиной команд, чем короче команда, тем быстрее. А поскольку современные компьютеры выполняют несколько команд за цикл, то вызов нескольких команд за цикл становится обязательным.

Второй аспект разработки – достаточно большой объем пространства в формате команд для выражения всех требуемых команд.

$$n_{КОП} \geq \log_2 M \quad (M \text{ – число команд}).$$

История доказывает, что обязательно нужно оставлять большее количество свободных кодов операций для будущих дополнений к наборам команд.

Третий критерий связан с числом битов в адресном поле. Для адресации S ячеек памяти адресная часть одного операнда должна иметь число разрядов

$$n_A \geq \log_2 S.$$

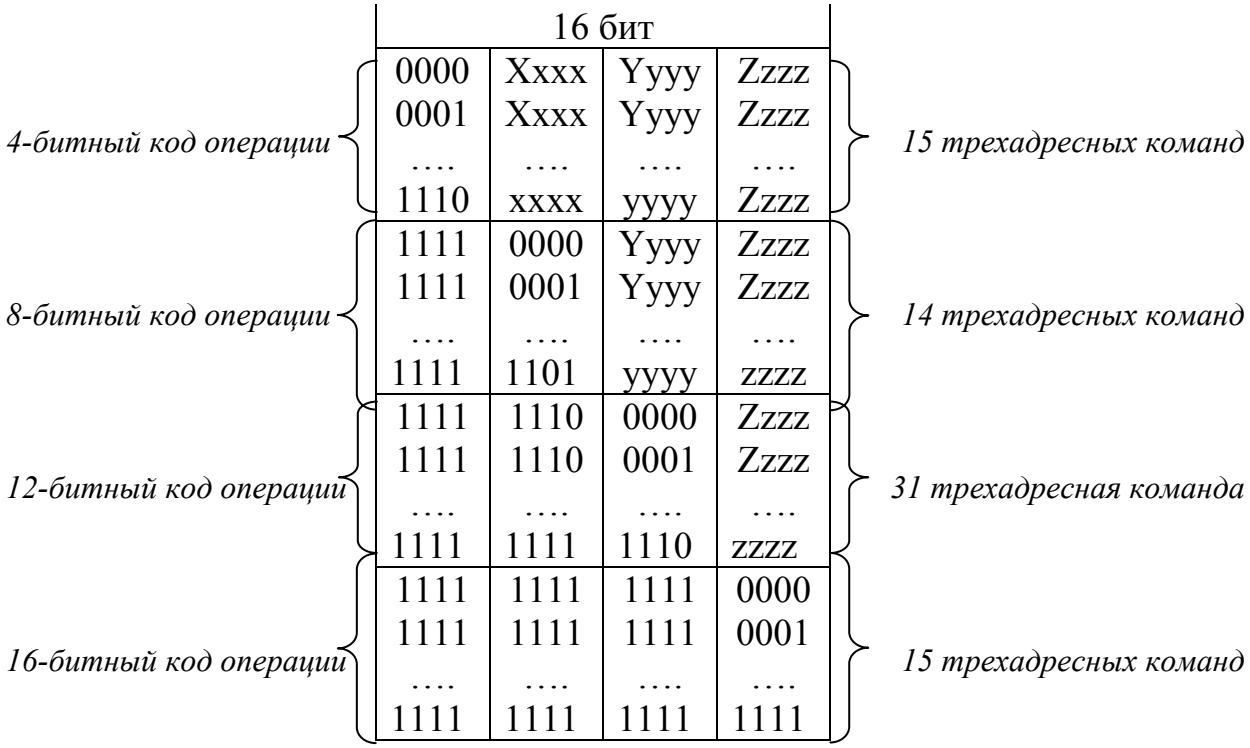
Для адресации памяти требуется более длинные адреса. Одна крайность – это организация памяти, при которой адресуется каждый бит (Burroughs B1700). Другая крайность – это память, состоящая из очень длинных слов (например, серия CDC Cyber содержала 60-битные слова).

Кроме того, для упрощения аппаратуры и увеличения производительности ЭВМ желательно, чтобы длина формата команды была согласована с длиной машинных слов.

Современные компьютерные системы пришли к компромиссу, который в некотором смысле объединяет в себе худшие качества обоих вариантов. Он требует, чтобы адреса были у каждого байта, но при обращении к памяти всегда считывается одно, два, а иногда и четыре слова сразу. В результате считывания одного байта в машине UltraSPARC II сразу вызывается минимум 16 байтов, а иногда и строка кэш-памяти в 64 байта.

Расширение кода операций

Наряду с обычными компромиссами между числом разрядов, отводимых под код операции и адресную часть при фиксированной длине команд, рассмотрим прием, который называется *расширением кода операций*. Суть приема заключается в том, чтобы увеличивать разрядность кода за счет изменения количества адресов в команде. Рассмотрим 16-битную команду:



Другой вариант расширения количества команд – использование команд различной длины. Предлагается делать часто используемые команды короткими, а редко используемые – длинными. Однако в этом случае возникает проблема с декодированием и выравниванием (например, Intel).

Форматы команд процессора Pentium II

Общий формат команды процессора приведен на рис. 4.16.

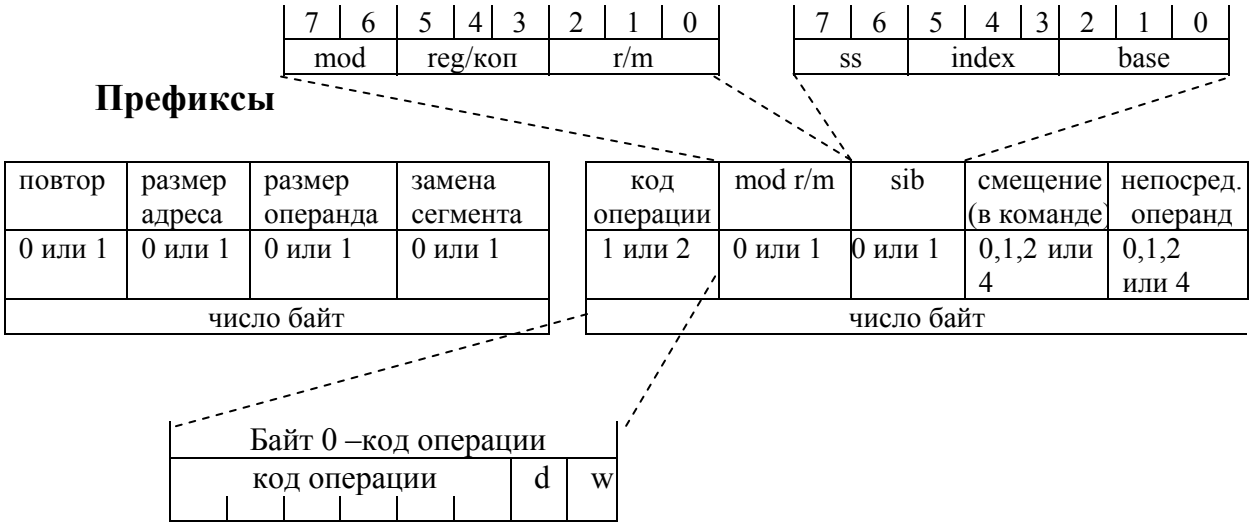


Рис. 4.16

Команда содержит следующие поля:

- префиксы команды (необязательны),которые могут следовать в произвольном порядке;
- одного или двух байт (главного) кода операции;
- спецификатора адреса, представленного битами *mod r/m* и *sib*;
- смещения в команде (*displacement*) – если необходимо;
- поле непосредственного значения (если такое значение присутствует).

(Base – база, MODe – режим, Register/Memory – регистр/память, Scale – масштаб, Index – индекс). Из всех полей команды обязательными являются только один или два байта кода операции.

Префикс – это байт со специальным кодированием, который модифицирует операцию одной находящейся за ним команды. Префиксы сгруппированы следующим образом:

- префиксы команды (*REP\REPE\REPZ\REPNE*, применяется в строковых командах, для автоматической обработки всех элементов цепочки и блокировки шины (LOCK) – допускается перед некоторыми командами с обращением к памяти для запрета другим процессорам в мультипроцессорных системах);
- замены сегмента (*Segment override*), явно определяет сегментный регистр для конкретной команды вместо сегментного регистра, принимаемого по умолчанию;
- размера операнда (*OperandSize*), переключает 16– и 32–битные операнды;

- размера адреса (*AddressSize*), коммутирует формирование 16– и 32–битных адресов;

По сравнению с 16-ти разрядными процессорами x86, начиная с I80386 система команд расширена в двух направлениях: во-первых, все команды, рассчитанные на 16-разрядные операнды работают и с 32–разрядными; во-вторых, доступны 32-разрядные режимы адресации. Такое расширение реализовано введением бита D (*Default*) в дескриптор сегмента кода и двух префиксов. Когда бит D установлен в 0, команды текущей программы работают с 16-разрядными данными и адресами, если D установлен в 1, то они работают с 32-разрядными данными и адресами.

Префиксы размера и адреса операнда заставляют процессор изменять для команды размеры операнда или адреса.

В большинстве команд вслед за байтом кода операции, который указывает местонахождение операнда в памяти, следует второй байт, который сообщает всю информацию об операнде. Эти 8 бит распределены в 2-битном поле mod и в двух 3-битных регистровых полях reg и r/m.

Рассмотрим поле *reg*. Это поле определяет один из регистров общего назначения. Кодирование этого поля зависит от того, имеется ли в КОП поле W (определяющее размер операнда), а при его наличии от типа операнда. Это поле может находится в байте КОП и в байте mod r\m.

Режим адресации определяют один или два байта адресации, которые находятся после байта адресации. Первым является байт *mod r/m*, вторым байт *sib* (масштаб, индекс, база). Байт *sib* может присутствовать только в командах с 32-разрядной адресацией, когда байт *mod r/m* содержит *r/m* = 100 и значение в поле *mod*≠11.

Таблица 4.1

reg	поля W нет		Поле W есть			
			16 разрядов		32 разряда	
	16 р	32 р	W=0	W=1	W=0	W=1
000	AX	EAX	AL	AX	AL	EAX
001	CX	ECX	CL	CX	CL	ECX
010	DX	EDX	DL	DX	DL	EDX
011	BX	EBX	BL	BX	BL	EBX
100	SP	ESP	AH	SP	AH	ESP
101	BP	EBP	CH	BP	CH	EBP
110	SI	ESI	DH	SI	DH	ESI
111	DI	EDI	BH	DI	BH	EDI

Форматы команд FPU

Особенности задания команд. Команды бинарных операций допускают несколько форм. В случае пустого поля операнда (безадресная команда) операция выполняется с двумя верхними элементами стека ST(0) и ST(1).

После выполнения операции осуществляется инкремент указателя стека и результат помещается в новую вершину стека, заменяя исходное содержание ST(1). Это классическая операция для машин со стековой адресацией.

Когда в бинарной операции определен один операнд, она выполняется с привлечением указанного в команде регистра (ячейки памяти) и содержимого вершины стека. Результат загружается в старую вершину стека и указатель стека не изменяется.

Если же в бинарной команде указаны два операнда, ими является содержимое двух регистров стека, причем одним из них будет ST(0), а вторым ST(i). Возможны три случая:

- источникам является ST(0), а получателем ST(i);
- источник — ST(i), получатель — ST(0);
- источникам является ST(0), получателем ST(i) и производится инкремент указателя стека, т.е. “источник” пропадает.

Отметим, что в несимметричных операциях деления и вычитания обычно получатель делится на источник (из получателя вычитается источник). FPU допускает и обратные (Reverse) формы таких команд.

Многие команды допускают несколько форм задания операндов, которые при этом принято записывать через слэш «/», например

FADD //src/dst,src .

Эта запись подразумевает три возможные формы команды: без операндов, с одним операндом, с двумя операндами.

src – Source — источник;

dst – Destionation – получатель.

В мнемониках команд FPU приняты следующие соглашения:

- первая буква команды всегда F (только команды FPU начинаются с F);
- вторая буква I обозначает операцию с целым двоичным числом из памяти, буква B – операцию с десятичным числом из памяти, без этих букв – вещественное число;
- предпоследняя или последняя буква R указывает обратную операцию (для вычитания или деления);
- последняя буква P идентифицирует команду, заключительным действием которой является извлечение из стека.

Рассмотрим машинный формат команд.

Для команд FPU существует пять различных форматов кодирования (рис. 4.17), однако в любом формате старшие пять битов двухбайтного кода команды FPU всегда содержит специальное значение 11011, по которому процессор распознает команды FPU.

Команда																Доп. поле	
Первый байт								Второй байт									
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0		
1	1	0	1	1	OP1		1	Mod		1	OP2		R/m			sib	Disp
1	1	0	1	1	MF		OP1	Mod		OP2			R/m			sib	Disp

1	1	0	1	1	d	p	OP1	1	1	OP2		R	ST(i)		
1	1	0	1	1	0	0	1		1	1	1	OP			
1	1	0	1	1	0	1	1		1	1	1	OP			

Рис. 4.17

Поля имеют следующее назначение и кодирование:

- MF – формат ячейки памяти. (00 – 32-битное вещественное, 01- 32-битное целое, 10 – 64-битное вещественное, 11 – 64-битное целое);
- P– извлечение из стека (0 – не извлекать; 1 – извлекать);
- D- назначение (0- ST(0), 1 – ST(1));
- R– направление операции;
- ST(i) – элемент регистрового стека;
- OP1, OP2 – первая и вторая части кода операции.
- Поля mod, r/m, sib кодируются также, как для базовых команд процессора.

В АССЕМБЛЕРЕ резервирование памяти осуществляется с помощью директив: DD – двойное слово; DQ – учетверенное слово; DT – определить 10 байт.

Форматы команд процессора UltraSPARC II

Команды 32-битные, выровненные в памяти. Каждая команда определяет только одно действие. Типичная команда указывает два регистра, из которых берутся операнды и выходной регистр. Вместо одного регистра допускается использование константы со знаком. При выполнении команды считывания два регистра (или один регистр и одна константа) складываются вместе и определяют адрес памяти, по которому производится считывание; результат записывается в указанный регистр.

Изначально SPARC имела ограниченное число форматов (рис. 4.18). Со временем добавлялись дополнительные форматы и сейчас их 31.

Первые два бита команды помогают определить формат команды и сообщают программному обеспечению, где найти оставшуюся часть кода операции, если она есть.

	2	5	6	5	1	8	5	
1a		Вых. Рг.	КОП	Вх. Рг1	0 x	Операция с п/з	Вх. Рг.2	*3 регистра*
1b		Вых. Рг.	КОП	Вх. Рг1	1	Непосредственная константа		**Непосредственный операнд**

	2	5	3	22	
2		Вых. Рг.	КОП	Непосредственная константа	**SETHI**

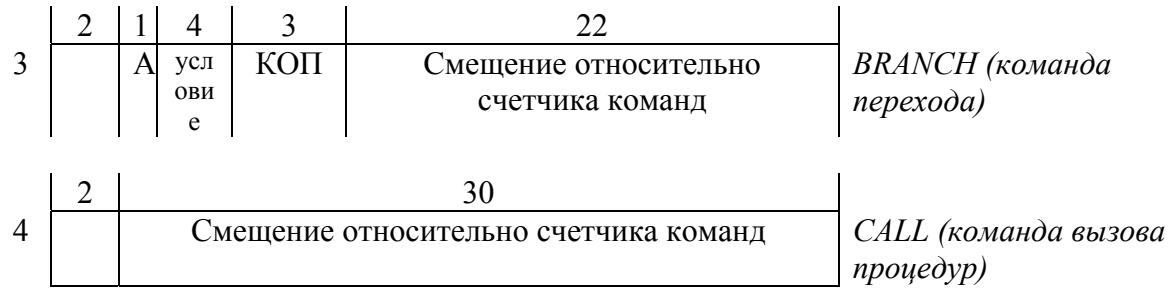


Рис. 4.18

В формате 1a оба источника расположены в регистрах. В формате 1b один операнд – константа. Бит 13 (0 или 1) определяет один из этих форматов.

Поскольку все команды 32-битные, то включить в команду 32-битную константу невозможно. Команда SETHI устанавливает 22 бита, оставляя пространство для другой команды, чтобы установить оставшиеся 10 битов. Это единственная команда, использующая данный формат.

Для непрогнозируемых условных переходов используется формат 3, в котором поле условие определяет, какое условие надо проверить. Бит A нужен для того, чтобы избегать пустых операций при определенных условиях. Прогнозируемые переходы используют тот же формат, но только с 19-битным смещением.

Последний формат используется для вызова процедур. Требуемый адрес – это целевой адрес, разделенный на четыре.

Форматы команд JVM

Большинство форматов команд просты. Это определяется тем, что машина JVM достаточно проста. Все команды начинаются с кода операции в 1 байт. В некоторых командах за кодом операции следует второй байт, который может быть индексом, константой или указателем типа данных (рис. 4.19).

Все команды JVM за исключением восьми особых команд используют простые форматы 1, 2 и 3. Команды JVM кодируются таким образом, чтобы большинство наиболее часто используемых команд кодировались 1 байтом.

4

КОП	Индекс	константа

5

КОП	Индекс	Размерность массива

6

КОП	Индекс	параметры	0

7

КОП	Индекс		константа

8

КОП	32-битное смещение перехода

9

КОП	Длина переменной

Рис. 4.19

4.4. АДРЕСАЦИЯ

Как было сказано ранее, для адресации S ячеек памяти адресная часть одного операнда должна иметь число разрядов

$$n_A \geq \log_2 S.$$

Это требование находится в противоречии с желанием иметь малую разрядность команды и возможностью использовать большое адресное пространство. Существуют два основных подхода к решению этой проблемы.

1. Использовать регистры общего назначения. Это существенно повышает быстродействие, но имеет эффект толь в том случае, если операнд используется многократно. Кроме того, существует ограничение на количество регистров общего назначения.
2. Второй подход подразумевает определение одного или нескольких операндов неявным образом. Это означает использование не трехадресной команды, а двух- одно- и безадресных команд.

Следует различать понятия адресный код в команде **Ак** и исполнительный адрес **Аи.** Адресный код –это информация об адресе команды, содержащаяся в команде. Исполнительный адрес – это номер ячейки памяти, к которой производится обращение. В современных ЭВМ адресный код, как правило, не совпадает с исполнительным адресом. В свою очередь исполнительный адрес может не совпадать с физическим из-за таких особенностей ЭВМ как сегментация и виртуальная память.

Рассмотрим способы адресации, используемые в современных ЭВМ.

Способы адресации

Подразумеваемый операнд. В команде не содержится в явном виде указаний об адресе операнда. Операнд подразумевается и фактически

задается кодом операции команды. Вообще, данный метод используется нечасто, однако в архитектуре x86 имеется достаточное количество таких команд: INC, DEC.

Подразумеваемый адрес. В команде не содержится явных указаний об адресе участвующих в операции операндов, или адреса, по которому помещается результат операции, но этот адрес подразумевается. Например, команда может содержать адреса обоих операндов, участвующих в операции, а результат помещается по адресу одного из операндов. Сюда также можно отнести операции с использованием подразумеваемого регистра.

Непосредственная адресация. В команде содержится не адрес операнда, а непосредственно операнд. Непосредственная адресация удобна при использовании различного рода констант.

Прямая адресация. Исполнительный адрес совпадает с адресной частью кода команды. Этот способ адресации был общепринятым в первых вычислительных машинах и продолжает применяться в настоящее время в комбинации с другими способами.

Регистровая адресация. В качестве операнда используется содержимое регистров процессора (ячейки сверхоперативной памяти). Например, если таких регистров 16, то для адреса регистра достаточно 4-х битов. Регистровая адресация наряду с сокращением длины адресов операндов позволяет увеличивать скорость выполнения операций.

Косвенная адресация. Адресный код команды указывает адрес ячейки памяти, в которой находится адрес операнда или команды. Т.е. происходит «адресация адреса». Если в качестве адресная часть команды – регистр, то такая адресация называется *регистровой косвенной адресацией*, а используемых регистр – *указателем*.

Косвенная адресация начала широко использоваться в свое время в малых ЭВМ и микропроцессорах, имеющих короткое машинное слово, для преодоления короткого формата команды.

Особенно часто применяется совместное использование регистровой и косвенной адресаций.

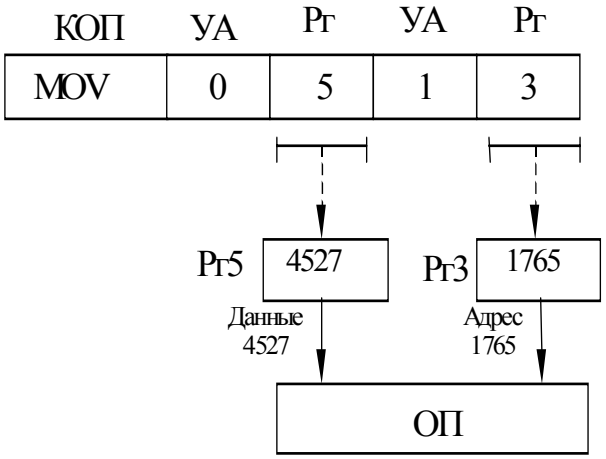


Рис. 4.20

На рис. 4.20 описаны следующие поля:

- КОП – код операции;
- Рг – номер регистра;
- УА – указатель косвенной адресации.

На рис. 4.20 показан механизм пересылки числа 4527 из регистра 5 в ячейку памяти 1765. Операнд 4527 указывается в регистровой прямой адресации, а для задания адреса 1765 используется регистровая косвенная адресация.

Относительная адресация или *базирование*. Исполнительный адрес определяется суммой адресного кода команды Ак и некоторого числа Аб, называемого базовым адресом:

$$Аи = Аб + Ак.$$

Для хранения базовых адресов в машине могут быть использованы специальные регистры или ячейки памяти (базовые регистры). В команде выделяется поле В для указания номера базового регистра.

Относительная адресация позволяет при меньшей длине адресного кода команды обеспечить доступ к любой ячейке памяти. Для этого число разрядов в базовом регистре выбирается таким, чтобы можно было адресовать любую ячейку ОП, а адресный код Ак самой команды используют для представления сравнительно короткого «смещения» (обозначим D). Смещение D определяет положение операнда относительно начала массива, задаваемого базовым адресом Аб. Схема формирования исполнительного адреса приведена на рис. 4.21.

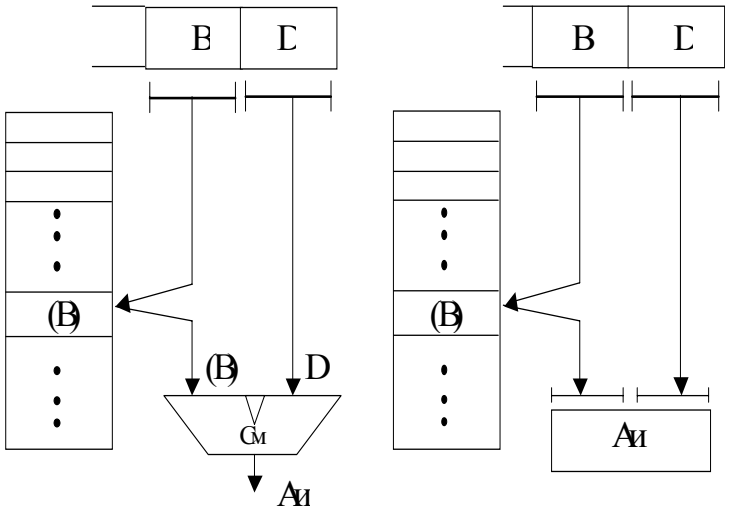
Большей частью исполнительный адрес при базировании образуется с помощью сумматора согласно выражению:

$$Аи = (B) + D,$$

где (В) – содержимое регистра с номером В.

Иногда применяют формирование исполнительного адреса методом совмещения. В этом случае базовый адрес содержит старшие, а смещение –

младшие разряды исполнительного адреса, которые объединяются в регистре адреса путем их составления.



а) б)

Рис. 4.21

Однако при совмещении базовый адрес может задавать не любые ячейки памяти, а только те, адреса которых содержат нули в младших разрядах, соответствующих смещению.

Относительная адресация обеспечивает так называемую *перемещаемость* программ, т.е. возможность перемещения программ в памяти без изменений внутри самой программы.

Индексная адресация. Характерной особенностью вычислительных процессов, происходящих в ЭВМ, является цикличность, при которой повторяются одни и те же процедуры, но над различными операндами (как правило, элементами массивов, расположенными упорядочено в памяти). Поскольку операнды, обрабатываемые при построении цикла, имеют разные адреса, можно для каждого повторения составить свою последовательность команд, отличающуюся только адресными частями.

Такая программа, состоящая из групп команд, отличающихся только адресной частью, является, очевидно, слишком длинной, а написание ее – слишком трудоемким.

Программирование вычислительных циклов существенно упроститься, если после каждого выполнения цикла обеспечить автоматическое изменение в соответствующих командах их адресных частей согласно расположению в ОП обрабатываемых операндов. Такой процесс называется *модификацией команд*, точнее адресных частей команд. Модификация команд основана на возможности выполнения над кодами команд или их частями арифметических и логических операций. Идея модификации команд была предложена фон Нейманом.

В первых вычислительных машинах действительно использовалась модификация команд. В современных ЭВМ используется механизм *индексации.* Это понятие включает в себя специальный способ кодирования команд, командные и аппаратные средства задания и выполнения модификации команд и управления вычислительными циклами. Упомянутые средства называются *индексной арифметикой*. По своей сути индексация является дальнейшим развитием базирования.

Для выполнения индексации в архитектуру процессора вводятся так называемые *индексные регистры*. В формате команды вводится поле X для

указания индексного регистра. Исполнительный адрес при индексации формируется путем сложения адресного кода команды (смещения) с содержимым индексного регистра (индексом). Во многих архитектурах содержимое индексного регистра дополнительно умножается на размер операнда. При наличии базирования добавляется еще и базовый адрес.

Стековая адресация. Стековая память реализует безадресное задание операндов. *Стек* представляет собой группу последовательно пронумерованных регистров *(аппаратный стек)* или ячеек памяти, снабженных указателем стека (обычно регистром SP), в котором автоматически при записи и считывании указывается номер (адрес) последней занятой ячейки стека *(вершины стека)*. При выполнении операции записи в стек слово помещается в следующую ячейку стека, а при считывании из стека последнее поступившее в него слово. Таким образом, в стеке реализуется дисциплина обслуживания «последний пришел – первый ушел» (LIFO).

Указанное правило при обращении к стеку реализуется автоматически, и поэтому при операциях со стеком возможно безадресное задание операндов.

Механизм стековой адресации может быть проиллюстрирован рисунком (рис. 4.22).

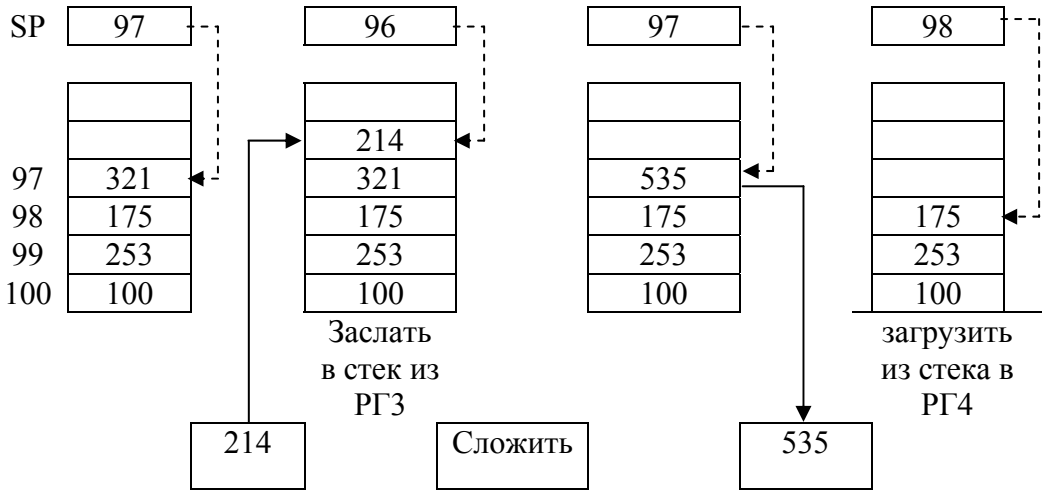


Рис. 4.22

При выполнении команды передачи в стек слова указатель стека увеличивается на 1, а затем слово помещается в ячейку стека, указываемую SP. При загрузке из стека сначала извлекается слово, а затем указатель уменьшается на 1.

При соответствующем расположении операндов в стеку можно вычислять выражения полностью безадресными командами, указывающими только тип операции. Такая команда извлекает из стека в соответствии с кодом операции один или два операнда, выполняет предписанную операцию и заносит результат в стек.

Вычисления с использованием стековой памяти удобно описывать и программировать с помощью Польской инверсной (бесскобочной) записи арифметических выражений. Эта запись производится по следующему правилу: читаем арифметическое выражение слева на право и последовательно, друг за другом выписываем встречающиеся операнды. Как только окажется, что все операнды некоторой операции выписаны, записываем знак этой операции и продолжаем выписывать операнды. Если операция имеет операндом результат некоторой предыдущей операции и знак последней выписан, то считаем этот операнд выписанным.

Например, выражение

$$(k+l-m)(p-s)$$

в бесскобочной записи имеет вид:

$$kl+m-ps-x.$$

Выражение в польской инверсной записи не содержит скобок, но порядок действий определен однозначно. Программа вычисления такого выражения может иметь следующий, достаточно условный вид:

```
push   k
push   l
add
push   m
sub
push   p
push   s
sub
mul
```

Безадресные команды на основе стековой адресации предельно сокращают форму команд, экономят память и способствуют повышению производительность ЭВМ.

Однако при такой структуре команд возникают сложности с построением команд передачи управления и работы с периферийными устройствами.

В современной архитектуре процессоров стек и стековая адресация применяется при организации переходов к подпрограммам и системах прерывания. Элементы стековой адресации используются в процессорах x86 при работе с данными с плавающей точкой.

Способы адресации команд перехода

Командам перехода (а также командам вызова процедур) также нужны особые способы адресации для определения целевого адреса. Способы,

существующие для адресации данных могут быть также использованы для вычисления исполнительных адресов команд перехода. Один из возможных вариантов – прямая адресация, когда целевой адрес просто полностью включается в команду.

Другие способы адресации также имеют смысл. Косвенная и регистровая адресация позволяют программа вычислить целевой адрес, помещать его в регистр, а затем переходить по этому адресу. Такой способ дает максимальную гибкость, поскольку целевой адрес вычисляется во время выполнения программы. Но такой подход предоставляет огромные возможности для появления ошибок, которые практически невозможно найти.

Индексная адресация, при которой известно смещение от регистра, также может быть использована.

Может быть также использована относительная адресация по счетчику команд. В этом случае для получения целевого адреса смещение (со знаком), находящееся в самой команде прибавляется к программному счетчику.

Способы адресации процессора Pentium II

Способы адресации процессора Pentium II нерегулярны и зависят от того, в каком формате находятся команды: 16- или 32-битном.

Режим адресации определяют один или два байта адресации, которые находятся после байта адресации. Первым является байт *mod r/m*, вторым байт *sib* (масштаб, индекс, база). Байт *sib* может присутствовать только в командах с 32-разрядной адресацией, когда байт *mod r/m* содержит *r/m* = 100 и значение в поле *mod*≠11.

Рассмотрим 16-разрядную адресацию, а именно, кодирование двухоперандной команды в 16-разрядном режиме (рис. 4.23).

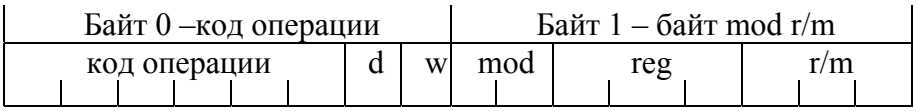


Рис. 4.23

Адрес формируется следующим образом:

[РгБазы]+[смещение в команде]+индекс=
(это эффективный адрес)+[Базовый адрес смещения]=
линейный адрес, он же физический.

Поле w определяет размер операнда; если w = 0, то операнд равен байту, при w = 1 – полный размер.

Поле d определяет направление передачи данных: из регистра в регистр/память (d=0) или из регистра/паять в регистр (d = 1).

Поле reg определяет регистровый операнд так, как было уже рассмотрено.

Двухбитное поле mod показывает, находится ли второй операнд в регистре или в памяти. Если он находится в регистре (mod = 11), то 3–х разрядное поле r/m определяет второй регистровый операнд. В противном случае (mod ≠11) операнд находится в памяти и поля mod и r/m задают режим адресации.

Таблица 4.2

r/m	Mod		
	00	01	10
000	[BX+SI]	[BX+SI]+d8	[BX+SI]+d16
001	[BX+DI]	[BX+DI] +d8	[BX+DI] +d16
010	SS:[BP+SI]	SS:[BP+SI] +d8	SS:[BP+SI] +d16
011	SS:[BP+DI]	SS:[BP+DI] +d8	SS:[BP+DI] +d16
100	[SI]	[SI] +d8	[SI] +d16
101	[DI]	[DI] +d8	[DI] +d16
110	d16	SS:[BP+d8]	SS:[BP+d16]
111	[BX]	[BX] +d8	[BX] +d16

Рассмотрим 32-разрядную адресацию. Для адресации теперь можно использовать любой регистр общего назначения, а индекс разрешается масштабировать (умножать) на 1, 2, 4 или 8. Ниже приводятся все способы 32–разрядной адресации.

Таблица 4.3

Непосредственная	mov eax, 12345678h
Регистровая	mov eax, ecx
Прямая (абсолютная)	mov eax, [3456789h]
Регистровая косвенная	mov eax, [ecx]
Базовая (индексная) со смещением	mov eax, [ecx]+1200h
Базовая индексная со смещением	mov eax, [ecx][edx]+40h
Индексная с масштабированием и смещением	mov eax, [eci*4]+400h
Базовая индексация с масштабированием	mov eax, [edx][ecx*8]
Базовая индексация с масштабированием и смещением	mov eax, [ebx][edi*2]+20h

Проблема заключается в том, что не все способы адресации применимы ко всем командам и не все регистры могут использоваться при всех способах адресации. Это существенно усложняет работу компилятора.

Схема формирования адреса в 32-разрядной адресации приведена на рис 4.24

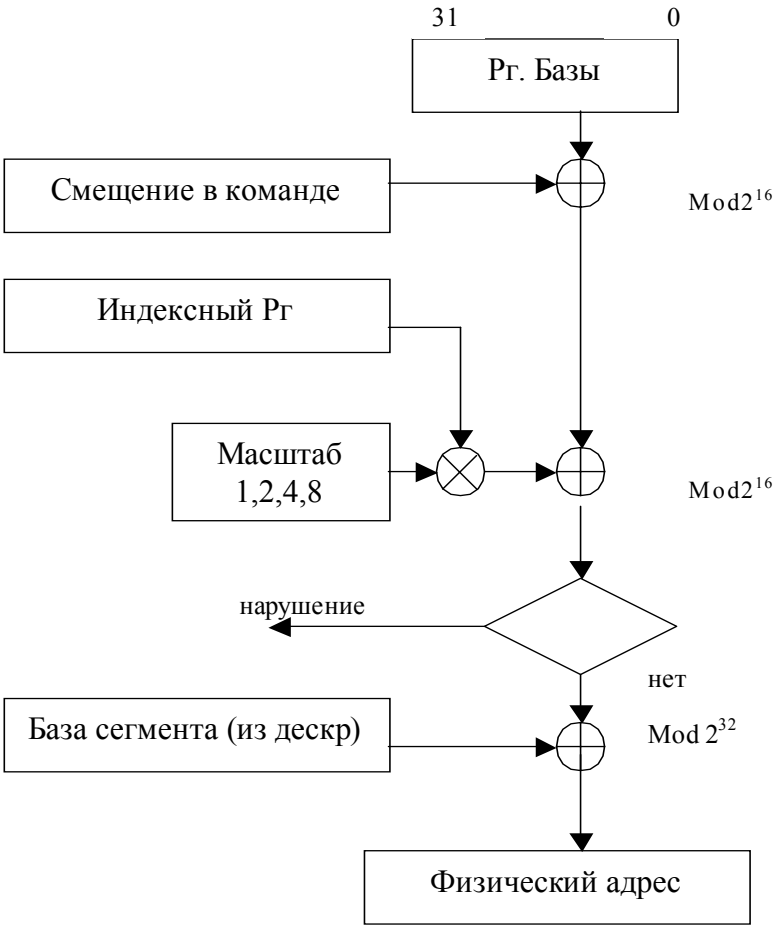


Рис. 4.24.

32–разрядные режимы адресации кодируются с помощью двух байт mod r/m и sib.

Байт *mod* управляет способами адресации. Один из операндов определяется по комбинации полей mod и r/m. Второй операнд всегда является регистром и определяется по значению поля reg. (таблицы 4.4. и 4.5).

Формирование адреса памяти в 32-битной адресации (r/m ≠100).

Таблица 4.4

r/m	Адрес памяти второго операнда			
	mod = 00	mod = 01	mod = 10	Mod = 11
000	EAX	EAX+d8	EAX+d32	EAX или AL
001	ECX	ECX+d8	ECX+d32	ECX или CL
010	EDX	EDX+d8	EDX+d32	EDX или DL
011	EBX	EBX+d8	EBX+d32	EBX или BL
100	имеется sib	имеется sib	имеется sib	ESP или AH
101	d32	SS:[EBP+d8]	SS:[EBP+d32]	EBP или CH
110	ESI	ESI+d8	ESI+d32	ESI или DH
111	EDI	EDI+d8	EDI+d32	EDI или BH

Колонки 01 и 10 включают способы адресации, при которых значение регистра прибавляется к 8-битному или 32-битному смещению, которое следует за командой.

Иногда вслед за байтом mod следует дополнительный байт sib. Байт sib определяет масштабный коэффициент и два регистра.

Поле SS указывает масштабный коэффициент индекса, поле index определяет любой регистр, кроме ESP, который служит индексным регистром, а поле base – определяет базовый регистр. Формирование адреса памяти в 32-битной адресации (r/m =100, имеется sib) приведено в табл. 4.6.

Таблица 4.6

поле base	Адрес памяти второго операнда		
	mod = 00	mod = 01	mod = 10
000	EAX+ss*ind	EAX+ss*ind+d8	EAX+ss*ind+d32
001	ECX+ss*ind	ECX+ss*ind+d8	ECX+ss*ind+d32
010	EDX+ss*ind	EDX+ss*ind+d8	EDX+ss*ind+d32
011	EBX+ss*ind	EBX+ss*ind+d8	EBX+ss*ind+d32
100	SS:[ESP+ss*ind]	SS:[ESP+ss*ind+d8]	SS:[ESP+ss*ind+d32]
101	d32+ss*ind	d32+ss*ind+d8	d32+ss*ind+d32
110	ESI+ss*ind	ESI+ss*ind+d8	ESI+ss*ind+d32
111	EDI+ss*ind	EDI+ss*ind+d8	EDI+ss*ind+d32

Таблица кодирования полей index и ss.

index	индексный регистр	ss	Множитель
000	EAX	00	*1
001	ECX	01	*2
010	EDX	10	*4
011	EBX	11	*8
100	нет		
101	EBP		
110	ESI		
111	EDI		

Способы адресации процессора UltraSPARC II

В архитектуре команд процессора UltraSPARC II все команды используют непосредственную и регистровую адресацию за исключением тех команд, которые непосредственно обращаются к памяти. При регистровом способе адресации 5 битов просто сообщают, какой регистр нужно использовать. При непосредственной адресации данные обеспечивает

13-битная константа со знаком. Для арифметических, логических и подобных команд никакие другие способы адресации не используются.

К памяти обращаются команды трех типов: считывания (Load), записи (Store) и одна команда синхронизации мультипроцессора. Для команд записи и считывания существует два способа обращения к памяти.

Первый. Вычисляется сумма двух регистров, а затем через полученное значение производится косвенная адресация.

Второй способ представляет собой обычное индексирование с 13-битным смещением со знаком.

Способы адресации машины JVM

У машины JVM нет общих способов адресации в том смысле, что каждая команда содержит несколько битов, которые сообщают, каким образом надо вычислить адрес. Вместо этого здесь с каждой командой связан один особый способ адресации. Поскольку в JVM нет видимых регистров, регистровая и косвенная адресация здесь невозможна. Несколько команд используют непосредственную адресацию. Единственный доступный способ адресации – индексная. Она используется командами, которые определяют переменную, связанную с каким-нибудь неявным регистром.

Сравнение способов адресации

Способ адресации	Pentium II	UltraSPARC II	JVM
Непосредственная	*	*	*
Прямая	*		
Регистровая	*	*	
Косвенная регистровая	*		
Индексная	*	*	*
Относительная индексная		*	
Стековая			*

На практике для эффективной архитектуры команд совсем не обязательно использовать большое число способов адресации. Поскольку практически весь код, написанный на этом уровне порождается компилятором, то способов адресации должно быть мало, они д.б. четкими и ясными.

Поэтому самые простые архитектуры используют небольшое число способов адресации, а на каждый используемый способ накладываются жесткие ограничения. Обычно вполне достаточно непосредственной, прямой, регистровой и косвенной адресации.

При изучении нового компьютера нужно изучать все команды и способы адресации не только для того, что бы знать, какие из них имеются в

наличии, но и для того, чтобы понять, почему был сделан такой выбор и как это можно использовать.

4.5. ТИПЫ КОМАНД

Команды можно грубо поделить на несколько групп, которые могут повторяться от машины к машине, хотя и различаются в деталях. Кроме того, в каждом компьютере имеется несколько необычных команд, которые появились или из соображений совместимости, или по недоразумению или по какой-нибудь другой причине.

Команды перемещения данных

Как правило, в машинах с фиксированной длинной слова единицей перемещаемых данных является слово. Однако, существуют архитектуры команд дают возможность копировать отрезки данных меньше слова, а также группу слов. Некоторые машины с изменяемой длиной слова содержат команды, которые определяют только адреса источника и получателя, а не количество данных. Копирование продолжается до тех пор, пока не появится специальное поле в конце данных.

Бинарные операции

Бинарные операции берут два операнда и получают результат. Все архитектуры команд содержат операции с фиксированной точкой. Большинство компьютеров сегодня поддерживают операции с плавающей точкой. Большинство машин содержит по крайней мере 2 варианта таких чисел: более короткие для скорости и более длинные для получения высокой точности вычислений.

Унарные операции

Унарные операции используют один операнд и производят один результат. Команды таких операций могут быть короче.

Сравнения и условные переходы

Команда вызова процедур

Особенность выполнения команды вызова процедур заключается в том, что программа должна вернуться к соответствующему оператору (как правило, следующему за процедурой). Следовательно, адрес возврата должен либо передаваться процедуре, либо сохраняться таким образом, чтобы была возможность определить его местонахождение по окончанию процедуры.

Адрес возврата может быть размещен в одном из трех мест: в памяти, в регистре или в стеке.

Самое худшее решение – размещение в фиксированной ячейке памяти. В таком случае, если процедура вызывает другую процедуру, то адрес теряется.

Более удачное решение – сохранять адрес возврата в первом слове процедуры. Недостаток такой схемы – процедура не может вызвать сама себя, т.к. первый адрес возврата будет уничтожен вторым вызовом.

Еще более удачное решение – помещать адрес возврата в регистр. Но если процедура рекурсивна, то каждый раз необходимо помещать адрес в новое место, а количество регистров ограничено.

Самое лучшее решение – поместить адрес возврата в стек. Когда процедура завершена, она выталкивает адрес возврата.

Сравнение наборов команд

Наборы команд сравниваемых архитектур существенно отличаются друг от друга.

Pentium II – классическая двухадресная 32-битная машина CISC. Эта машина с долгой историей и она содержит много команд, которые обращаются к памяти.

UltraSPARC II – это современная трехадресная 64-битная машина RISC с архитектурой загрузки/сохранения, всего двумя способами адресации и компактным набором команд.

JVM – машина со стековой организацией, практически без способов адресации, с регулярными командами и плотным кодированием команд.

Говорят, что в основу разработки компьютера Pentium II лежали три основных фактора:

1. Обратная совместимость.
2. Обратная совместимость.
3. Обратная совместимость.

Сейчас бы никто не начал разработку машины с такой нерегулярной системой команд, с таким маленьким количеством абсолютно разных регистров. По этой причине очень сложно писать компиляторы. Из-за недостатка регистров компиляторам постоянно приходится сохранять переменные в памяти, затем загружать их, что очень невыгодно даже при наличии трех уровней кэш-памяти. Для обеспечения высокой производительности компьютера было найдено много оригинальных технических решений.

Современная разработка уровня команд представлена в процессоре UltraSPARC II. Он содержит полную 64-битную архитектуру команд. Процессор содержит много регистров и имеет набор команд, в которых преобладают трехрегистровые операции, а также небольшая группа команд записи/считывания. Все команды одного размера, хотя число форматов

вышло из-под контроля. Большинство новых разработок очень похожи на UltraSPARC II, но содержат меньшее число форматов команд.

JVM – машина совершенно другая. Здесь уровень команд изначально разрабатывался таким образом, чтобы небольшие программы можно было передавать по Internet и интерпретировать на программном обеспечении другого компьютера. Эта была разработка для одного языка. Это привело к использованию стека и коротким командам разной длины с очень высокой плотностью (в среднем всего 1,8 байт на команду). Создание аппаратного обеспечения, которое выполняет одну команду JVM за один раз и при выполнении одной команды обращается к памяти два или три раза казалось невозможным. Однако помещение на микросхему стека из 64 слов и преобразованию целых последовательностей команд в трехадресные команды RISC позволило создать достаточно эффективную машину PicoJava.

Ядро современного компьютера представляет собой сильно конвейеризированное трехрегистровое устройство загрузки/сохранения типа RISC. UltraSPARC II декларирует эту структуру. Pentium II скрывает систему RISC, перенимает старую архитектуру команд и разбивает команды CISC на микрооперации RISC. Машина PicoJava также использует ядро архитектуры RISC, но для этого комбинируется несколько команд для получения одной операции RISC.

4.6. ПОТОК УПРАВЛЕНИЯ

Поток управления – последовательность, в которой команды выполняются динамически, т.е. во время работы программы. При отсутствии переходов и вызовов процедур команды вызываются из последовательных ячеек памяти. Вызов процедуры влечет за собой изменение поток управления, выполняемая в данный момент процедура останавливается, и начинается выполнение вызванной процедуры. Сопрограммы связаны с процедурами и вызывают схожие изменения в потоке управления. Сопрограммы используются при решении задач моделирования, для моделирования параллельных процессов. Ловушки и прерывания также меняют поток управления и возникают при определенных ситуациях.

Последовательный поток управления и переходы

В большинстве команды выполняются последовательно и состояние счетчика команд представляет собой возрастающую функцию.

Если программа содержит переход, то соответствие между порядком расположения команд и порядком их выполнения нарушается. В результате последовательность выполнения команд из самой программы уже не видна. Это привело к появлению статьи Дейкстры под названием «Оператор goto следует считать вредным», в которой предлагалось избегать использовать этот оператор. Эта статья дала толчок революции в программировании. Одним из нововведений было устранение операторов goto более

структурированными формами потока управления, например, while. Конечно, эти программы компилируются в программы второго уровня, которые могут содержать команды переходов, поскольку без них не обойтись.

Процедуры

Самым важным способом структурирования программ является процедура. Процедура с одной стороны вызывает изменение порядка следования операций, но с другой стороны процедуру можно интерпретировать как определение новой команды на более высоком уровне.

Особый интерес представляет *рекурсивная процедура*. Это такая процедура, которая вызывает сама себя либо непосредственно, либо через цепочку других процедур.

Сопрограммы

В обычной последовательности вызовов существует четкое различие между вызывающей процедурой и вызываемой процедурой.
Рассмотрим две процедуры А и В. Процедура А вызывает процедуру В. По окончанию процедуры В происходит возврат к процедуре А, и она продолжает выполняться дальше, с команды, следующей за командой вызова процедуры В. При повторном вызове процедуры В она будет выполняться с самого начала, а не с места возврата в процедуру А (рис. 4.25).

Иногда нужно иметь две процедуры А и В, каждая из которых вызывает другую в качестве процедуры так, как это показано на рис. 4.26.

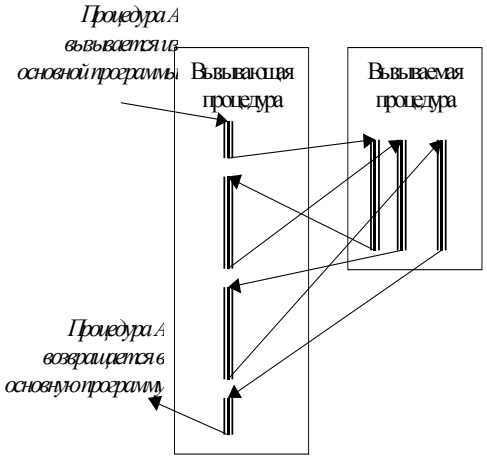


Рис. 4.25

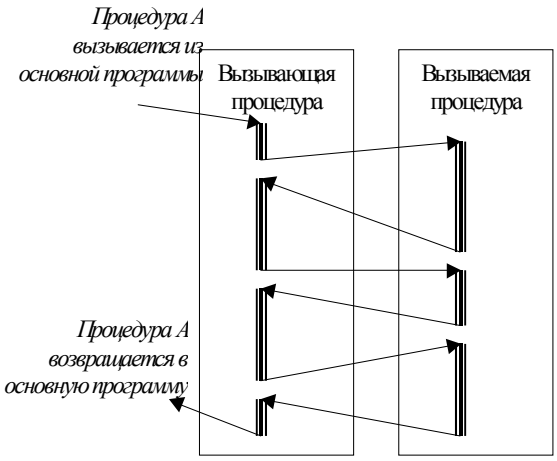


Рис. 4.26

При возврате из В к А процедура В совершает переход к тому оператору, за которым последовал вызов процедуры В. Когда процедура А передает управление процедуре В, она возвращается не к самому началу процедуры в, а к тому месту, на котором произошел предыдущий вызов. Две процедуры, работающие таким образом, называются *сопрограммами.*

Сопрограммы обычно используются для того, чтобы производить параллельную обработку данных на одном процессоре. Каждая сопрограмма работает как бы параллельно с другими сопрограммами, как будто у нее есть собственный процессор. Такой подход упрощает программирование некоторых приложений, например, при решении задач моделирования. Такой подход также полезен для проверки программного обеспечения, которое потом будет работать на мультипроцессоре.

Ловушки

Ловушка (trap) – это особый тип вызова процедуры, который происходит при определенном условии. Обычно это очень важное, но редко встречающееся условие (например, переполнение). При возникновении такого условия поток управления переходит в какую-то фиксированную ячейку памяти. В этой ячейке памяти находится команда перехода к специальной процедуре – *обработчику системных прерываний*, которая выполняет какое-то определенное действие.

Важно то, что этот вид прерываний вызывается каким-то исключительным условием, вызванным самой программой и обнаруженным аппаратными средствами или микропрограммой. Конечно, обнаружить такую ситуацию можно и программными средствами, но при этом потребуется достаточно времени для постоянного программного контроля такой ситуации, а ловушка экономит время.

Прерывания

Прерывания – это изменения в потоке управления, вызванные не самой программой, а чем-либо другим и обычно связано с процессом ввода-вывода. Как и ловушка, прерывание останавливает работу программы и передает управление программе обработки прерываний, которая выполняет некоторое действие. После окончания этого действия программа обработки прерываний передает управление прерванной программе. Она должна заново начать прерванный процесс в том же самом состоянии, в котором она находилась, когда произошло прерывание. Это значит, что прежнее состояние всех регистров должно быть восстановлено.

Различие между ловушкой и прерыванием следующее: ловушки синхронны с программой, а прерывания асинхронны.

Для иллюстрации работы прерываний, рассмотрим пример: компьютер должен вывести на терминал строку символов. Программное обеспечение сначала собирает в буфер все символы, инициирует глобальную переменную prt, указывающую на начало буфера и устанавливает вторую глобальную переменную count, которая равна числу выводимых символов. Затем ПО проверяет, готов ли терминал и, если терминал готов, выводит первый символ. Начав процедуру вывода, центральный процессор освобождается и может запустить другую программу.

Через некоторое время символ появляется на экране. Теперь может начаться прерывание. Основные шаги в упрощенной форме следующие.

Действия аппаратного обеспечения:

1. Контроллер устройства устанавливает линию прерывания на системной шине.
2. Когда центральный процессор готов к обработке прерывания, он устанавливает символ подтверждения прерывания на шине.
3. Когда контроллер устройства узнает, что сигнал прерывания был подтвержден, он помещает небольшое целое число на информационные линии, что бы «представиться» (то есть показать, что это за устройство). Это число – номер прерывания.
4. Центральный процессор удаляет номер прерывания с шины и временно его сохраняет.
5. Центральный процессор помещает в стек счетчик команд и слово состояния программы.
6. Затем центральный процессор определяет местонахождение нового счетчика команд, используя номер прерывания в качестве индекса. Например, если размер счетчика команд составляет 4 байта, тогда номер прерывания n соответствует адресу 4n. Новый счетчик команд указывает на начало программы обслуживания прерываний для устройства, его вызвавшего. Иногда помимо этого загружается или изменяется слово состояния программы.

Действия программного обеспечения:

1. Программа обработки прерываний сохраняет все нужные ей регистры таким образом, чтобы их можно было восстановить позднее. Их можно сохранить в стеке или в системной таблице.
2. каждый номер прерывания разделяется всеми устройствами данного типа, поэтому в данный момент времени еще не известно, какое устройство вызвало прерывание. Номер устройства можно считать из какого-нибудь регистра.
3. Теперь можно считывать любую другую информацию о прерывании, например, коды состояния.
4. Если происходит ошибка ввода-вывода, ее нужно обработать здесь.
5. Глобальные переменные prt и count обновляются. Первая увеличивается на1 для того, чтобы показать следующий байт, а вторая уменьшается на 1, чтобы указать, что осталось ввести на один байт меньше. Тот символ, на который указывает в данный момент prt, копируется в выходной буферный регистр.
6. В случае необходимости выдается специальный код, который сообщает устройству или контроллеру прерывания, что прерывание обработано.
7. Восстанавливаются все сохраненные регистры.
8. Выполнение команды RETURN FROM INTERRUPT (выход из прерывания): возвращение центрального процессора в то состояние, в котором он находился до прерывания. После этого компьютер продолжает работать с того места, в котором ее приостановил.

С прерываниями связано понятие *прозрачности*. Когда происходит прерывание, производятся какие-либо действия и запускаются какие-то программы, но когда все закончено, компьютер должен вернуться точно в то же состояние, в котором был до прерывания. Программа обработки прерываний, обладающая этим свойством, называется прозрачной.

Если компьютер имеет только одно устройство ввода-вывода, тогда прерывания работают точно так, как было описано. Однако большой компьютер может содержать несколько устройств ввода-вывода, и возможна ситуация, что во время работы программы обработки прерываний другое устройство захочет произвести свое прерывание.

Для разрешения этой ситуации существуют два подхода.

Первый подход – для всех программ обработки прерываний в первую очередь (даже до сохранения регистров) предотвратить последующие прерывания.

Такой подход имеет недостаток, который заключается в том, что существуют устройства, которые не могут ждать. Если компьютер имеет подобные устройства, то необходимо приписать каждому устройству приоритет. Центральный процессор тоже должен иметь приоритет, который определяется по одному из полей слова состояния программы. Если устройство с некоторым приоритетом вызывает прерывание, то и программа обработки прерываний должна обладать таким же приоритетом.

4.7. INTEL IA-64

Со временем увеличивать скорость работы IA-32 становилось все сложнее и решением проблемы стало разработка совершенно новой архитектуры команд. Новая архитектура, которая разрабатывалась совместно компаниями Intel и Hewlett Packard получила название **IA-64.** Это полностью 64-битная машина. Самым первым процессором этого типа был процессор **Merced,** обладающий высокой производительностью.

Проблема с Pentium II

Основная проблема заключается в том, что IA-32 – старая архитектура команд с совершенно не подходящими для современной техники свойствами.

IA-32 – это архитектура, которая ориентирована на двухадресные команды. В настоящее время популярны архитектуры команд типа загрузка/сохранение, где обращение в память происходит только при выполнении команд записи или считывания, а действия над данными выполняются с использование регистров. Поскольку скорость работы процессоров растет гораздо быстрее, чем скорость обращения в память, то положение дел с IA-32 все больше ухудшается.

Архитектура IA-32 содержит небольшой и нерегулярный набор регистров. Из-за этого приходится постоянно записывать в память промежуточные результаты, что снижает скорость работы компьютера.

Из-за недостатка регистров возникает множество ситуаций зависимости (особенно WAR-зависимостей). Во избежание частых промахов кэш-памяти, поэтому команды приходится выполнять не по порядку. Однако, семантика языка требует, чтобы результаты выполнения команд записывались в регистры в строгом порядке. Для разрешения этой проблемы требуется сложное аппаратное обеспечение.

Для повышения быстродействия процессора необходима сильно конвейеризированная система (12 стадий). Однако это означает, что для выполнения команды необходимо 11 циклов. Следовательно, существенным становиться предсказание ветвлений. Даже при низкой вероятности ошибки предсказания производительность системы значительно снижается.

В результате получается, что огромное количество транзисторов в процессоре Pentium II используется для преобразования команд CISC в команды RISC, разрешения конфликтов, прогнозирование переходов и решения других задач подобного рода, оставляя малую часть на долю реальной работы, которая нужна пользователю. Поэтому компания Intel пришла к следующему выводу: нужно выбросить IA-32 и начать все заново.

Модель IA-64: открытое параллельное выполнение команд

Первой реализацией архитектуры IA-64 -64 был 64-битный процессор RISC. Поскольку IA-64 была разработана совместно с компанией Hewlett Packard, в ее основу легла архитектура PA-RISC. Merced – это двухрежимный процессор, который может выполнять и программы IA-32, и программы IA-64. Рассмотрим архитектуру IA-64.

Архитектура IA-64 – это архитектура типа загрузка/сохранение с 64-битными адресами и регистрами. Здесь имеется 64 регистра общего назначения. Все команды имеют фиксированный формат: код операции, два 6-битных поля для указания входных регистров, одно 6битное поле для указания выходного регистра и дополнительное 6-битное поле. Для параллельного выполнения различных операций существует много функциональных блоков. Такую архитектуру имеет большинство RISC-процессоров.

Отличительной особенностью IA-64 является идея о *пучке* связанных команд. Команды поступают группами по три штуки (рис. 4.27). Такая группа называется пучком. Каждый 128-битный пучок содержит три 40-битных команды фиксированного формата и 8-битный шаблон. Пучки могут быть связаны между собой, поэтому в пучке может быть более трех команд. Формат содержит информацию о том, какие команды могут выполняться параллельно. При такой системе компилятор может выделять блоки команд и сообщать процессору, что эти команды можно выполнять параллельно. Таким образом компилятор должен переупорядочивать команды, проверять,

нет ли взаимосвязей и т.д. вместо аппаратного обеспечения. Основная идея – работа упорядочивания и распределения RISC-команд передается от аппаратного обеспечения к компилятору. Такая технология называется EPIC (Explicitly Parallel Instruction Computing – технология параллельной обработки команд с явным параллелизмом).

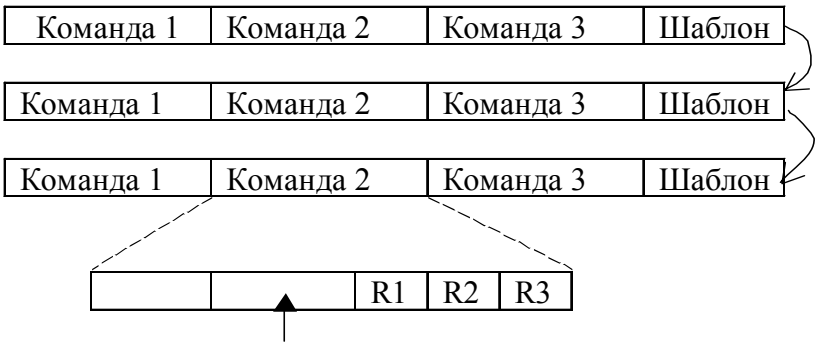


Рис. 4.27

Причины, по которым упорядочивание команд возложены на компилятор:

- Теперь всю работу компилятор, а аппаратное обеспечение можно упростить и увеличить объем кэш-памяти.
- Для любой программы распределение происходит только один раз на этапе компиляции.

Идея пучков может быть использована при создании семейств компьютеров: компьютеры с низкой производительностью могут запускать один пучок, а высокопроизводительные – несколько.

Предикация

Еще одна особенность архитектуры IA-64 – новый способ обработки условных переходов. Если бы была возможность избавится от команд переходов, то быстродействие процессора существенно повысилась бы. Полностью избавиться от операций переходов невозможно, но в архитектуре IA-64 используется специальная технология, называемая *предикацией*, которая позволяет сократить их число.

В современных компьютерах все команды являются безусловными в том смысле, что когда в центральный процессор попадает команда, то она выполняется. В архитектуре с предикацией команды содержат условия, которые сообщают, в каком случае надо выполнять команду, а в каком нет. Именно этот сдвиг к командам с предикацией позволяет избавиться от многих условных переходов. Вместо того, чтобы выбирать ту или иную последовательность безусловных команд, все команды сливаются в одну последовательность с предикацией, используя разные предикаты для различных команд.

Например.

Оператор if	Ассемблер	Команда с предикацией
If (R1==0) *R2=R3;*	*CMP R1,0* *BNE L1* *MOV R2,R3* *L1:*	*CMOVZ R2, R3, R1*

В архитектуре IA-64 с предикатными регистрами связаны и команды сравнения, и арифметические команды и т.д. Команды с предикацией могут помещаться в конвейер без каких-либо дополнений и простаиваний.

А архитектуре IA-64 предикация происходит следующим образом. Каждая команда действительно выполняется, и в самом конце конвейра, когда уже нужно сохранять результат в выходной регистр, производится проверка, истинно ли предсказание. Если истинно – результат записывается в выходной регистр, а если ложно – запись не происходит.

Спекулятивная загрузка

Идея заключается в том, чтобы компилятор помещал команды считывания из памяти в более ранние позиции относительно других команд. Поскольку эти команды начинаются раньше, чем нужно, то они и могут завершиться до того, как потребуются результаты. Компилятор вставляет команду CHECK в том месте, где ему нужно получить значение определенного регистра. Если значение есть, то выполнение программы продолжается; в противном случае команда простаивает.

Если все эти нововведения функционируют, то процессор Merced действительно будет мощным.

Однако:

- Такая продвинутая машина не создавалась раньше.

- Придется составлять компиляторы для IA-64 – а это дело сложное. Многочисленные исследования в параллельном программировании оказались не очень успешными. Если компилятор не сможет связывать команды в длинные пучки, то большого эффекта не получится.
- Для реализации этой идеи должна существовать полностью 64-битная операционная система. А это значит, что всем придется перейти на Windows NT или UNIX. Этот переход не безболезненный для пользователя.
- Многие будут судить о IA-64 по тому, как работает на нем старые 16-битные игры. А тут никакая предикация и спекулятивная загрузка не помогут. А если нет разницы – зачем платить больше?
- Кроме того, другие производители тоже будут выпускать новую продукцию.

Вероятно, пройдет еще много времени, прежде чем IA-64 будет доминировать на рынке подобно архитектуре IA-32.

5. УРОВЕНЬ ОПЕРАЦИОННОЙ СИСТЕМЫ

Операционная система – это программа, которая добавляет ряд команд и особенностей к тем, которые обеспечиваются уровнем команд. Обычно операционная система реализуется программными средствами, но нет никаких веских причин, по которым ее нельзя было бы реализовать в аппаратном обеспечении.

Хотя уровень операционной системы и уровень архитектуры команд абстракты (это не аппаратное обеспечение), между ними есть важное различие. Набор уровня операционной системы – это полный набор команд, доступных для прикладных программистов. Он содержит практически все команды более низкого уровня, а также новые команды, которые добавляет операционная система. Эти команды называются *системными вызовами*. Уровень операционной системы всегда интерпретируется. Поскольку имеется отдельный курс «Системное программирование», то в нашем курсе обратим внимание только на следующих важных особенностях: виртуальная память, файл ввода-вывода, параллельная обработка.

5.1. ВИРТУАЛЬНАЯ ПАМЯТЬ

В первых компьютерах память была маленькой по объему. Например, компилятор ALGOL был написан для компьютеров с объемом памяти в 1024 слова. Система с разделением времени на PDP-1 работала на машине с памятью в 4096 18-битных слов для операционной системы и пользовательских программ.

Решением проблемы нехватки памяти явилось использование вспомогательной памяти, например, диска. Программист делил программу на несколько частей, т.н. *оверлеев,* каждый из которых помещался в память. Программист отвечал за разбиение программы на оверлеи и решал, в каком месте вспомогательной памяти следует разместить каждый оверлей,

контролировал передачу оверлеев между основной и вспомогательной памятью и вообще сам управлял процессом без какой-либо помощи компьютера.

В 1961 году группа исследователей из Манчестера (Англия) предложила метод автоматического выполнения процесса наложения, при котором программист мог вообще не знать об этом процессе. Этот метод, который сейчас называется виртуальной памятью, имел очевидное преимущество, поскольку освобождал программиста от кропотливой и нудной работы.

Проблема усложняется при переходе к мультипрограммным системам, так как в них ОП одновременно используется для нескольких программ (задач). В таких системах необходимо исключить несанкционированное воздействие одних программ на другие. Это достигается за счет механизма защиты памяти.

Эффективное распределение ресурса памяти между программами не может быть статическим, т.е. производиться предварительно, до пуска программы. Необходимо распределять память между программами динамически, непосредственно в ходе вычислительного процесса, т.е. осуществлять динамическое распределение памяти, при этом должна обеспечиваться «прозрачность» этого механизма для программистов. Динамическое распределение памяти не должно приводить к дроблению ее свободного пространства–фрагментации памяти, затрудняющему ее использование. Это достигается использованием одноуровневой виртуальной памяти, допускающей адресацию на все адресное пространство, размерность которого определяется полем адресного кода команды или базового регистра.

Защита памяти

Чтобы воспрепятствовать разрушению одних программ другими, достаточно защитить область памяти данной программы от попыток записи в нее со стороны других программ, а в некоторых случаях и своей программы. Это т.н. защита от записи. При этом чтение областей памяти допускается.

В других случаях, например, в целях защиты информации необходимо иметь возможность запрещать другим программам производить как запись, так и считывание в данной области памяти. Это защита от записи и чтения.

Для облегчения процесса отладки программ желательно выявлять и такие характерные ошибки в программах, как попытки использовать данные вместо команд и команд вместо данных в собственной программе.

Отметим следующие варианты защиты при различных операциях с памятью:

2. задается отношение к областям памяти чужой программы, определяющее, относится защита только к операции записи или к любому обращению к памяти;
3. задается одно из следующих отношений к области памяти собственной программы:

- разрешается полный доступ к данному блоку памяти;
- разрешается только считывание;
- разрешается обращение только через счетчик команд;
- разрешается обращение за исключением счетчика команд.

Если нарушается защита памяти, исполнение программы приостанавливается и вырабатывается запрос прерывания по нарушению защиты памяти.

Защита памяти может быть организована различным образом, при этом организация защиты не должна заметно снижать производительность системы и требовать слишком больших аппаратурных затрат.

Защита отдельных ячеек памяти. С каждой ячейкой памяти связывается дополнительная информация о защите, например, в простейшем случае это может быть один бит, указывающий можно ли в данную ячейку записывать информацию или нет.

Как правило, в современных ВС защищаются не отдельные ячейки, а целые области памяти (блоки, сегменты).

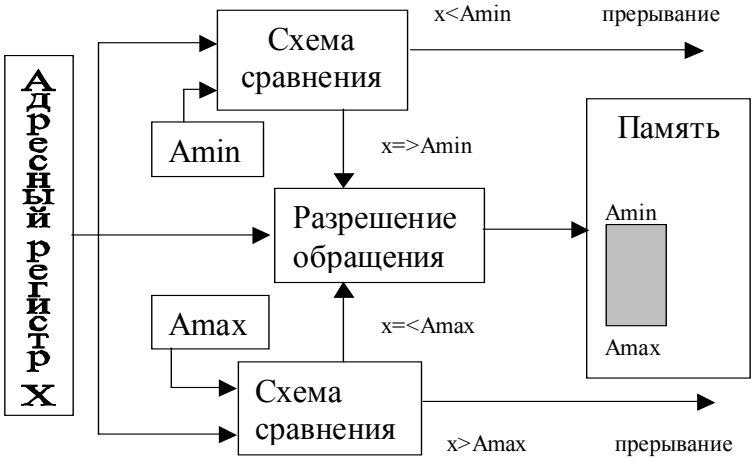


Рис. 5.1

Метод граничных регистров (рис.5.1) состоит во введении двух граничных регистров, указывающих верхнюю и нижнюю границы области памяти, куда программа имеет право доступа. При каждом обращении памяти проверяется, находится ли используемый адрес в установленных границах. При выходе за границы, обращение к памяти не происходит, вырабатывается запрос на прерывание, передающий управление ОС. Содержимое граничных регистров устанавливается ОС перед тем, как для очередной программы начинается активный цикл.

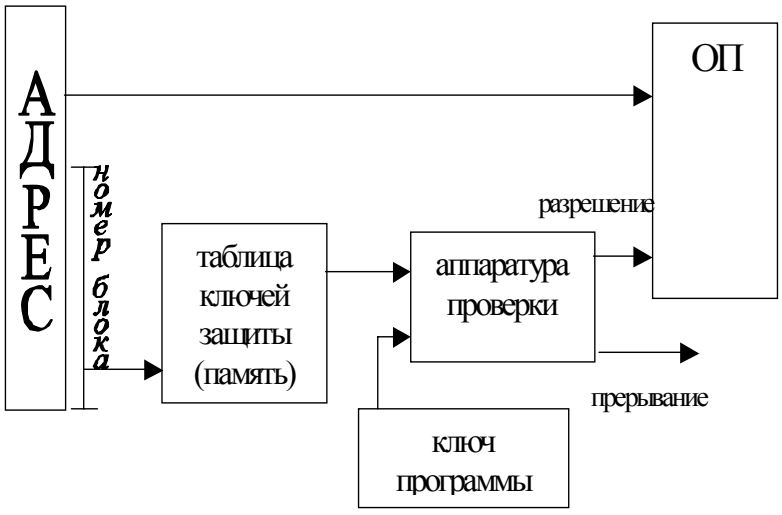


Рис. 4.3

Более гибким является метод, который условно назовем методом ключей защиты (терминология из IBM 360).

Память в логическом отношении делится на блоки (сегменты). Каждому блоку ставится в соответствие некоторый код, который назовем

ключом защиты. Каждой программе также ставится в соответствие некоторый код, который назовем ключом программы. Доступ программы к данному блоку памяти определяется аппаратурой процессора на основе анализа ключа программы и ключа защиты памяти (рис. 5.2). Конкретная реализация этого метода для каждой ВС является оригинальной.

Организация виртуальной памяти

Принцип виртуальной памяти предполагает, что пользователь при подготовке своей программы имеет дело не с физической ОП, действительно имеющейся в ВС, и имеющей некоторую фиксированную емкость, а с виртуальной одноуровневой памятью, объем которой равен всему адресному пространству (В x86 это 4 Гбайт).

На всех этапах подготовки программ, включая загрузку в ОП, программа представляется в виртуальных адресах и лишь при самом исполнении машинной команды производится преобразован виртуальных адресов в реальные адреса действующей памяти (физические адреса).

Физическая и виртуальная память разбивается на блоки, называемые страницами, содержащими одно и тоже число байт. Страницам виртуальной и физической памяти присваиваются номера. Каждая физическая страница способна хранить одну из виртуальных страниц. Порядок расположения (нумерация) байт в виртуальной и физической страницах сохраняется одним и тем же.

Соответствие между виртуальной и физической памятью устанавливается так называемой страничной таблицей, приведенной на рис. 5.4.

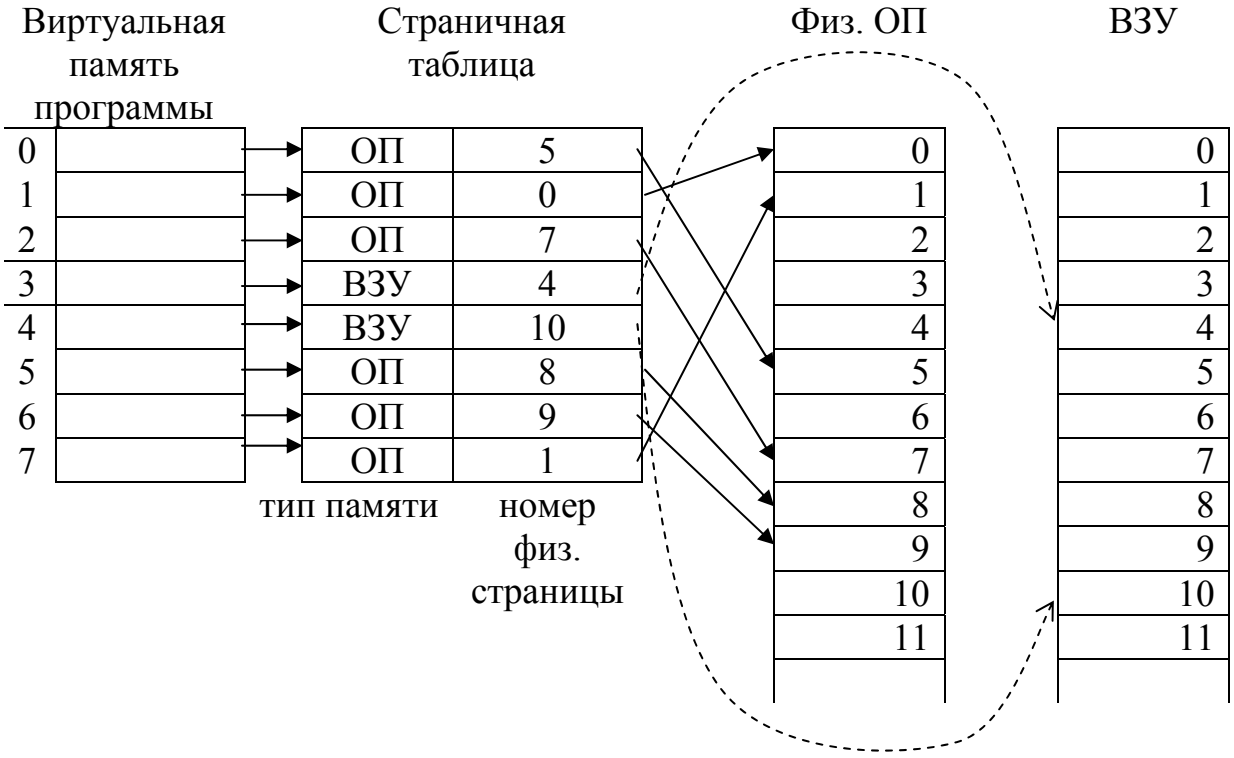


Рис. 5.4

Физические страницы могут содержаться в текущий момент времени как в оперативной, так и во внешней памяти. Страничная таблица формируется ОС. Процедура обращения к памяти состоит в том, что номер виртуальной страницы извлекается из адреса и используется для входа в страничную таблицу, которая указывает номер соответствующей физической страницы. Этот номер вместе с номером байта, взятым непосредственно из виртуального адреса представляет собой физический адрес, по которому происходит обращение к ОП.

Если страничная таблица указывает на размещение страницы в ВЗУ, то обращение к ОП не может состояться немедленно, ОС должна организовать передачу из внешней памяти в ОП нужной страницы.

Для каждой из программ, обрабатываемых в мультипрограммном режиме, организуется своя виртуальная память и создается своя страничная таблица, при этом все программы делят между собой общую физическую память.

Вызов страниц по требованию и рабочее множество

При обращении к адресу страницы, которой нет в основной памяти, происходит *ошибка из-за отсутствия страницы.* В случае такой ошибки ОС должна считать нужную страницу из памяти, ввести новый адрес в таблицу страниц, а затем повторить команду, которая вызвала ошибку.

Такой метод работы с виртуальной памятью называется *вызовом страниц по требованию*. Особенность: страницы вызываются по мере необходимости, а не заранее. Вопрос о целесообразности использования метода вызова страниц по требованию имеет смысл только в начале запуска программы, т.к. в процессе работы нужные страницы будут собраны в основной памяти. Если компьютер работает в режиме разделения времени и процессы откачиваются обратно, то каждая программа будет запускаться многократно. Для каждой программы распределение памяти уникально и при переключении программ оно изменяется. Поэтому в системах с разделением времени такой подход неприемлем.

Альтернативный подход заключается в том, что большинство команд обращается к адресному пространству неравномерно. Обычно большинство обращений относится к небольшому числу страниц. В каждый момент времени t существует набор страниц, которые использовались за последние k обращений. Деннинг назвал этот набор страниц *рабочим множеством*.

Политика замещения страниц

В идеале рабочее множество страниц можно хранить в памяти. Однако места не хватает. Если программа обращается в память за страницей, то

новой странице необходимо освободить место. Возникает необходимость определения страницы, которую можно отправить в память.

Алгоритм *LRU (Least Recently Used – алгоритм удаления наиболее давно используемых элементов)*. По этому алгоритму удаляется та страница, которая использовалась наиболее давно. В целом алгоритм работает достаточно хорошо, но возможны и исключения. Например, выполняется цикл, для реализации которого требуется n страниц, а в памяти может разместиться только n-1. После выполнения n-1 страницы происходит обращение за недостающей n-й, а выбрасывается первая, которая потребуется сразу же после выполнения n-й. Затем считывается первая, а выбрасывается вторая, которая тут же оказалась нужна и т.д. Очевидно, что в такой ситуации алгоритм LRU не работает; однако, в такой ситуации другие алгоритмы тоже работать не будут. Выход – расширение рабочего множества или учет малого объема основной памяти при составлении программ.

Алгоритм FIFO (First-in First-out – первым пришел, первым ушел) удаляет ту страницу, которая первая поступила в память. В такой ситуации возможно ошибочное удаление наиболее «активной» страницы.

Для ускорения процедуры удаления страниц может использоваться проверка на модификацию ее содержимого. Если в страницу не производилась запись нет необходимости терять время на запись ее на диск. Такую страницу можно просто заместить и внести изменение только в таблицу страниц.

Размер страницы и фрагментация

Если данные или программа пользователя занимает не целое число страниц, то на последней странице возникает свободное пространство, которое не может быть занято другим пользователем. Такой эффект называется внутренней фрагментацией.

Для сокращения объема неиспользованного пространства можно использовать страницы меньшего размера, но это приводит к увеличению объема таблицы страниц. Кроме того, маленькие страницы снижают эффективность пропускной способности диска, т.к. увеличивается удельное время на поиск информации и вращение диска.

Сегментация

До сих пор предполагалось, что речь идет об одной виртуальной памяти. Возможно использование двух или нескольких виртуальных адресных пространств. Например, компилятор может иметь несколько таблиц, которые создаются в процессе компиляции:

- таблица символов, которая содержит имена и атрибуты переменных;
- Исходный текст;

- Таблица, содержащая все используемые целочисленные константы и константы с плавающей точкой;
- Дерево, содержащее синтаксический анализ программы;
- Стек, используемый для вызова процедур в компиляторе.

В рассмотренной ситуации, а также при работе в мультипрограммном режиме необходимо иметь возможность создания независимых адресных пространств. Такие адресные пространства называются сегментами. Каждый сегмент состоит из линейной последовательности адресов от 0 до некоторого максимума. Длина сегмента может быть любой (в допустимых пределах). Кроме того, длина сегмента может меняться в процессе выполнения программы.

Чтобы определиться в таком двухмерном пространстве необходимо указать номер сегмента и адрес внутри сегмента.

Сегментированная память имеет ряд преимуществ при работе с памятью.

- Если каждая процедура занимает отдельный сегмент, у которого первый адрес равен 0, то связывание процедур, которые компилируются отдельно сильно упрощается. Для обращения к I-му слову n-й процедуры используем адрес (n,i).
- Если процедура в некотором сегменте изменялась и перекомпилировалась, то остальные можно не трогать.
- Сегментация облегчает разделение общих процедур и данных между несколькими программами.
- Разные сегменты могут иметь разные виды защиты. Например, кодовый сегмент допускает только считывание выполнение, для массивов данных – запись и считывание и т.д.

Реализация сегментации

Сегментацию можно организовать одним из двух способов. Это подкачка и разбиение на страницы.

При первом подходе некоторый набор сегментов находится в памяти в данный момент. Если происходит обращение к сегменту, которого нет в данный момент в памяти, этот сегмент переносится в память. Если для него нет места в памяти, один или несколько сегментов нужно сначала записать на диск. В каком-то смысле подкачка сегментов очень похожа на вызов страниц по требованию: сегменты загружаются и удаляются только в случае необходимости.

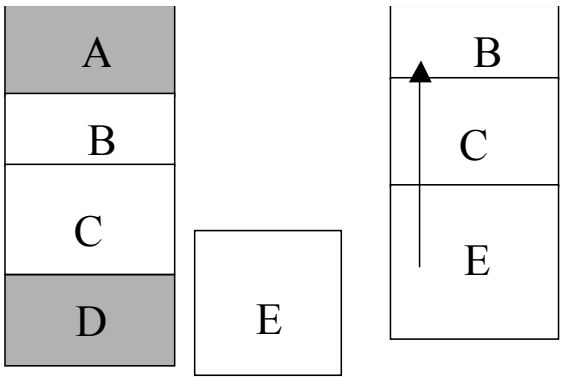


Рис. 5.5

Отличие сегментации от разбиения на страницы: размер страниц фиксирован, а размер сегментов – нет. Поэтому при сегментации может возникнуть эффект *внешней фрагментации*, которая иллюстрируется рисунком 5.5. Для ввода нужной программы может понадобиться сдвиг содержимого памяти. На рис. 5.3 показано распределение памяти между программами ABCD, из которых две (A и D) являются в данный момент неактивными и могут быть удалены во внешнюю память. Если вновь вводимая программа E больше, чем имеющийся после удаления A и D участок памяти, то для ее размещения необходимо сдвигать программы B и C. Это перемещение связано с потерей времени. Подобная схема использовалась в ОС Windows 3.x в так называемом стандартном режиме работы. Если на уплотнение памяти требуется слишком много времени, нужен специальный алгоритм для определения, какую именно «дырку» лучше использовать для определенного сегмента. Для этого требуется список адресов и размеров всех «дырок». Популярный алгоритм *оптимальной подгонки* выбирает самую маленькую «дырку», в которую помещается сегмент.

Второй способ реализации сегментации – разделение каждого сегмента на страницы фиксированного размера и вызов страниц по требованию. Для того, чтобы разбить сегмент на страницы необходимо иметь отдельную таблицу страниц для каждого сегмента.

MULTICS (Multiplexed Information and Computing Service – служба общей информации и вычислений) – это древняя операционная система, которая совмещала сегментацию с разбиением на страницы. Она была разработана в МТИ совместно с компаниями Bell Labs и General Electric. Адреса в MULTICS состоят из двух частей: номера сегмента и адреса внутри сегмента. Для каждого процесса существовал сегмент дескриптора, который содержал дескриптор для каждого сегмента. Когда аппаратное обеспечение получало виртуальный адрес, номер сегмента использовался в качестве индекса в сегменте дескриптора для определения дескриптора нужного сегмента. Дескриптор указывал на таблицу страниц, что позволяло разбивать на страницы каждый сегмент обычным способом. Для повышения производительности недавно используемые комбинации сегмента и страницы помещались в ассоциативную память из 16

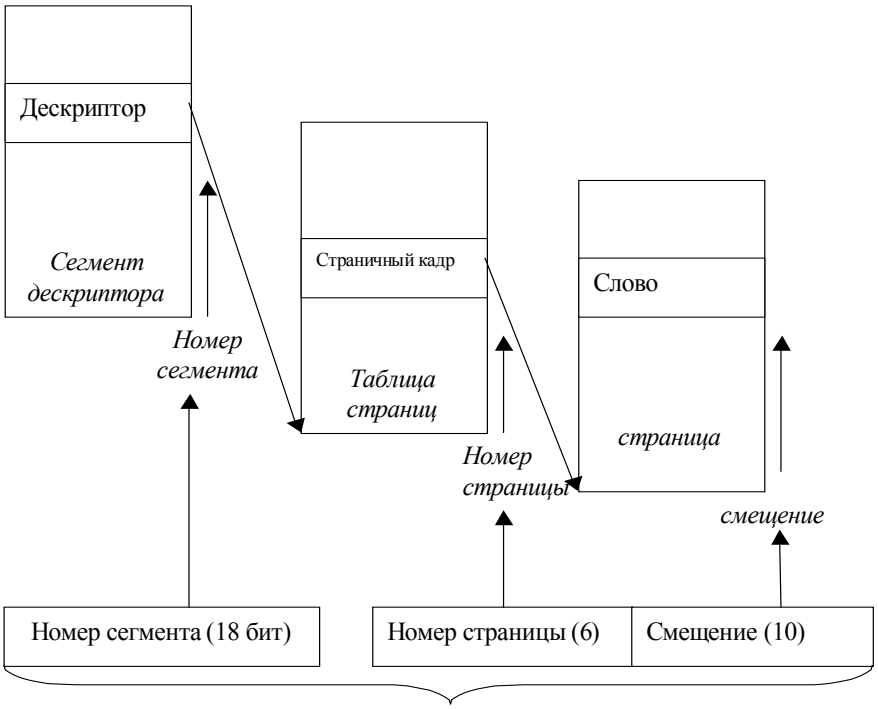


Рис. 5.6.

элементов. Операционная система MULTICS уже давно не применяется, управление памятью в процессорах Intel, начиная с 386-го, очень похожа на эту систему.

Управление памятью в процессоре Pentium II

Каждый сегмент в системе характеризуется специальной 64-разрядной (8 байт) структурой данных, называемой дескриптором сегмента. В описание сегмента включается базовый адрес сегмента, размер сегмента, тип, уровень привилегий и дополнительная информация.

Формат дескриптора приведен на рис. 5.7.

Рис.5.7

Рассмотрим значения отдельных полей дескриптора.

Базовый адрес занимает 32 разряда (2,3,4,7 байты). Определяет начальный адрес сегмента в линейном адресном пространстве в 4 Гбайт. Именно этот адрес сформирует процессор при задании нулевого смешения.

Предел. Размер 20 бит. Задает размер сегмента. Этот размер определяется так. Если значение бита G = 0 (бит гранулярности), то размер сегмента в байтах равен значению этого поля минус 1. Если G = 1, то поле предела задает предел не в байтах, а в страницах (4 Кбайт). При G = 0 значение размера максимум 1 Мбайт, при G = 1 максимум размера сегмента 4 Гбайта. Смещение в команде не должно превышать размер сегмента.

Права доступа. (Access Rights) Это байт 5 дескриптора сегмента и часто называется байтом AR.

Если бит присутствия P установлен в 1, то сегмент доступен. Когда P = 0, то процессор отвергает все последующие попытки использовать дескриптор и все определяемое этим дескриптором адресное пространство как бы пропадает. Если P=0, то процессор игнорирует остальные поля дескриптора.

Двухбитное поле DPL определяет уровень привилегий, ассоциируемый с той областью памяти, которую описывает дескриптор. Принимает значения

от 0 (максимум) до 3 (минимум). Привилегии будем рассматривать при изучении механизмов защиты.

Бит S (системный) установлен в 1 для сегментов памяти и равен 0 для специальных системных объектов.

Трехбитное поле типа определяет использование сегмента:

000 -	данные, только чтение;
001 -	данные, чтение/запись;
010 -	стек, только чтение (на практике не применяется);
011 -	стек, чтение/запись;
100 -	код, только выполнение;
101 -	код, выполнение/чтение
110 -	подчиненный сегменту код, только выполнение;
111 -	подчиненный сегменту код, выполнение/чтение

Например, в x86 нельзя загружать сегменты с типом 0-3 сегменты данных и заставить их выполняться как код. Нарушение этих правил вызывает особый случай защиты (а в 8086 допустимо).

Бит доступа A. Этот бит процессор автоматически устанавливает в 1, когда осуществляется обращение к тому сегменту памяти, который определяет дескриптор. Используется для выявления сегментов, к которым не было обращения.

Дополнительные биты. Бит D размера по умолчанию обеспечивает совместимость с 16-разрядным процессором 80286. Когда бит D=0, считается, что находящиеся в сегменте операнды или команды относятся к 16-разрядным, если D=1, то сегмент считается 32-разрядным и в нем используется 32-разрядные команды и 32-разрядная адресация.

Бит 53 – резервный.

Бит пользователя U предназначен для использования системными программистами по их усмотрению, а процессор игнорирует этот бит.

Поле права доступа (AR)

Рассмотрим более подробно это поле. Отметим, что не все комбинации допустимы. Если возникает запрещенная комбинация, то процессор фиксирует особый случай

Бит P присутствия служит флажком, показывающим возможность доступа к сегменту. При P=0 процессор игнорирует остальные поля, за исключением поля AR. В этой ситуации системный программист может распоряжаться остальными байтами дескриптора по своему усмотрению. Например, в них можно закодировать нахождение выгруженного из памяти сегмента на диске.

Поле типа играет важную роль в определении сегментов памяти. Каждый бит этого поля несет свою смысловую нагрузку рис. 5. 8.

7		S	тип				
P	DPL	1	1	C	R	A	Сегмент кода
P	DPL	1	0	ED	W	A	Сегмент данных
P	DPL	0	x	x	x	x	Системный объект

Рис. 5.8

Бит 3 AR различает сегменты кода (1) и данных (0). Для сегментов кода бит 2 называется битом подчинения C (об этом будем говорить при рассмотрении механизма защиты), а бит 1 называется битом считывания R. Он показывает возможность считывания кода как данных с помощью префикса замены сегмента.

Для сегментов данных бит 2 называется битом расширения вниз ED, который различает сегменты стека ED=1 и сегменты собственно данных ED=0. Бит 1 записи W показывает возможность модификации содержимого сегмента посредством записи в него W=1.

Отметим, что сегменты типов 0 могут хранить только данные и командами FAR JMP и FAR CALL невозможно заставить процессор выполнить данные как код.

Сегменты стека отличаются от сегментов данных интерпретацией поля предела. В сегментах данных разрешенный диапазон адресов простирается от базового адреса до базового адреса плюс предел. Смещение, которое больше предела вызывает особый случай защиты. С сегментах стека ситуация оказывается противоположной в том смысле, что поле предела определяет не адресную область сегмента. Здесь все смещения должны быть строго больше предела, а если смещение меньше или равно пределу фиксируется особый случай нарушения стека.

Максимальный адрес стека: если D=0, то Amax = Абаз + 0FFFF (64 Кб-1); если D=1, то Amax = Абаз + 0FFFFFFFF (4 Гб-1).

На рис. 5.9 показаны сегмент стека и сегмент данных, имеющих предел 1FFFh (G=0, для упрощения считаем сегменты 16-битными). Для сегмента стека первый адресуемый байт находится по смещению 2000H.

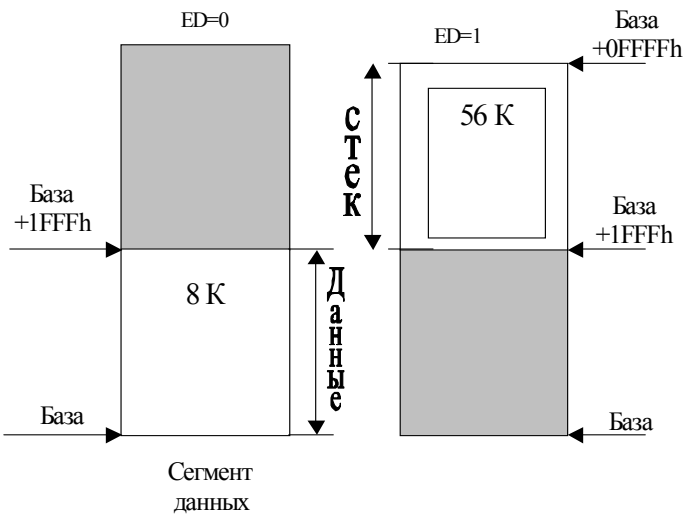


Рис. 5.9

Такая интерпретация предела сегмента стека объясняется тем, что указатель стека обычно инициализируется на наибольший адрес в сегменте, а

затем, по мере включений в стек, содержимое указателя стека уменьшается. Если использовать при этом обычные сегменты данных, то при нехватке стекового пространства будут возникать трудности с увеличением размера стека. В сегменте же стека достаточно просто уменьшить значение предела.

Типы сегментов 4–7 определяют исполняемые сегменты, т.е. сегменты, в которых находится код. В таких сегментах можно разрешить/запретить считывание. Отметим, что запись в сегмент кода никогда не допускается, поэтому организовывать трюки с самомодифицированными программами невозможно.

Бит подчинения C, установленный в состояние 1, позволяет намеренно лишить соответствующий сегмент кода защиты по уровню привилегий. (используется для создания библиотек, доступных всем программам, независимо от уровня привилегий).

Дескрипторные таблицы

Область памяти, предназначенная для хранения дескрипторов, называется дескрипторной таблицей. Она представляет собой массив из 8–байтных элементов – дескрипторов. Порядок размещения дескрипторов в таблице не играет роли, а максимальное число дескрипторов составляет 8192, соответственно максимальный размер таблицы 64 Кбайт. В процессоре предусмотрены дескрипторные таблицы трех типов.

Глобальная дескрипторная таблица (GDT). Является главной общесистемной таблицей дескрипторов. Все программы (задачи), выполняющиеся в системе, могут использовать эту таблицу. Местонахождение этой таблицы определяет специальный регистр GDTR. В нем находится 32–разрядное поле линейного базового адреса и 16–разрядное поле предела L=8*N-1, где N – число дескрипторов.

Дескрипторная таблица прерываний (IDT). Является общесистемной и содержит дескрипторы специальных системных объектов, называемых «шлюзами» (gate), которые определяют точки входов в процедуры обработки прерываний и особых случаев. Системный регистр IDTR служит для локализации этой таблицы (аналогично по структуре с GDTR).

Локальная дескрипторная таблица (LDT). Для каждой задачи в дополнение к GDT можно простроить LDT. Она определяет сегменты, доступные только этой конкретной задаче. Для локализации LDT служит 16-разрядный регистр LDTR, который содержит только селектор сегмента, содержащего LDT.

Загрузка регистров GDT и IDT осуществляется только один раз в ходе подготовки процессора к работе в защищенном режиме. Предварительно готовятся сами таблицы, после чего командами LGDT mem48 и LIDT mem48 из памяти загружаются 48-битная структура. (Команды SGDT mem48 и SIDR mem48 сохраняют содержимое регистра).

Таблицы LDT не являются обязательными и создаются по мере необходимости. Для хранения LDT применяются сегменты памяти, а

дескрипторы этих сегментов хранятся в таблице GDT. Селектор из регистра LDTR выбирает в таблице GDT нужный дескриптор. Для ускорения работы этот дескриптор загружается в отдельный «теневой» регистр, ассоциированный с LDT автоматически всякий раз, когда содержимое LDTR меняется.

Команда LLDT reg16/mem16 загружает селектор в регистр LDTR, аналогично, команда SLDT reg16/mem16 сохраняет содержимое LDTR.

Селекторы сегментов

Напомним, что отправной точкой доступа к дескриптору служит содержимое сегментного регистра, называемого селектором. Если в процессоре 8086 содержимое сегментного регистра служит просто старшими битами начального адреса сегмента, то в рассматриваемом процессоре селектор определяет сегмент памяти косвенно, через дескрипторную таблицу. Формат селектора приведен на рис. 5.10.

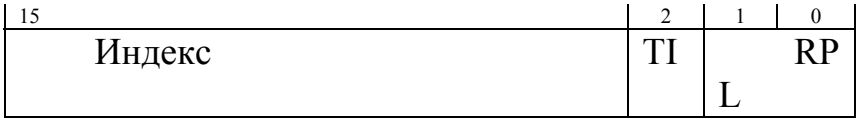


Рис. 5.10

Двухразрядное поле запрашиваемого уровня привилегий RPL не участвует в выборе дескриптора, а привлекается для контроля привилегий в механизме защиты.

Бит индикатора таблицы TI показывает, из какой дескрипторной таблицы выбирается дескриптор. Если TI равен 0, то это таблица GDT; при TI равном 1 – таблица LDT.

Старшие 13 бит селектора определяют нужный дескриптор в соответствующей таблице.

Отметим, что первый элемент таблицы GDT зарезервирован, тем не менее селектор может содержать значения индекса равное 0 и TI также равное 0. Такой селектор называется пустым селектором. Пустой селектор можно загружать в сегментные регистры ES, DS, GS, PS, но обращаться к памяти с ними нельзя.

Формирование адреса.

Логический адрес – это адрес, которым обычно оперирует программное обеспечение. Он формируется из двух величин: 16-битного селектора (указателя) сегмента (берется из соответствующего сегментного регистра) и 32 (16) – разрядного смещения относительно начала сегмента. Логический адрес существует только внутри программного обеспечения. Его преобразование в физический адрес для преобразования в физический адрес

осуществляется при помощи достаточно сложного механизма, функционирование которого зависит от текущего режима работы процессора.

Линейный адрес – формируется из логического и предназначен для обращения к линейному (непрерывному и несегментированному) пространству объемом 2^{32} байт. При отключении страничного механизма линейный адрес полностью совпадает с физическим, а способ его формирования зависит от текущего режима работы процессора.

Физический адрес – передается на внешнюю шину для обращения к ячейкам памяти. Адресуется 2^{32} байт в обычном режиме и до 2^{36} байт в случае поддержки механизма расширения физического адреса (CR4.PAE).

Процессор может работать в одном из двух режимов работы и переключаться из одного режима в другой достаточно быстро.

Real Address Mode – режим реальной адресации (или просто реальный режим) полностью совместим с 8086. В этом режиме возможна адресация до 1 Мб физической памяти. Механизм формирования физического адреса в этом режиме предельно упрощен и иллюстрируется рис. 5.11.

Protected Virtual Address Mode – защищенный режим виртуальной адресации (или просто защищенный режим). В этом режиме процессор позволяет адресовать до 4 Гб физической памяти. Действия механизма образования физического адреса основано на использовании дескрипторных таблиц. Схема формирования адреса приведена на рис. 5.12.

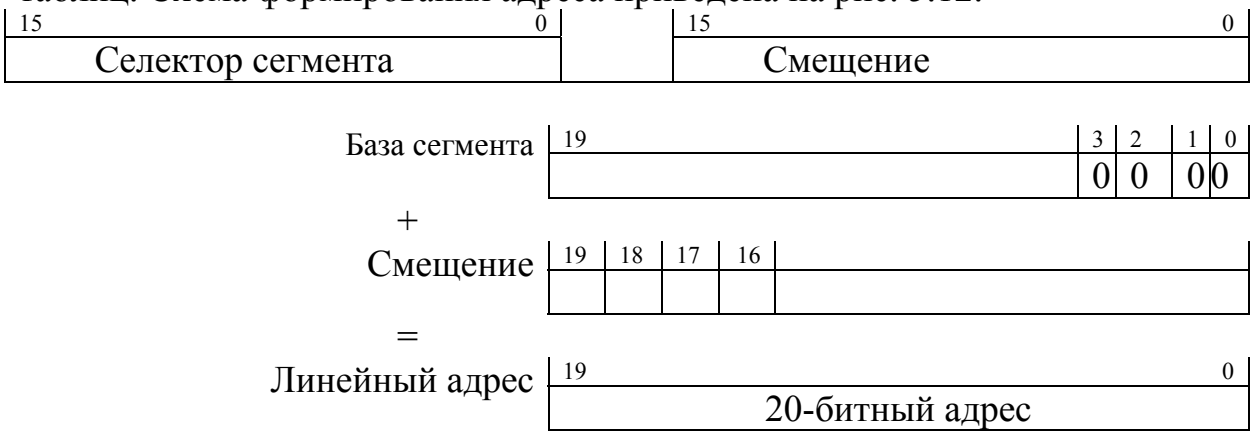


Рис. 5.11

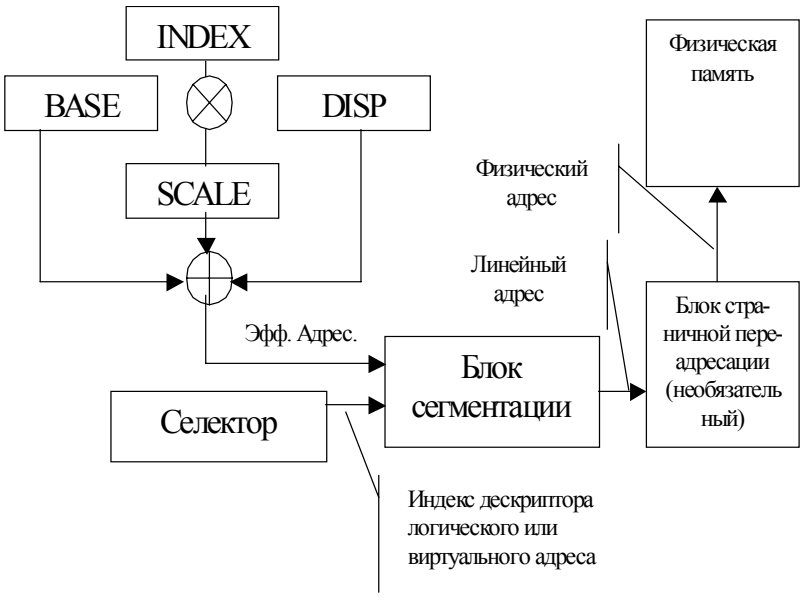


Рис. 5.12

Существенным дополнением является Virtual 8086 Mode – режим виртуального процессора 8086. Этот режим является особым состоянием задачи защищенного режима, в котором процессор функционирует как 8086. На одном процессоре в таком режиме могут параллельно выполняться несколько задач с изолированными друг от друга ресурсами. При этом исполь-

зование физического адресного пространства памяти управляется механизмами сегментации и трансляции страниц. Попытки выполнения недопустимых команд, выхода за пределы отведенного пространства памяти контролируются системой защиты.

Процессоры, начиная с Pentium и некоторых моделей 486, имеют особый режим системного управления System Management Mode (SMM), в котором процессор выходит в иное, изолированное от остальных режимов пространство памяти. Этот режим используется в служебных и отладочных целях.

Рассмотрим пример.

Пусть выполняется команда:

MOV EAX, [ECX][ECI+20H].

Эта команда обращается к текущему сегменту данных, селектор которого находится в регистре DS. Пусть DS=00 … 0110xx B. Бит TI = 0, дескриптор находится в GDT, номер дескриптора 3. Выполнение команды включает в себя такие действия:

1. Образовать т.н. эффективный адрес EA = (ECX)+(ESI)+20H.
2. Выбрать третий дескриптор из GDT. Для этого необходимо обратиться к полю базы GDTR, прибавить к нему индекс, умноженный на 8 и считать в процессор дескриптор по этому адресу.
3. Просуммировать эффективный и базовый адрес сегмента из дескриптора. В результате получим т.н. линейный адрес операнда.
4. Обратиться к памяти по линейному адресу и передать двойное слово в регистр EAX.

Описанные выше операции обращения к сегменту данных требуют считывания из памяти 8-байтного дескриптора из GDT (а при обращении к LDT обращений к памяти будет еще больше). Это может существенно снизить производительность процессора. Во избежание этого в процессоре используется кэширование дескрипторов. Оно опирается на тот факт, что обращения к памяти производятся гораздо чаще, чем изменение содержимого сегментных регистров. Для этого с каждым сегментным регистром связывается «теневой» регистр (аналогично LDTR), в котором хранится дескриптор, соответствующий содержимому соответствующего сегментного регистра. Когда программа загружает селектор в сегментный регистр, процессор автоматически считывает нужный дескриптор в «теневой» регистр.

При загрузке сегментного регистра процессор осуществляет несколько проверок. Часть проверок связана с механизмом защиты, остальные предотвращают загрузку бессмысленных селекторов:

- проверяется поле Index селектора на предмет нахождения в пределах таблицы, определяемой TI;
- при загрузке DS, ES, FS, GS дескриптор должен разрешать считывание из сегмента;
- для SS в сегменте должны быть разделены и чтение, и запись;
- для CS сегмент должен быть исполняемым;
- бит присутствия P должен быть равен 1.

Если хотя бы одна проверка дает отрицательный результат, формируется особый случай и загрузка селектора не производится.

Прикладной программист по селектору не может узнать, какое адресное пространство определяет сегментный регистр и не должен модифицировать селекторы, поскольку ему не известно содержимое дескрипторных таблиц.

Тем не менее прикладной программист может получить некоторую информацию о дескрипторе.

Команда lar reg, reg16/mem16 загружает в регистр получателя права доступа селектора, адресуемого источником (байт AR, байты G, D, X, U).

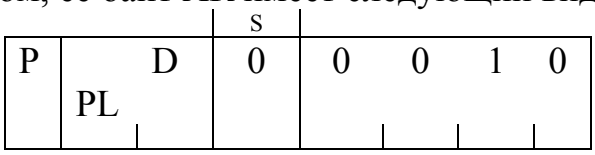
Предел сегмента можно получить командой lsl reg31, reg16/mem16.

Можно проверить, допускает ли сегмент считывание или запись:

verr reg16/mem16, verw reg16/mem16.

Эти команды устанавливают ZF=1 если запрашиваемый доступ разрешен.

Отметим в заключение, что локальная дескрипторная таблица является системным объектом, ее байт AR имеет следующий вид:



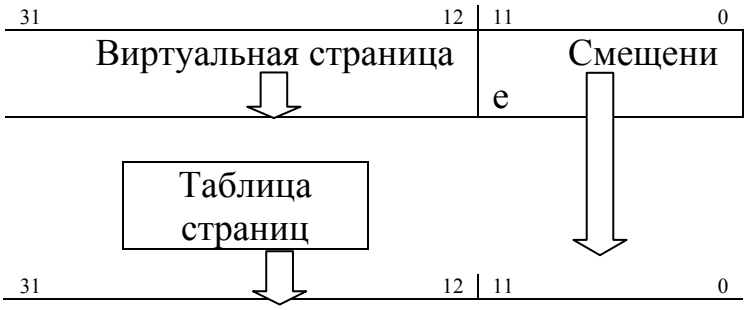
Естественно, что в регистр LDTR можно загружать только селекторы таких дескрипторов.

Виртуальная память со страничной организацией в процессорах x86

В защищенном режиме наряду с обязательной сегментацией процессоры x86 могут поддерживать виртуальную память со страничной организацией, которая реализует еще один уровень косвенности в формировании физического адреса. Принцип виртуальной памяти был уже рассмотрен. Здесь рассмотрим как реализуется страничное преобразование в x86.

Как линейное, так и физическое адресное пространство x86 делится на 1 Мб страниц по 4 Кб. Границы сегментов и границы страниц не зависят друг от друга и не обязаны быть выровнены.

В процессе страничного преобразования старшие 20 бит 32-разрядного линейного адреса через страничную таблицу замещаются номером физической страницы. Младшие 12 бит остаются неизменными (рис. 5.13).



Номер физ. страницы		Физический адрес

Рис. 4.12

Таблица страниц в 1 М элементов будет слишком большой, особенно с учетом того, что для каждой задачи нужна своя таблица. Поэтому в x86 на самом деле реализовано более гибкое двухбитное преобразование, показанное на рис. 5.14.

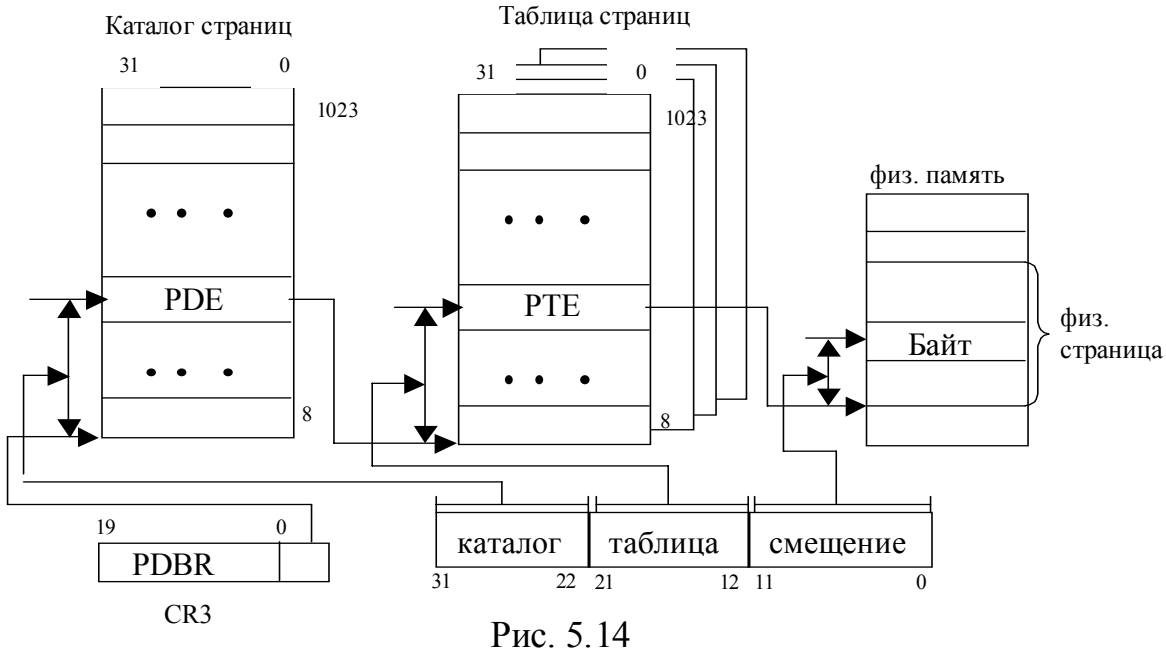


Рис. 5.14

Основой страничного преобразования выступает регистр управления CR3, который содержит 20-битный физический базовый адрес каталога страниц текущей задачи (регистр PDBR). Отметим, что это единственный внутренний регистр процессора, который содержит физический адрес памяти. Младшие 12 бит адреса считаются нулевыми, т.е. каталог выровнен по границе страниц. Предполагается, что каталог страниц постоянно находится в ОП.

Корневая таблица называется таблицей страниц первого уровня или просто каталог страниц, содержит 1024 32-разрядных дескриптора, называемых элементами каталога страниц PDE. Каждый из них адресует подчиненную таблицу страниц (таблицу страниц второго уровня). Каждая подчиненная таблица содержит1024 32-разрядных дескриптора, называемых элементами таблицы страниц (PTE), каждый из которых адресует страницу в адресной памяти.

Преобразование линейного адреса в физический состоит из следующих действий:

- старшие 10 бит (31–22) линейного адреса служат индексом в каталоге страниц, выбирая один из 1024 элементов PDE, который выбирает таблицу страниц;

- средние 10 бит (21–12) индексируют таблицу страниц, выбирая из нее элемент PTE, который содержит 20–разрядный базовый физический адрес в памяти;
- базовый адрес из PTE объединяется с 12 младшими битами линейного адреса, после чего получается физический адрес.

Формат элемента таблицы страниц. Элементы таблиц страниц обоих уровней т.е. элементы PDE и PTE имеют одинаковый формат, представленный на рис. 5.15.

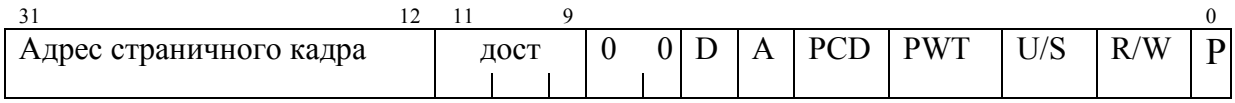


Рис. 5.15

Адрес страничного кадра – *в этом поле находится физический базовый адрес страницы (младшие 12 бит –0).*

Биты системного программиста. Процессор никогда не использует биты 11, 10, 9. Программисты могут использовать их по своему усмотрению.

Бит присутствия P. Показывает, находится ли страница в физической памяти (P=1). Если P=0, то страницы в памяти нет и остальная часть элемента таблицы страниц доступна для ОС, например для информации о местонахождении отсутствующей страницы. При попытке использования такой страницы возникает особый случай страничного нарушения. При этом ОС должна подгрузить страницу в физическую ОП из ВЗУ.

Биты *обращения A* и *«грязный» D* содержат информацию об использовании страницы. Бит A сообщает об любом обращении к странице, бит D – об обращении к странице для записи. Эти биты аппаратно устанавливаются процессором. ОС по ним может следить за частотой использования страниц памяти.

Биты R/W счет/запись, *U/S* – супервизор/пользователь используются механизмом защиты.

Биты PCD – управление кэшированием и *PWT* сквозной записи употребляются при кэшировании страниц и их рассматривать не будем.

Страничное преобразование включается установкой в 1 старшего бита PG в регистре CR0 с помощью команды MOV CR0, src. Со следующей команды начинается страничное преобразование. При этом требуется соблюдать осторожность, например линейные адреса этой программы должны совпадать с физическими адресами и после включения страничного преобразователя. Также должен быть предусмотрен ряд других мер, например, запрещены прерывания.

Защита по привилегиям в архитектуре x86

Практически во всех ЭВМ для целей защиты предусматриваются как минимум два режима работы – системный режим, называемый также режимом супервизора (Superevisor) и пользовательский режим (User). Основное различие между ними состоит в том, что программам, работающим

в режиме супервизора (ОС) доступны все ресурсы системы. В пользовательском режиме программам запрещается выполнение некоторых команд, влияющих на общесистемные ресурсы (т.н. привилегированные команды).

В архитектуре x86 этот принцип реализуется через поддержку 4-х уровней привилегий – PL. Механизм защиты x86 опирается на описание различных системных объектов с помощью дескрипторов. В каждом дескрипторе имеется двухбитное поле уровня привилегий DPL, которое определяет, каким программам разрешен доступ к описываемому им объекту.

Уровни привилегий

Средства защиты должны предотвращать неразрешенные взаимодействия пользователей друг с другом, несанкционированный доступ пользователей к данным, повреждения программ и данных из-за ошибок в программах, намеренные попытки разрушить целостность системы и случайные искажения данных. Механизм защиты процессоров x86 делится на две части: управление памятью (уже рассмотрели) и защиты по привилегиям.

Термин «привилегия» подразумевает права и возможности, которые обычно не разрешаются. Процессоры x86 поддерживают 4 уровня привилегий: 0, 1, 2, 3. Чем меньше номер уровня, тем он больше привилегирован. Уровни привилегий обычно изображаются в виде т.н. колец защиты (рис. 5.16). При выполнении почти каждой машинной команды осуществляется проверка защиты по привилегиям. Процессор в защищенном режиме постоянно контролирует, что текущая программа достаточно привилегированна, чтобы

- выполнять некоторые команды;
- обращаться к данным других программ;
- передавать управление внешнему (по отношению к самой программе) коду командами передачи управления типа FAR.

С каждым сегментом кода, данных или стека ассоциируется уровень привилегий и все, что находится внутри этого сегмента имеет этот уровень привилегий. Уровень привилегий выполняющегося в данный момент сегмента кода называется текущим уровнем привилегий (CPL) и он задается полем RPL селектора в регистре CS. При передаче управления сегменту кода с другим уровнем привилегий процессор будет работать на новом уровне привилегий. Часто говорят, например, «программа выполняется в кольце 0», подразумевая, что CPL процессора равен 0.

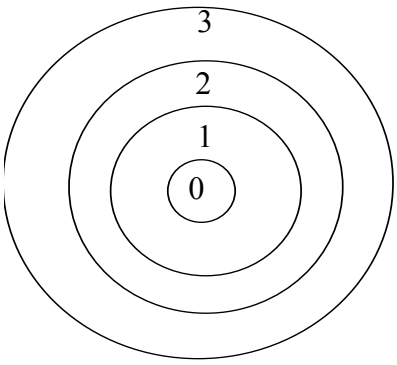


Рис. 5.16

Правила доступа по привилегиям иллюстрируется рис. 5.17 Видно, что программам не

разрешается доступ к данным, которые имеют более высокий уровень привилегий. Передача управления (FAR JMP, FAR CALL) не допускается за пределы своего поля защиты.

Рассмотрим подробнее правила защиты.

Привилегированные команды. Команды, воздействующие на механизм сегментации и защиты могут выполняться только в нулевом кольце (PL0-программы). К ним, в частности относятся HLT, LGDT, LIDT, LLDT, команды MOV с регистрами управления CRn, отладки DRn, проверки TRn и некоторые другие.

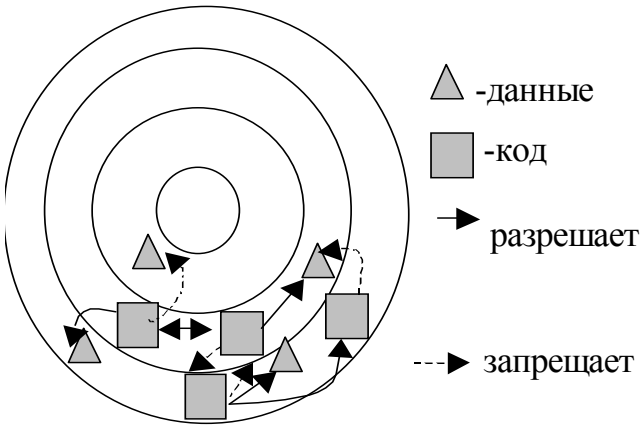


Рис. 5.17

Вторую группу образуют команды, которые изменяют состояние флажка прерываний IF и проводят ввод–вывод:

CLI	(IF=0)	IN	INS
STI	(IF+1)	OUT	OUTS

Для выполнения этих команд программа не обязательно должна иметь уровень привилегий 0. Однако их могут выполнять только те программы, уровень привилегий которых не ниже, определенного полем IOPL (2 бит) в регистре EFLAGS. Т.е. для выполнения этих команд требуется, чтобы CPL ≤ IOPL. Если IOPL = 3, то эти команды доступны всем программам.

Поскольку поле IOPL находится в доступном регистре EFLAGS, может показаться, что эту защиту можно преодолеть, например, рассмотрим код:

```
pushfd ; флажки в стек
or dwodprt ss:[esp], 3000h      ; iopl = 3
popfd  ; теперь ввод/вывод доступен.
```

Однако этот прием не срабатывает. Команда popfd сама по себе не является привилегированной, но их действие зависит от значения CPL. Команды popf и popfd могут изменить поле IOPL только если CPL = 0, т.е. к нулевом кольце. Если это не PL0-программа, процессор просто не модифицирует биты поля IOPL.

Примерно такая же ситуация характерна для флажка прерываний IF. Чтобы команды POPF или POPFD изменили состояние флажка IF, значение CPL должно быть меньше или равно значению IOPL.

Защита доступа к данным. Процессор x86 контролирует по привилегиям также обращение к данным. Процессор не разрешает

обращаться к данным, которые более привилегированны, чем выполняемая программа. Основное правило защиты имеет вид:

$$CPL \le DPL.$$

Когда программа пытается осуществить обращение с нарушением этого правила, процессор отказывается произвести его и сообщает о нарушении общей защиты.

Контроль реализуется двумя способами.

1. Контроль осуществляется при загрузке селектора в один из регистров DS, ES, FS, GS.
2. После успешной загрузки селектора при использовании его для фактического обращения к памяти процессор контролировать, чтобы запрашиваемая операция была разрешена.

При загрузке селектора в сегментный регистр стека правила защиты ужесточаются, требуется, чтобы CPL = DPL, т.е. не разрешается использовать стек даже с меньшими привилегиями.

Поле RPL. Младшие два бита селектора сегмента содержат поле запрашиваемого уровня привилегий RPL. Это поле не влияет на выбор дескриптора, но учитывается при контроле привилегий.

Защита по значению DPL (CPL ≤ DPL) позволяет изолировать код и данные на различных уровнях привилегий. Однако возможно, что ошибочный указатель, т.е. селектор, переданный более привилегированной программе, приводит к недопустимой модификации привилегированных данных.

Поле RPL предназначено для того, чтобы показывать уровень привилегий источника селектора. Селектор может передаваться через несколько процедур на различных уровнях. Содержимое RPL должно быть меньше или равно DPL. Возможность доступа к сегменту имеет вид:

$$\max(CPL,RPL) \le DPL.$$

Например, привилегированная процедура ОС, получив указатель (селектор) от пользовательской программы может принудительно установить в нем RPL = 3, предотвратив тем самым в принципе попытки доступа к привилегированным сегментам.

Защита сегментов кода. Напомним, что x86 запрещает передачу управления сегменту кода, находящемуся на другом уровне привилегий, тем самым предотвращая произвольное изменение уровня привилегий. Если бы CPL можно было бы легко изменять, все остальные средства защиты оставались бы бессмысленными.

Передачи управления в другой сегмент осуществляют команды FAR JMP, FAR CALL и FAR RET, поскольку при их выполнении меняются регистры CS и EIP.

При загрузке CS процессор предпринимает следующие проверки:

- проверяется, что новый дескриптор является сегментом кода;
- проверяется DPL = CPL, и бит присутствия P.

Если все проверки прошли, то дескриптор используется.

Передача управления между уровнями привилегий

Теперь возникает вопрос о том, как же осуществить передачу управления между уровнями привилегий? Необходимость такой передачи в первую очередь связана с доступом пользовательских программ к процедурам ОС, работающим на более высоком уровне привилегий. Имеется два способа решения этой проблемы: подчиненные сегменты кода и специальные дескрипторы, называемые шлюзами вызова.

Подчиненные сегменты кода. Представим себе, что в системе имеется процедура для преобразования целых чисел в строку символов. Эта процедура должна находится в кольце 0 (чтобы ее можно было вызвать из этого кольца). Обычно такая процедура требует достаточных привилегий только для обращения к своему параметру (целому числу) и возвращению результата, т.е. таких же привилегий, что и вызывающая программа. Именно для таких случаев в x86 предусмотрены подчиненные сегменты и код.

Напомним, что в байте AR дескриптора сегмента кода есть бит C – подчинения. Если C = 1, то это подчиненный сегмент кода и защита по CPL и DPL не действует.

С подчиненными сегментами кода не ассоциируется конкретный уровень привилегий, т.к. они подчиняются уровню привилегий того кода, который передает или управление с помощью команд CALL или JMP. Если, например, подчиненный сегмент кода вызывает программа из кольца 3, то он работает с CPL = 3, если из кольца 0, то с CPL = 0.

Применительно к подчиненному сегменту кода действует одно ограничение. Оно заключается в том, что значение DPL дескриптора подчиненного сегмента кода всегда должно быть меньше или равно текущему значению CPL. Другими словами, передача управления разрешается только во внутренне более защищенные кольца.

Шлюзы вызова. Для реализации фактического изменения уровней привилегий привлекаются особые системные объекты, называемые шлюзами вызова. Напомним, что x86 имеет дескриптор для системных объектов (например, есть шлюзы задач, шлюзы прерываний).

Формат шлюза вызова приведен на рис. 5.18.

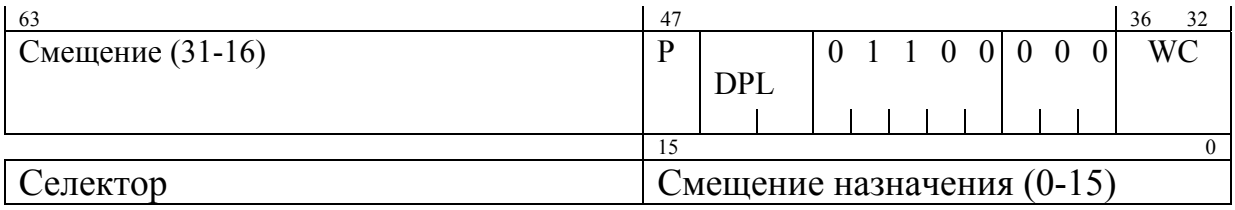


Рис. 5.18

48 разрядов в дескрипторе шлюза вызова определяют полный указатель селектор: смещение точки входа той процедуры (назначения), которой шлюз вызова передает управление. Дескриптор шлюза вызова действует как посредник между сегментами кода, находящимися на различных уровнях

привилегий. Шлюзы вызова идентифицируют разрешенные точки в более привилегированном коде, которым может быть передано управление, и являются единственным средством смены уровня привилегий.

WC – поле счетчика дескриптора шлюза (Word Count)

Дескрипторы шлюзов вызова не определяют никакого адреса пространства, поэтому в них нет полей базы и предела. Селекторы шлюзов вызова можно загружать только в сегментный регистр CS. Более того, адресовать шлюз вызова можно только с помощью команды межсегментного вызова FAR CALL. Сама команда CALL должна адресовать шлюз вызова, а не сегмент кода подключения (рис. 5.19).

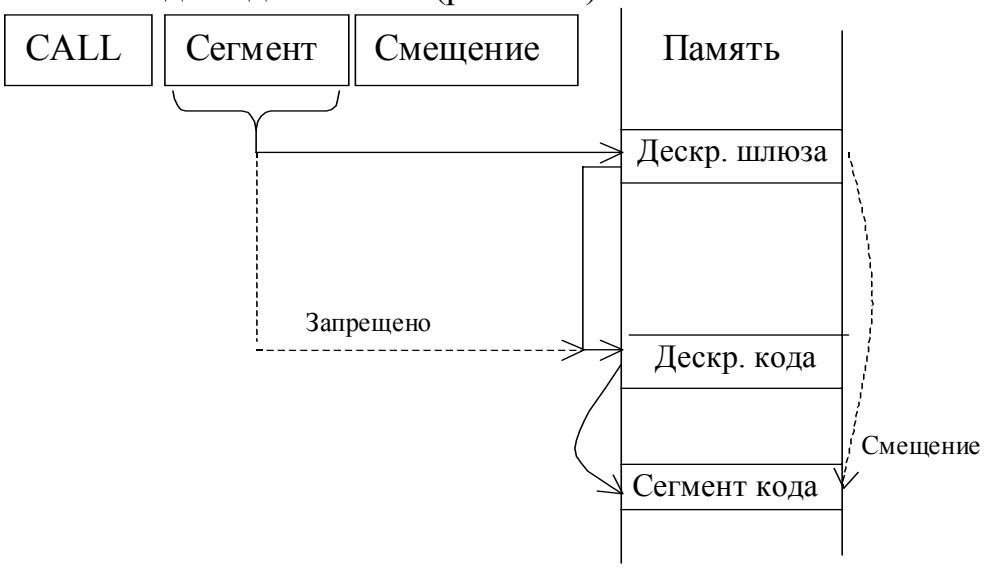


Рис. 5.19

По существу, в команде FAR CALL, которая обращается к шлюзу вызова, имеющееся смещение игнорируется, а селектор определяет только шлюз вызова.

Реализованный в x86 такой косвенный вызов привилегированных процедур имеет несколько преимуществ:

1. Привилегированный код сильно защищен и вызывающие программы не могут его разрушить. Разумеется, предполагаем, что сам этот код тщательно отлажен, не содержит ошибок и не может привести к катастрофе.
2. Шлюзы вызова делают код процедуры невидимым для программ на внешних уровнях привилегий.
3. Так как вызывающая программа прямо адресует только шлюз, реализуемые процедурой функции можно изменять или перемещать в адресном пространстве, не затрагивая интерфейс со шлюзом.

Доступность шлюза вызова. Шлюзы вызова имеют определенное ограничение на использование. При этом учитывается

- значение DPL самого шлюза вызова;
- значение DPL дескриптора вызываемого (целевого) сегмента кода;
- значение RPL селектора в команде FAR CALL;
- значение CPL текущей (вызывающей) программы.

Правило разрешения вызова через шлюз принимает вид:

DPL целевого сегмента ≤ max(RPL,CPL) ≤ DPL шлюза.

Таким образом, сами шлюзы должны иметь уровень привилегий не выше текущего кода, а целевой сегмент – не ниже текущего кода.

Например, если процессор выполняет PL2 – программу (CPL=2), и ей требуется вызвать PL0-процедуру, (целевой DPL=0), необходимо использовать дескриптор шлюза со значением DPL равным 2 или 3.

Суммарные правила для использования шлюза вызова имеют следующий вид:

- значение DPL шлюза вызова должно быть больше или равно значению текущего уровня привилегий CPL;
- значение DPL шлюза вызова должно быть больше или равно значению поля RPL селектора шлюза;
- значение DPL шлюза вызова должно быть больше или равно значению DPL целевого сегмента кода;
- значение DPL целевого сегмента кода должно быть меньше или равно значению текущего уровня привилегий CPL;

Переключение стека. Когда происходит изменение уровня привилегий, возникает еще одна проблема, связанная со стеком. Одно из правил защиты по привилегиям требует, чтобы уровень привилегий стека всегда был равен уровню CPL. Чтобы не нарушать это правило процессор x86 при смене уровня привилегий автоматически переключает и стек для соответствия новому, более привилегированному коду. При возврате управления исходному коду возобновляется использование старого стека. Процесс переключения стека для программ невидим.

Необходимость переключения стека диктуется двумя основными причинами:

- вызываемая процедура должна защищаться от возможного переполнения стека, возникающего в том случае, если вызывающая программа распределила недостаточное стековое пространство;
- сегмент стека, используемый более привилегированной процедурой, может быть разрушен менее привилегированными программами, разделяющими стек с программой, осуществляющей вызов через шлюз.

Необходимость переключения стека ставит вопрос о том, где взять указатель для регистров SS:ESP нового стека. Для этого придется привлечь еще один специальный сегментный дескриптор, который определяет сегмент состояния задачи TSS, ассоциируемый с каждой задачей (программой). Фрагмент сегмента состояния приведен на рис. 4.5.

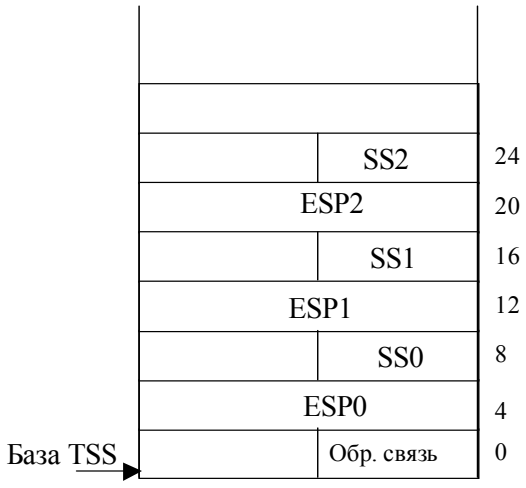


Рис. 5.20

В сегменте TSS находятся селекторы сегмента и указатели стека для

уровней привилегий 0, 1 и 2. Пара SS:ESP для уровня 3 не нужны, т.к. не существует вызова, в котором вызывающая программа менее привилегированна, чем вызвавшая в кольце 3.

После успешного прохождения командой CALL шлюза вызова без нарушений правил защиты, процессор должен начать работать с новым стеком.

Для этого в регистры SS:ESP из сегмента TSS загружаются селектор сегмента и указатель стека, соответствующий новому уровню привилегий. Например, если вызвана PL1 – процедура, в регистры SS:ESP загружаются значения SS1 и ESP1.

После загрузки регистров SS:ESP для адресации нового стека процессор сразу же включает в новый стек старые селектор сегмента и указатель стека, которые адресуют вершину старого стека. Затем процессор анализирует поле счетчика WC дескриптора шлюза вызова и автоматически копирует из старого стека в новый указанное число двойных слов. Они представляют собой параметры, передаваемые вызываемой процедуре. Эти параметры размещаются в новом стеке так же, как и в старом. Наконец, в новый стек включается адрес возврата CS:EIP. Таким образом, после производства этих действий новый стек выглядит точно так же, как будто во внутреннем кольце произведено включение в стек параметров и выполнена команда FAR CALL (рис. 5.21).

Начальные значения SS:ESP для трех уровней привилегий в TSS только считываются и никогда не изменяются при работе процессора.

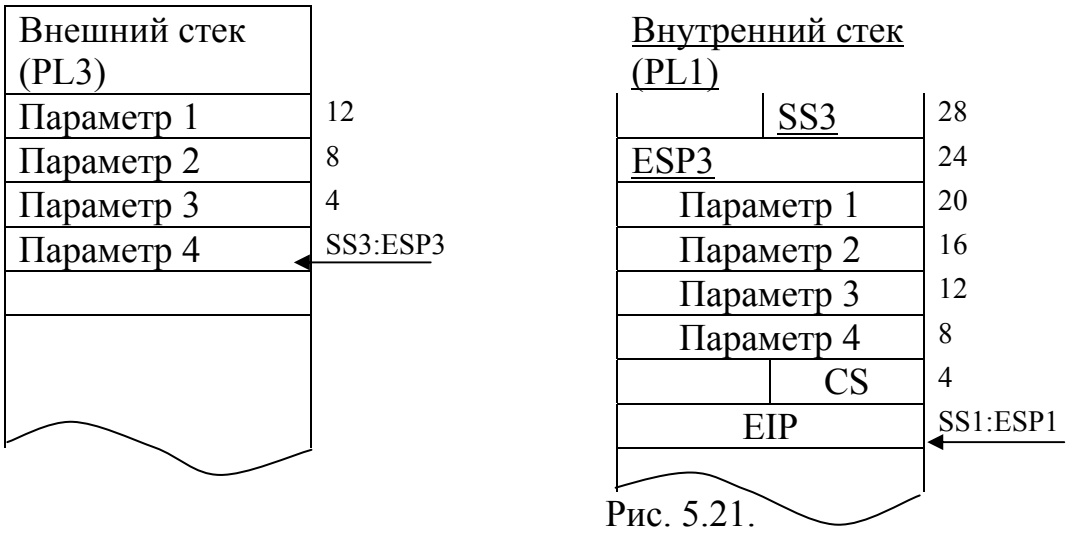


Рис. 5.21.

Для инициирования выполнения вызванной процедуры остается только загрузить значения селектора и смещения из дескриптора шлюза вызова в регистры CS:EIP.

При переключении стека процессор осуществляет следующие защитные проверки:

- контролирует задание правильного указателя стека для более привилегированного уровня;
- проверяет нахождение сегмента стека на правильном уровне привилегий (например, селектор SS0 должен адресовать сегмент на уровне 0 и иметь поле RPL=0);
- убеждается в том, что в новом стеке достаточно места для старого указателя стека (8 байт), копируемых параметров (0-128 байт), адреса возврата (8 байт) и тех локальных переменных, которые может включать в стек вызываемая процедура;

Привилегированная процедура осуществляет возврат при помощи команд FAR RET, FAR RETn. При этом происходит обратное переключение стека и удаление параметров из обоих стеков.

Виртуальная память UltraSPARC II

UltraSparc II – это 64-разрядная машина, которая поддерживает виртуальную память со страничной организацией и с 64-битными виртуальными адресами. Однако, по ряду причин программы не могут использовать полное 64-битное виртуальное адресное пространство. Допустимая виртуальная память делится на две зоны по 2^{43} байтов каждая, одна из которых находится в верхней части виртуального адресного пространства, а другая – в нижней. Между ними находится «дырка», содержащая адреса, которые не используются.

Максимальная физическая память компьютера UltraSPARC II составляет 2^{41} байт (2200 Гбайт). Поддерживается четыре размера страниц: 8, 64, 512 Кбайт и 4 Мбайта. Отображение этих страниц приведено на рис. 5.22.

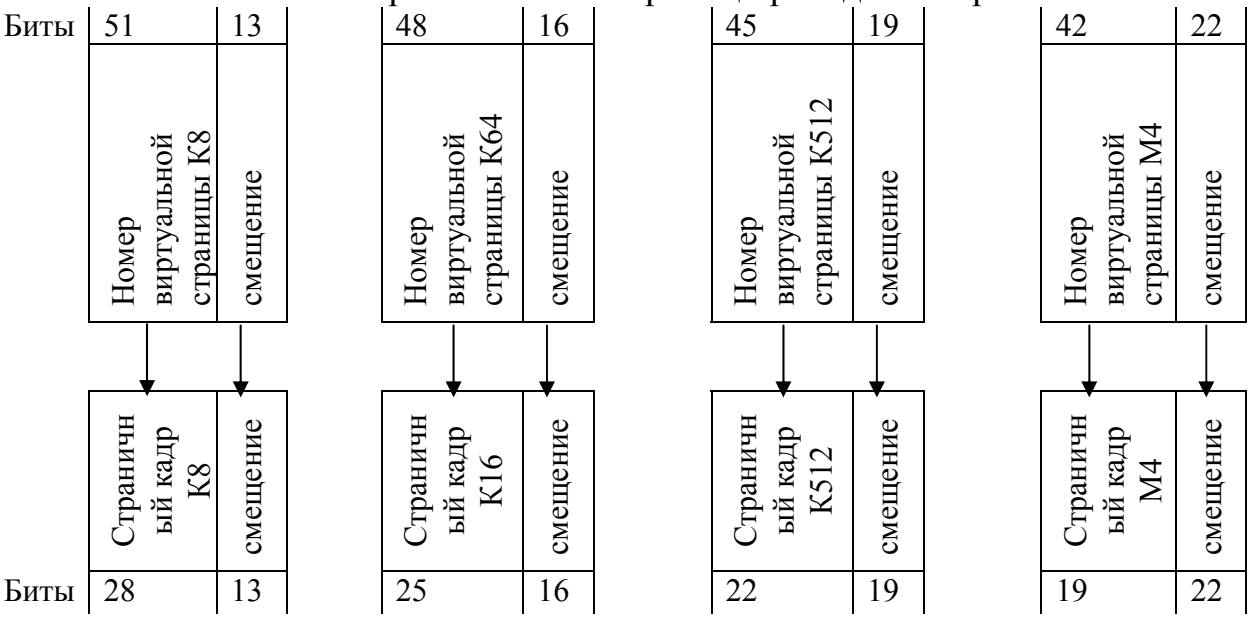


Рис. 5.22

Из-за огромного виртуального адресного пространства обычная таблица страниц (как в Pentium II) не будет практичной. Здесь применяется

следующий подход. Устройство управления памятью содержит таблицу *TLB (Translation Lookaside Buffer – буфер быстрого преобразования адреса)*. Эта таблица отображает номера виртуальных страниц в номера физических страниц кадров. Для страниц размером в 8 К существует 2^{31} номеров виртуальных страниц, которые все не могут быть отображены.

Поэтому TLB содержит только номера самых последних используемых виртуальных страниц. Страницы команд и страницы данных рассматриваются отдельно. Для каждой из этих категорий в TLB включены номера 64 последних используемых виртуальных страниц. Каждый элемент буфера включает номер виртуальной страницы и соответствующий номер физического страничного кадра. Когда номер процесса вызывает его *контекст*, виртуальный адрес в этом контексте передается в контроллер управления памятью, то он с помощью специальной схемы сравнивает номер виртуальной страницы со всеми элементами буфера быстрого преобразования адреса TLB для данного контекста одновременно. Если обнаружится совпадение, номер страничного кадра в этом элементе буфера соединяется со смещением, взятым из виртуального адреса, что бы получить 41-битный физический адрес и обработать некоторые флаги. Буфер быстрого преобразования адреса изображен на рис. 5.23 а. Если совпадение не обнаружено происходит *промах в TLB,* который вызывает ловушку в операционной системе. Данный промах отличается от ошибки из-за отсутствия страницы. Промах буфера TLB может произойти, даже если нужная страница присутствует в памяти. Теоретически операционная система может сама загрузить новый элемент этого буфера для нужной виртуальной страницы. Однако для ускорения данной операции к этой работе подключается аппаратное обеспечение, если программное обеспечение взаимодействует с ним.

Рис. 5.23 а

Операционная система должна сохранять наиболее часто используемые элементы буфера TLB в таблице под названием буфер хранения преобразований (*TSB – translation storage buffer*). Эта таблица простроена как кэш-память прямого отображения виртуальных страниц. Каждый 16-байтный элемент данной таблицы содержит бит достоверности, номер контекста, туг виртуального адресного пространства, номер физической страницы и несколько флаговых битов. Если размер кэш-памяти составляет, допустим, 8192 элемента, тогда все виртуальные страницы, у которых младшие 13 битов отображаются в 0000000000000, будут претендовать на элемент 0 в данной таблице (рис. 5.23 б). Размер таблицы определяется программным обеспечением и передается в контроллер управления памятью через специальные регистры, доступные только для операционной системы.

При промахе буфера преобразований операционная система проверяет, содержит ли соответствующий элемент буфера TLB нужную виртуальную страницу. Контроллер управления памятью вычисляет адрес этого элемента и помещает его в свой внутренний регистр, доступный для операционной системы. Если нужный элемент есть в таблице хранения преобразований, то какой-нибудь элемент удаляется из буфера TLB, а соответствующий элемент буфера хранения преобразований копируется туда. Аппаратное обеспечение с помощью алгоритма LRU выбирает, какой именно элемент нужно выкинуть.

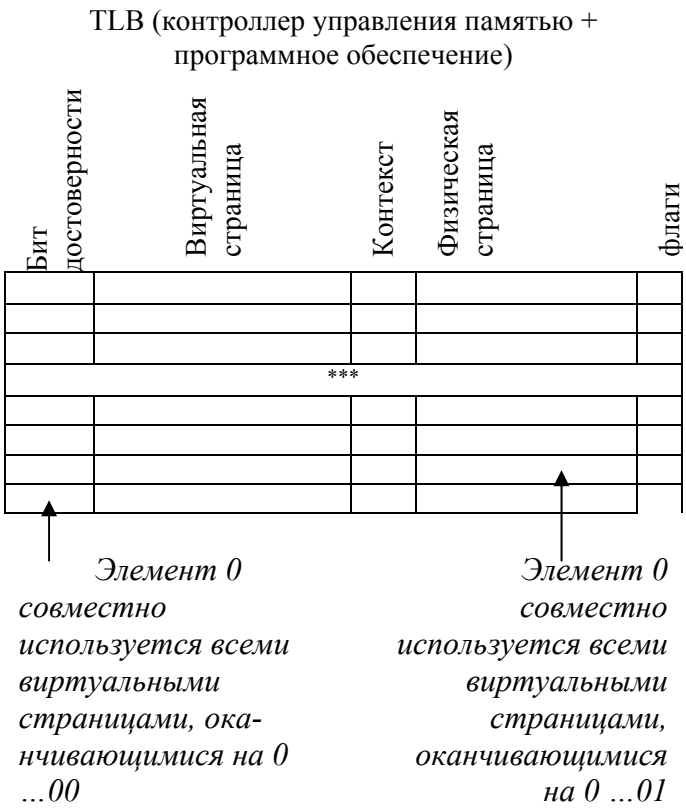


Рис. 5.23 б

Если нужной виртуальной страницы нет в кэш-памяти, операционная система использует другую таблицу для нахождения информации о странице, которая может находится или не находится в основной памяти. Таблица, которая применяется для этого процесса, называется *транслирующей таблицей*. Поскольку здесь аппаратное обеспечение не участвует в поиске элементов, операционная система может использовать любой формат. Если поиск страницы в таблице трансляции привел к нахождению нужной страницы в памяти, то элемент TSB в кэш-памяти обновляется. Если в результате поиска обнаружилось, что нужной таблицы нет в памяти, то происходит стандартная ошибка.

Сравним схемы разбиения на страницы в Pentium II и UltraSPARC II. Pentium II поддерживает чистую сегментацию, чистое разбиение на страницы и сегментацию в сочетании с разбиением на страницы. UltraSPARC II поддерживает только разбиение на страницы. Pentium II использует аппаратное обеспечение для перезагрузки буфера TLB только в случае промаха TLB. UltraSPARC II в случае такого промаха просто передает управление операционной системе.

Причина этого различия состоит в том, что Pentium II использует 32-разрядные сегменты, а такие маленькие сегменты (только 1 млн страниц) могут обрабатываться только с помощью страничных таблиц. Теоретически у Pentium II могли бы возникнуть проблемы, если бы программа использовала тысячи сегментов, но т.к. ни в одной версии Windows или UNIX не поддерживается более одного сегмента на процесс, никаких проблем не возникает. UltraSPARC II — 64-битная машина. Она может содержать до 2 млрд страниц, поэтому таблицы страниц не работают. В будущем все

машины будут иметь 64-битные виртуальные адресные пространства, и схема UltraSPARC II станет нормой.

Виртуальная память и кэширование

На первый взгляд виртуальная память и кэширование никак не связаны, но на самом деле они сходны. При наличии виртуальной памяти вся программа хранится на диске и разбивается на страницы фиксированного размера. Некоторое подмножество этих страниц находится в основной памяти. Если программа главным образом использует страницы из основной памяти, то ошибки из-за отсутствия страницы будут встречаться редко и программа будет работать быстро. При кэшировании вся программа хранится в основной памяти и разбивается на блоки фиксированного размера. Некоторое подмножество этих блоков находится в кэш-памяти. Если программа главным образом использует блоки из кэш-памяти, то промахи будут происходить редко и программа будет работать быстро. Как видно, виртуальная и кэш-память идентичны, только работают на разных уровнях иерархии.

Различия в виртуальной и кэш-памяти. Промахи кэш-памяти обрабатываются аппаратным обеспечением, а ошибки из-за отсутствия страниц обрабатываются операционной системой. Блоки кэш-памяти обычно гораздо меньше страниц (64 байта или 8 Кбайт). Кроме того, таблицы страниц индексируются по старшим битам виртуального адреса, а кэш-память индексируется по младшим битам адреса памяти. Однако, по существу, различие существует только в реализации.

5.2. ВИРТУАЛЬНЫЕ КОМАНДЫ ВВОДА-ВЫВОДА

Набор команд уровня архитектуры команд полностью отличается от набора команд микроархитектурного уровня.

Набор команд уровня операционной системы содержит большую часть команд из уровня архитектуры команд, а также несколько новых, очень важных команд. Ввод-вывод – это одна из областей в которых два этих уровня различаются очень сильно. Причины различия.

- Пользователь, способный выполнить команды ввода-вывода уровня архитектуры команд, может считать конфиденциальную информацию, которая хранится в системе и может представлять угрозу системе;
- Обычные нормальные программисты не хотят осуществлять ввод-вывод на уровне команд, поскольку это утомительно.

Вместо этого для осуществления ввода-вывода нужно установить определенные поля и биты в ряде регистров устройств, затем подождать,

пока операция закончится и проверить, что произошло. Диски обычно содержат биты регистров устройств для обнаружения следующих ошибок:

1. Аппаратура диска не смогла выполнить позиционирование.
2. Несуществующий элемент памяти определен как буфер.
3. процесс ввода-вывода на диск (с диска) начался до ого, как закончился предыдущий.
4. Ошибка синхронизации при считывании.
5. Обращение к несуществующему диску.
6. Обращение к несуществующему цилиндру.
7. Обращение к несуществующему сектору.
8. Ошибка проверки записи после операции записи.

При наличии одной из этих ошибок устанавливается соответсвующий бит в регистре устройств.

Файлы

Один из способов организации виртуального ввода-вывода – использование абстракции под названием *файл*. Файл состоит из последовательности байтов, записанных на устройство ввода-вывода. Если устройство является устройством хранения информации (диск), то файл можно считать обратно. Эта абстракция позволяет легко организовать виртуальный ввод-вывод.

Для операционной системы файл является последовательностью байтов. Ввод-вывод файлов осуществляется с помощью системных вызовов для открытия, чтения, записи и закрытия файлов. Процесс открытия файлов позволяет операционной системе найти файл на диске и передать в память информацию, необходимую для доступа к этому файлу.

После открытия файла его можно считывать. Системный вызов для считывания должен иметь как минимум следующие параметры:

- Указание, какой именно открытый файл нужно считывать;
- Указатель на буфер в памяти, в который нужно поместить данные;
- Число считываемых байтов.

Данный системный вызов помещает требуемые данные в буфер. Обычно он возвращает число считанных байтов. Это число может быть меньше запрашиваемого.

С каждым открытым файлов связан указатель, который сообщает, какой бит будет считываться следующим.

После команды ***read*** указатель дополняется число считанных байтов, поэтому последовательные команды ***read*** считывают последовательные блоки данных из файла. Обычно этот указатель можно установить на особое значение, чтобы программы могли получать доступ к любой части файла. Когда программа закончила считывание файла, она может закрыть его и сообщить операционной системе, что она больше не будет использовать этот файл. Операционная система может освободить пространство в таблице, в которой хранилась информация об этом файле.

В операционных системах универсальных вычислительных машин файл представляет собой более сложную структуру. Здесь файл может быть последовательностью логических записей, каждая из которых имеет строго определенную структуру. Некоторые операционные системы различают файлы, у которых все элементы имеют одинаковую структуру, и файлы, содержащие разные типы данных.

Основная виртуальная команда ввода считывает следующую запись из нужного файла и помещает ее в основную память, начиная с определенного адреса (рис. 5.24). Чтобы выполнить эту операцию, виртуальная команда должна получить сведения о том, какой файл считывать и купа поместить запись. Часто существуют параметры для чтения некоторой записи, которые определяются или по ее месту в файле, или по ее ключу.

Рис. 5.24

Основная виртуальная команда вывода записывает логическую запись из памяти в файл. Последовательные команды ***write*** производят последовательные логические записи в файл.

Реализация виртуальных команд ввода-вывода

Основной вопрос в проблеме организации виртуальных команд ввода-вывода – распределение памяти. Единичным блоком может быть один сектор на диске, но чаще он состоит из последовательных секторов.

Еще одно фундаментальное свойство реализации системы файлов – хранится ли файл в последовательных блоках или нет. Восприятие файлов прикладным программистом сильно отличается от восприятия файла операционной системой. Программист воспринимает файл как линейную последовательность байтов или логических записей. Операционная система

воспринимает файл как упорядоченную, но необязательно последовательную, совокупность единичных блоков на диске.

Для того, чтобы операционная система могла доставить байт или логическую запись n из какого-то файла, она должна пользоваться некоторым методом для определения местонахождения данных. Если файл расположен последовательно, то операционная система должна знать только место начала файла, чтобы вычислить позицию нужной записи.

Если файл расположен на диске непоследовательно, то только по начальной позиции невозможно определить местоположение требуемой записи. Чтобы найти произвольный байт или логическую запись, нужна таблица (так называемый *индекс файла*), которая выдает единичные блоки и их физические адреса на диске. Индекс файла может быть организован либо в виде списка адресов блоков (UNIX), либо в виде списка логических записей, для каждой из которых дается адрес на диске и смещение. Иногда каждая логическая запись имеет *ключ*, и программы могут обращаться к записи по этому ключу, а не по номеру логической записи. В этом случае каждый элемент таблицы должен содержать не только информацию о местонахождении записи на диске, но и ее ключ. Такая структура обычно применяется в универсальных вычислительных машинах.

Альтернативный метод нахождения блоков файлов – организовать файл в виде связного списка. Каждый единичный блок содержит адрес следующего единичного блока. Для реализации этой схемы нужно в основной памяти иметь таблицу со всеми последующими адресами. Например, для диска с 64 К блоками операционная система должна иметь в памяти таблицу из 64 К элементов, в каждом из которых дается индекс следующего единичного блока. Так, если файл занимает блоки 4, 52, 19, то элемент 4 таблицы будет содержать число 52, элемент 52 – число 19, а 19 элемент – специальный код, указывающий на конец файла. Так работают файловые системы в MS DOS, Windows 95 b Windows 98. Windows NT поддерживает эту систему файлов, но и кроме того имеет свой собственную систему, которая больше похожа на файловую систему UNIX.

Сравнение двух типов расположения файлов.

Последовательно расположенными файлами легко управлять, но если максимальный размер файла не известен заранее, эту технологию нельзя использовать. Если файл располагается непоследовательно, то проблем с поиском связного пространства не возникает.

С другой стороны, если максимальный размер файла известен заранее, то программа может заранее определить серии секторов, точно соответствующих размерам различных файлов.

Чтобы распределить пространство на диске для файла, операционная система должна следить, какие блоки доступны, а какие уже заняты другими файлами. При записи на компакт-диск такие вычисления производятся один раз и навсегда, а на жестком диске файлы постоянно записываются и удаляются. Один из способов – сохранить список всех «дырок». Этот список называется *списком свободной памяти*.

Рассмотрим размер единичного блока. Здесь играют роль несколько факторов.

Время поиска и время, затрачиваемое на вращение диска, затормаживает доступ к диску. Поэтому, увеличение размеров блока позволяет снизить удельный вес непроизводственных затрат.

Чем меньше размер единичного блока, тем больше их должно быть, что влечет за собой длинные индексы файлов и большие таблицы в памяти.

Маленькие блоки имеют свои преимущества. В маленьких блоках меньше потери памяти за счет эффекта фрагментации. Однако, в последнее время самым важным фактором считается быстрота передачи данных, поэтому размер блоков постоянно увеличивается.

Команды управления директориями

Информация, доступная компьютеру без вмешательства человека называется *неавтономной*. *Автономная* информация требует вмешательства человека (вставить диск).

Неавтономная информация хранится в файлах. Обычно операционная система группирует неавтономные файлы в директории. Минимально необходимые системные вызовы для работы с файловой системой следующие:

1. создание файла и введение его в директорию;
2. стирание файла из директории;
3. переименование файла;
4. изменение статуса файла.

Применяются различные схемы защиты файлов. Например, пароль.

Виртуальные команды для параллельной обработки

Некоторые вычисления удобно проводить с помощью дух и более параллельных процессоров, т.е. как будто бы на разных процессорах. Чтобы несколько процессов могли проистекать параллельно, нужны специальные виртуальные команды. Повышение производительности компьютеров за счет развития технологии ограничено законами физики. Единственный путь, не входящий в противоречие с законами природы – создание параллельных вычислений.

В компьютере с несколькими процессорами каждый из нескольких взаимодействующих процессов можно приписать к одному определенному процессору, чтобы выполнять несколько действий одновременно. Если в компьютере имеется только один процессор, эффект параллельной обработки можно смоделировать. Для этого процессы будут выполняться по очереди, каждый понемножку. Иначе говоря, процессор будет разделяться между несколькими процессами. Но даже в том случае, когда параллельная обработка моделируется, удобно считать, что каждому процессу приписывается свой собственный процессор. При моделированной

параллельной обработке возникают те же проблемы, что и при реальной параллельной обработке.

Формирование процесса

Программа работает как часть какого-то процесса. Этот процесс, как и другие процессы характеризуется состоянием и адресным пространством, через которое можно получить доступ к программам и данным. Состояние включает как минимум счетчик команд, слово состояния программы, указатель стека и регистры общего назначения.

Большинство современных операционных систем позволяют формировать и прерывать процессы динамически. Для формирования нового процесса требуется системный вызов. Этот системный вызов может просто создать клон вызывающей программы или позволить исходному процессу определить начальное состояние нового процесса, т.е. его программу, данные и начальный адрес.

В одних случаях исходный процесс сохраняет частичный или полный контроль над порожденным процессом. Виртуальные команды позволяют исходному процессу останавливать и снова запускать процесс, проверять и завершать подчиненные процессы. В других случаях исходный процесс никак не контролирует подчиненный, так, что два процесса работают независимо друг от друга.

Состояние гонок

Во многих случаях параллельные процессы должны взаимодействовать и их работу нужно синхронизировать. Пусть существует два процесса 1 и 2, которые взаимодействуют через общий буфер основной памяти. Пусть процесс 1 – *производитель (producer),* а процесс 2 – *потребитель (consumer)*. Производитель формирует числа и помещает в память, а потребитель выводит их на печать.

Эти два процесса работают с различной скоростью, печать идет медленнее, чем производство чисел. Если производитель видит, что буфер заполнен, то он временно приостанавливает операцию, переходит в режим ожидания. Кодда потребитель взял число из буфера, он сообщает производителю, что можно работать дальше. Если потребитель видит, что буфер пуст, он приостанавливает работу.

При программной реализации рассмотренного случая две программы выполняют проверку состояния буфера и в зависимости от результатов проверки либо продолжают работы, либо приостанавливаются. Возможно ложное поведение процессов в случае параллельной обработки процессов. Допустим, процесс считал информацию о состоянии буфера. После этого начался квант времени обслуживания другого процесса, который изменил состояние буфера. Первый процесс принимает решение на основе информации, которая была взята ранее, до изменения состояния буфера, что может войти в конфликт с текущим его состоянием.

Такая ситуация называется *состоянием гонок*.

Синхронизация процесса с использованием семафоров

Проблему состояния гонок можно решить по крайней мере двумя способами. Первый способ – снабдить каждый процесс специальным битом ожидания пробуждения. Если процесс, который функционирует в данный момент, получает сигнал «пробуждения», то этот бит устанавливается. Если процесс отключается в тот момент, когда бит установлен, он немедленно перезапускается, а бит сбрасывается. Этот метод решает проблему состояния гонок только в том случае, если у нас всего два процесса. В общем случае, при наличии n процессов он не работает.

Дейкстра предложил другое решение этой проблемы. Где-то памяти находятся две переменные, которые могут содержать неотрицательные целые числа. Эти переменные называются *семафорами*. Операционная система предоставляет два системных вызова, ***in*** и ***down***, которые оперируют семафорами. ***Up*** прибавляет 1 к семафору, а ***down*** – отнимает 1. Если операция down совершается над семафором, значение которого больше 0, то этот семафор уменьшается на 1 и процесс продолжается. Если значение семафора равно 0, то операция ***down*** не может завершиться. Тогда процесс отключается до тех пор, пока какой-нибудь процесс не выполнит операцию ***up*** над этим семафором.

Команда ***up*** проверяет, не равен ли семафор нулю. Если он равен нулю и другой процесс находится в режиме ожидания, то семафор увеличивается на 1. После этого процесс, который «спит», может завершить операцию ***down***, установив семафор на 0. Теперь оба процесса продолжают работать. Если семафор не равен 0, команда ***up*** просто увеличивает его на 1. Семафор позволяет сохранить сигналы пробуждения, так что они не пропадут зря. У семафорных команд есть одно важное свойство: если один из процессов начал выполнять команду над семафором, то другой процесс не может получить доступ до тех пор, пока первый не завершит выполнение команды или не будет приостановлен при попытке выполнить команду down над 0.

Операции над семафорами неделимы. Если операция над семафором уже началась, то никакой другой процесс не может использовать этот семафор до тех пор, пока первый процесс не завершит операцию или не пока он не будет приостановлен. Более того, при наличии семафоров сигналы пробуждения не пропадают. В сущности, проблема была решена за счет введения неделимых системный прерываний ***up*** и ***down***. Чтобы эти операции были неделимы, операционная система должна запретить двум и более процессам использовать один семафор одновременно. Если был сделан системный вызов up или down, ни один другой код пользователя не будет запущен, пока данный вызов не завершится. Для этого обычно во время выполнения операций над семафорами вводится запрет на прерывания.

Технология с использованием семафоров работает для произвольного количества процессов. Несколько процессов могут «спать», не завершив

системный вызов down на одном и том же семафоре. Когда какой-нибудь другой процесс выполнит операцию up на том семафоре, один из ждущих процессов может завершить вызов down и продолжить работу. Семафор сохраняет значение 0 и другие процессы продолжают ждать.

6. АРХИТЕКТУРЫ КОМПЬЮТЕРОВ ПАРАЛЛЕЛЬНОГО ДЕЙСТВИЯ

6.1 ВОПРОСЫ РАЗРАБОТКИ КОМПЬЮТЕРОВ ПАРАЛЛЕЛЬНОГО ДЕЙСТВИЯ

Когда мы сталкиваемся с новой компьютерной системой параллельного действия, возникает три вопроса:

1. Каков тип, размер и количество процессорных элементов?
2. каков тип, размер и количество модулей памяти?
3. Как взаимодействуют элементы памяти и процессорные элементы?

Процессорные элементы могут быть самых различных типов – от минимальных АЛУ до полных центральных процессоров. Сейчас компьютеры параллельного действия конструируются из серийно выпускаемых частей. Разработка компьютеров параллельного действия часто зависит от того, какие функции выполняют эти части и каковы ограничения.

Системы памяти часто разделены на модули, которые работают независимо друг от друга, чтобы несколько процессоров могли осуществить доступ к памяти. Память работает гораздо медленнее процессоров, поэтому для повышения скорости доступа к памяти используются различные схемы кэш-памяти. Кэш-память может быть двух- трех- и даже четырехуровневой.

Системы параллельного действия различают в основном тем, как соединены разные части. Системы взаимодействия можно разделить на две категории – статические и динамические. В статических схемах компоненты просто связываются друг с другом определенным образом. В качестве примеров статических схем можно привести кольцо, звезду, решетку. В динамических схемах все компоненты подсоединены к переключательной схеме, которая может трассировать сообщения между компонентами.

Компьютеры параллельного действия можно рассматривать как набор микросхем, которые соединены между собой определенным образом. Это один из подходов. При другом подходе рассматривают процессы, выполняемые параллельно. И здесь рассматриваются различные варианты.

Некоторые компьютеры параллельного действия одновременно выполняют несколько независимых задач. Эти задачи никак не связаны между собой.

Другие компьютеры параллельного действия выполняют одну задачу, состоящую из нескольких параллельных процессов.

Далее идут машины с высокой степенью конвейеризации или с большим количеством АЛУ, которые обрабатывают одновременно один поток команд. Это компьютеры со специальным аппаратным обеспечением для обработки векторных данных.

Эти три примера различаются *степенью детализации*. Параллельная работа больших частей программного обеспечения без взаимодействия между этими частями – *параллелизм на уровне крупных структурных единиц.* Диаметрально противоположный случай (обработка векторных данных) – параллелизм на уровне мелких структурных единиц.

Термин «степень детализации» применяется по отношению к алгоритмам и программам, но у него есть прямой аналог в аппаратном обеспечении.

Системы с небольшим числом больших процессоров, которые взаимодействуют по схемам с низкой скоростью передачи данных, называют *системами с косвенной (слабой) связью*. Им противопоставляются *системы с непосредственной (тесной) связью*, в которых компоненты обычно меньше по размерам.

6.1.1.Информационные модели

В любой системе параллельной обработки процессоры, выполняющие разные части одной задачи, должны взаимодействовать друг с другом, чтобы обмениваться информацией. Были предложены и реализованы две разработки: мультипроцессоры и мультикомпьютеры.

Мультипроцессоры

В первом случае все процессоры разделяют общую физическую память как показано на рис. 6.1а. Такая система называется *мультипроцессором* или *системой с совместно используемой памятью*. Мультипроцессорная модель распространяется и на программное обеспечение. Все процессы, работающие вместе на мультипроцессоре, разделяют одно адресное пространство.

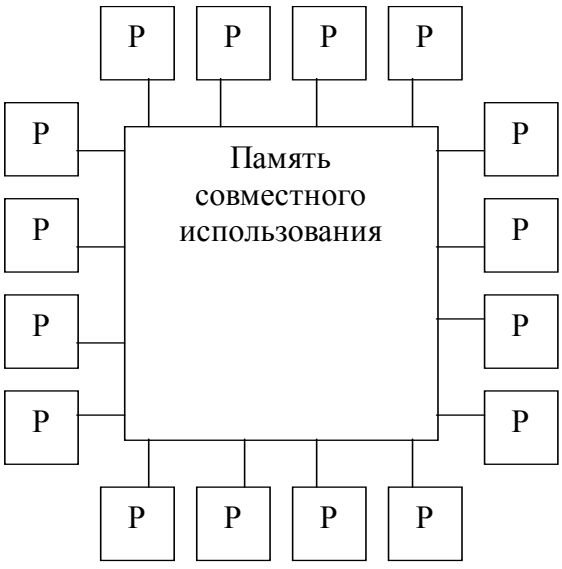


Рис. 6.1

В такой системе два процесса могут обмениваться информацией, если один из них будет записывать в память, а второй считывать. Благодаря такой возможности взаимодействия двух и более процессов мультипроцессорные системы очень популярны.

В качестве примеров мультипроцессоров можно назвать Sun Enterprise 10000, Sequent NUMA-Q, SGI Origin, HP/Convex Exemplar.

Мультикомпьютеры

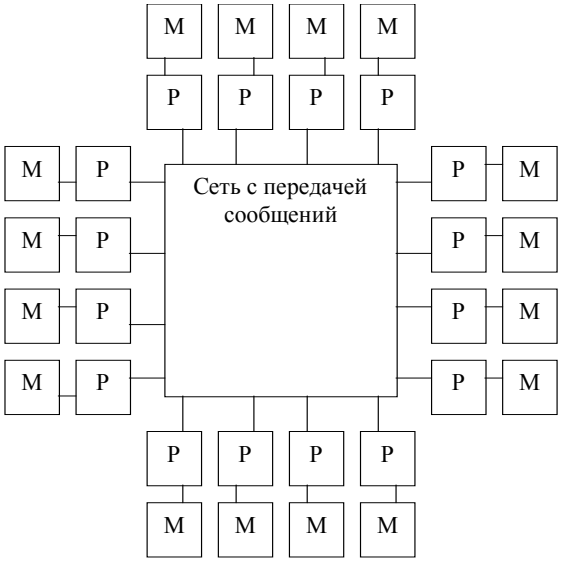


Рис. 6.2

Во втором типе параллельной архитектуры каждый компьютер имеет свою собственную память, доступную только этому процессору. Такая разработка называется *мультикомпьютером* или *системой с распределенной памятью*. Она изображена на рис. 6.2 а. Мультикомпьютеры в большинстве случаев являются системами со слабой связью. Ключевое отличие мультикомпьютера от мультипроцессора состоит в том, что каждый процессор в мультикомпьютере имеет свою собственную локальную память и никакой другой процесс не может получить доступ к этой памяти. Мультикомпьютеры содержат отдельное физическое адресное пространство для каждого центрального процессора.

Поскольку процессоры в мультикомпьютере не могут взаимодействовать путем совместного обращения в общую память, то здесь необходим другой механизм взаимодействия. Они посылают друг другу сообщения, используя сеть межсоединений. В качестве примеров мультикомпьютеров можно назвать IBM SP/2, Intel/Sandia Option Red, Wisconsin COW.

При отсутствии памяти совместного использования в аппаратном обеспечении предполагается определенная структура программного обеспечения. В мультекомпьютере невозможно иметь одно виртуальное адресное пространство.

Если процессу 1 потребовались данные, не содержащиеся в его локальной памяти, то ему надо каким-то образом определить, какой процессор содержит эти данные и послать этому процессору сообщение с запросом копии данных. Процессор 1 блокируется до получения ответа. Когда процессор 2 получает сообщение, программное обеспечение его анализирует и отправить необходимые данные. Когда процессор 1 получает необходимые данные, программное обеспечение разблокируется и продолжает работу.

В мультикомпьютере для взаимодействия между процессорами часто используются примитивы *send* и *receive*. Программное обеспечение мультикомпьютера имеет более сложную структуру, чем программное обеспечение мультипроцессора. При этом основной проблемой становится правильное разделение данных и разумное их размещение. В мультипроцессоре размещение частей данных не влияет на правильность решения задачи, хотя может повлиять на производительность. Таким

образом, программировать мультикомпьютер гораздо сложнее, чем мультипроцессор.

Сточки зрения аппаратной реализации, гораздо проще и дешевле построить большой мультикомпьютер, чем мультипроцессор с таким же количеством процессоров. Реализация общей памяти, разделяемой несколькими сотнями процессоров, -- весьма сложная задача, а построить мультикомпьютер, содержащий 10000 процессоров и более, довольно легко.

Известны попытки создания гибридных систем, которые достаточно легко строить и относительно легко программировать. Это привело к осознанию того, что совместную память можно реализовывать по-разному, находя компромиссы между достоинствами и недостатками каждого похода. Все последние исследования в области архитектур с параллельной обработкой направлены на создание гибридных форм. Важно получить такую систему, которая *расширяема*, т.е. будет исправно работать при добавлении новых элементов.

Один из подходов основан на том, что современные компьютерные системы состоят из рада уровней. Это дает возможность реализовать общую память на любом из нескольких уровней, как это показано на рис. 6.3.

Рис. 6.3 а

Рис. 6.3 б

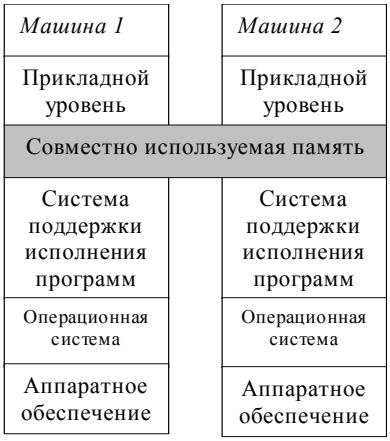


Рис. 6.3 в

Второй подход – использовать аппаратное обеспечение мультикомпьютера и операционную систему, которая моделирует разделенную память, обеспечивая единое виртуальное адресное пространство, разбитое на страницы. При таком подходе, который называется *DSM (Distributed Shared Memory – распределенная совместно используемая память)*, каждая страница расположена в одном из блоков памяти. Каждая машина содержит свою собственную виртуальную память и собственные таблицы страниц. Если процессор совершает операцию считывания или записи над одной из страниц, которой у него нет, происходит прерывание операционной системы. Затем ОС находит нужную страницу и требует, чтобы процессор, обладающий нужной страницей, преобразовал ее в исходную форму и послал по сети межсоединений. Когда страница достигла пункта назначения, она отображается в память и выполнение программы продолжается. По

существу, операционная система просто вызывает недостающие страницы не с диска, а из памяти.

Третий подход – реализация общей разделенной памяти на уровне программного обеспечения. При таком подходе абстракцию разделенной памяти создает язык программирования, и эта абстракция реализуется компилятором. При таком подходе общая разделенная память реализуется только программными средствами без вмешательства операционной системы и аппаратного обеспечения.

6.1.2. Сети межсоединений

Итак, мультикомпьютеры связываются через сети межсоединений. Мультикомпьютеры и мультипроцессоры очень сходны в этом отношении, т.к. мультипроцессоры также содержат модули памяти, которые также должны быть связаны между собой и с процессором. Основная причина сходства коммуникационных связей в мультипроцессоре и мультикомпьютере заключается в том, что в обоих случаях применяется передача сообщений. Сети межсоединенй могут состоять максимум из пяти компонентов:

1. Центральные процессоры.
2. модули памяти.
3. Интерфейсы.
4. Каналы связи.
5. Коммутаторы.

Интерфейсы – устройства, которые вводят и выводят сообщения из центральных процессоров и модулей памяти.

Каналы связи – каналы, по которым передаются биты. Каналы могут быть электрическими или оптико-волоконными, последовательными или параллельными. Каждый канал связи характеризуется максимальной пропускной способностью. Каналы могут быть симплексными (передавать только в одном направлении, полудуплексными (передавать в обоих направлениях, но не одновременно), дуплексными (передавать в обоих направлениях одновременно.

Коммутаторы – устройства с несколькими входными и несколькими выходными портами.

При разработке и анализе сети межсоединений важно учитывать следующие моменты:

- Топология (способ расположения компонентов);
- реализация системы переключения и осуществление связи между ресурсами;
- алгоритм выбора маршрута для доставки сообщений.

Топология

Топология сети межсоединений определяет, как расположены каналы связи и коммутаторы (это может быть кольцо, решетка, дерево и т.д.).

Топологии изображают в виде графов, в которых дуги (ребра) – каналы связи, вершины (узлы) – коммутаторы. Степень вершины для инженеров – *коэффициент разветвления*.

Расстояние между двумя вершинами – число ребер (дуг) которые нужно пройти, чтобы попасть из одной вершины в другую. *Диаметр графа* – максимальное расстояние между двумя вершинами. *Диаметр сети соединений* определяет самую большую задержку при передаче пакетов от одного процессора к другому или от процессора к памяти.

Пропускная способность сети межсоединений – количество данных, которое она может передать в секунду.

Бисекционная пропускная способность – минимальная из всех возможных (минимальный разрез сети). По мнению многих разработчиков, бисекционная пропускная способность – это самая важная характеристика сети.

Сети межсоединений можно характеризовать по их *размерности*. Размерность определяется по числу возможных вариантов перехода. Если выбора нет (один путь) – сеть нульмерная (рис. 6.4 а-в); два варианта (направо/налево) – одномерная (рис. 6.4 г); четыре варианта (направо/налево/вверх/вниз) – двумерная и (рис. 6.4 д,е) т.д. Трехмерная размерность – куб – приведена на рис. 6.4.ж, черырехмерный куб – рис. 6.4 з. N-мерный куб называют гиперкубом.

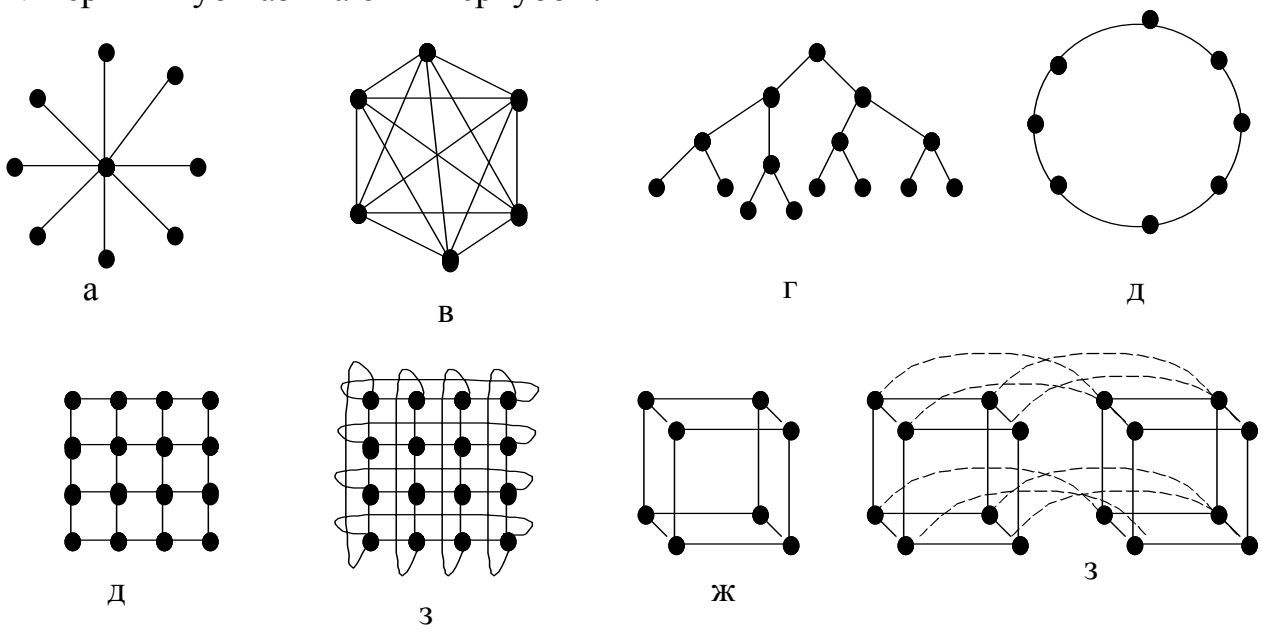


Рис.6.4

Коммутация

На рис. 6.5 приведена сеть межсоединений с четырьмя коммутаторами. На рисунке каждый коммутатор имеет 4 входных и 4 выходных порта. Каждый коммутатор содержит несколько центральных процессоров. Задача коммутатора – принимать пакеты, которые приходят на любой входной порт и отправлять пакеты из соответствующих выходных портов. Каждый

выходной порт связан с входным портом другого коммутатора через последовательный или параллельный канал (пунктирная линия). Существуют специальные сигналы для управления каналами. Параллельные каналы характеризуются более высокой производительностью, но в них возникает проблема *расфазировки данных* (нужно быть уверенным, что все биты прибывают одновременно) и они стоят гораздо дороже.

Существует несколько стратегий переключений.

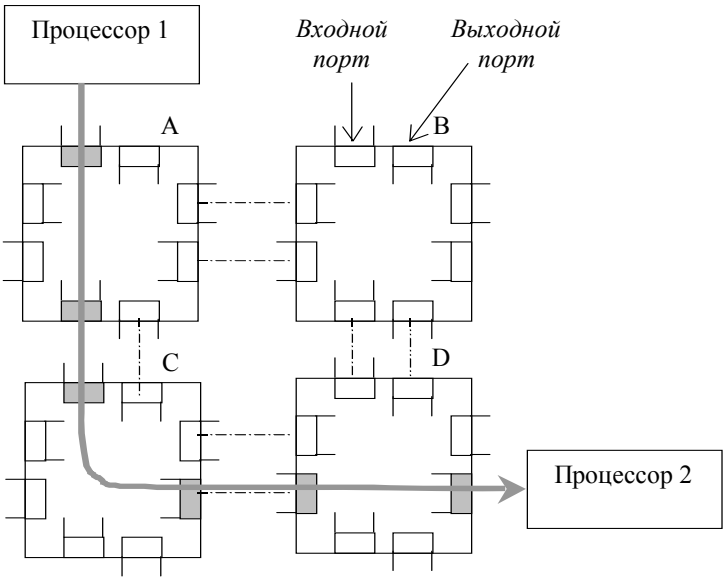


Рис. 6.5

Коммутация каналов – перед тем, как послать пакет, весь путь от начала до конца резервируется заранее. Все порты и буферы затребованы заранее и поэтому биты на полной скорости перемещаются от источника к потребителю. На рис. 6.5 – резервируется канал с использованием трех входных и трех выходных портов.

Коммутация с промежуточным хранением – здесь не требуется предварительного резервирования. Из исходного пункта посылается целый пакет к первому коммутатору, где он хранится целиком. Затем он передается следующему коммутатору и так до тех пор, пока не прибудет к месту назначения. Процесс перемещения пакета приведен на рис. 6.6. Никакого предварительного резервирования ресурсов не требуется.

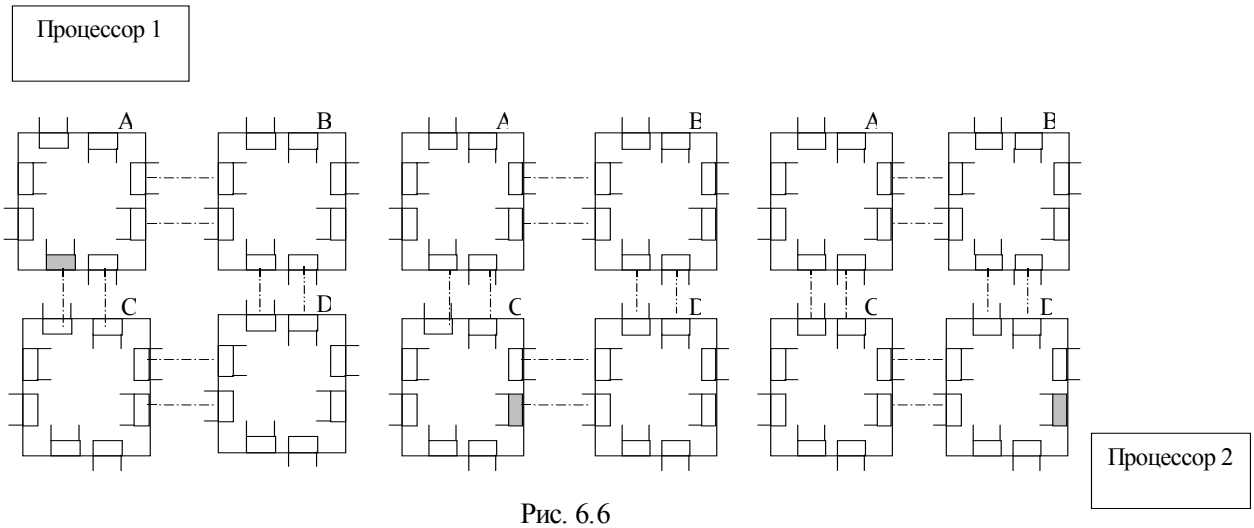


Рис. 6.6

Коммутаторы с промежуточным хранение должны отправлять пакеты в буфер. Если бы не было *буферизации*, входящие пакеты, которым нужен занятый в данный выходной порт, пропадали бы. Применяется три метода буферизации.

При буферизации на входе один или несколько буферов связываются с каждым входным портом в форме очереди типа FIFO. Если пакет нельзя передать, то он просто ждет своей очереди. Но если пакет ожидает, то следующий пакет также ожидает очереди, даже если требуемый ему порт свободен. Такая ситуация называется *блокировкой начала очереди*.

Проблема может быть устранена с помощью *буферизации на выходе*. В этой системе буферы связаны с выходными портами. И при буферизации на входе и при буферизации на выходе с каждым портом связано определенное количество буферов. Если место недостаточно для хранения всех пакетов, то пакеты пропадают. Для разрешения этой проблемы используют *общую буферизацию*, при которой один буферный пул динамически распределяется по портам по мере необходимости. Однако такой метод требует достаточно сложного управления.

Хотя метод коммутации с промежуточным хранением гибок и эффективен, здесь возникает проблема с возрастающей задержки при передаче данных по сети межсоединений.

Существует идея разработки гибридной сети межсоединений, объединяющую в себе коммутацию каналов и коммутацию пакетов. Например, каждый пакет можно разделить на части и как только первая часть пакета поступила в коммутатор, ее можно передавать дальше, даже если оставшиеся части пакета еще не прибыли.

Такой подход отличается от коммутации каналов тем, что ресурсы не резервируются заранее. Следовательно, возможна конфликтная ситуация в соревновании за право обладания ресурсами. При коммутации *без буферизации пакетов*, если первый блок пакета не может двигаться дальше, оставшаяся часть пакета продолжает поступать в коммутатор. В худшем случае эта схема превращается в коммутацию с промежуточным хранением. При другом типе маршрутизации, т.н. «wormhole routing» (червоточина), если первый блок не может двигаться дальше, то в исходный пункт подается сигнал остановить передачу и пакет может растянуться на несколько коммутаторов. После освобождения ресурсов продвижение пакета продолжается. Это схоже с конвейерным выполнением команд в центральном процессоре: в любой момент времени каждый коммутатор выполняет часть работы.

Алгоритмы выбора маршрута

В любой сети соединений с размерностью от один и выше можно выбирать, по какому пути передавать пакеты от одного узла к другому. Правило, определяющее, какую последовательность узлов должен пройти

пакет при движении от исходного пункта к пункту назначения, называется *алгоритмом выбора маршрута*.

Алгоритмы выбора маршрута можно разделить на две категории: маршрутизация от источника и распределенная маршрутизация. При *маршрутизации от источника* источник определяет весь путь по сети заранее. Этот путь выражается списком из номеров портов, которые нужно будет использовать в каждом коммутаторе по пути к пункту назначения. Если путь лежит через k коммутаторов, то первые k байтов в каждом пакете будут содержать k номеров выходных портов, по 1 байту на каждый порт. Когда пакет доходит до коммутатора, первый байт отсекается и используется для определения выходного порта. Оставшаяся часть пакета используется для направления в соответствующий порт.

При *распределенной маршрутизации* каждый коммутатор сам решает, в какой порт отправить каждый приходящий пакет. Если выбор одинаков для каждого пакета, направленного к одному и тому же конечному пункту, то маршрутизация является *статической*. Если коммутатор при выборе принимает во внимание текущий трафик, то маршрутизация является *адаптивной*.

Популярным алгоритмом маршрутизации, который применяется для прямоугольных решеток с любым числом измерений, является *пространственная маршрутизация*. В соответствии с этим алгоритмом пакет сначала перемещается вдоль оси x до нужной координаты, и т.д.

6.1.3. Производительность

Цель создания компьютера параллельного действия – сделать так, чтобы он работал быстрее, чем однопроцессорная машина. Если цель не достигнута – нет смысла создавать такой компьютер. Кроме того, эта цель должна быть достигнута при минимальных затратах. Рассмотрим некоторые вопросы производительности, связанные с созданием архитектур параллельных компьютеров.

Метрика аппаратного обеспечения

В аппаратном обеспечении наибольший интерес представляет скорость работы процессоров, устройств ввода/вывода и сети. Поскольку скорость работы процессоров и устройств ввода/ вывода такая же как и в однопроцессорных системах, то особый интерес представляют те параметры, которые связаны с межсоединениями. Здесь два основных момента6 время ожидания и пропускная способность.

Полное время ожидания – это время, которое требуется на то, чтобы процессор отправил пакет и получил ответ. Обычно интерес представляет время ожидания для пакетов минимального размера (одно слово или небольшая строка кэш-памяти). Для различных схем коммутации это время различно.

Для *сетей с коммутацией каналов* время ожидания складывается из времени установки соединения и времени передачи. Для установки нужно выслать пробный пакет для резервации ресурсов, а затем передать сообщение об этом. Когда пакет готов, его можно передавать на полной скорости. Тогда для передачи пакета размером в p бит в одну сторону потребуется время T_s+p/b секунд (T_s – общее время установки, b – пропускная способность). Если схема дуплексная, т.е. время установки на ответ не требуется, то для получения ответа потребуется T_s+2p/b секунд.

При пакетной коммутации не нужно посылать пробный пакет заранее, но все равно требуется некоторое время установки T_a. Здесь время передачи равно T_a+p/b, но за этот период пакет доходит только до первого коммутатора. При прохождении через коммутатор возникает задержка T_d, которое состоит из времени обработки и задержки в очереди. Для n коммутаторов общее время ожидания в одну сторону составляет $T_a+n(p/b+T_d)+p/b$, где последнее слагаемое – копирование пакета из последнего коммутатора в пункт назначения.

Время ожидания в одну сторону *для коммутации без буферизации* пакетов и «червоточины» в лучшем случае будет приближаться к T_a+p/b, поскольку здесь нет пробных пакетов и нет задержки, обусловленной промежуточным хранением.

Следующая характеристика аппаратного обеспечения – пропускная способность. Существует несколько показателей пропускной способности. Бисекционную пропускную способность уже рассматривали.

Суммарная пропускная способность – вычисляется путем суммирования пропускной способности всех каналов связи. Это число показывает максимальное число битов, которые можно передать сразу.

Средняя пропускная способность каждого процессора. Пропускная способность сети должна быть согласована с пропускной способностью процессоров.

Теоретически возможная пропускная способность никогда не достигается. Это определяется тем, что каждый пакет содержит помимо собственно данных необходимую служебную информацию. Чем больше пакет – тем меньше удельный вес этой информации. С другой стороны – увеличение объема пакета увеличивает время ожидания ответа. Отсюда конфликт между достижением малого времени ожидания ответа и высокой пропускной способностью сети. Важно понимать, что купить высокую пропускную способность можно (например, расширив шину), но нельзя купить низкое время ожидания. Поэтому при разработке сетей межсоединений сначала добиваются снижения времени ожидания ответа, а уже потом решают задачи повышения пропускной способности.

6.1.4. Метрика программного обеспечения

Для пользователя компьютера параллельного действия ключевым является вопрос о коэффициенте ускорения, который показывает, насколько

быстрее работает программа в n-процессорной системе по сравнению с однопроцессорной. Идеальное повышение скорости – использование n процессоров заставляет программу работать в n раз быстрее. На практике все на так.

Пусть T_1 – время решения задачи на однопроцессорной системе, Tn – время решения той же задачи на n-процессорной системе.

Пусть W=Wck + Wpr, где W – общее число операций, Wpr – число операций, которые можно выполнить параллельно, Wck – число скалярных (не распараллеливаемых операций). Обозначим через t время выполнения одной операции; тогда получаем выражения для закона Амдала:

$$r = \frac{Wt}{\left(Wck + \frac{Wpr}{n}\right)t} = \frac{1}{a + \frac{1-a}{n}} \xrightarrow[n\to\infty]{} \frac{1}{a},$$

где $a = Wck / W$ – удельный вес скалярных операций.

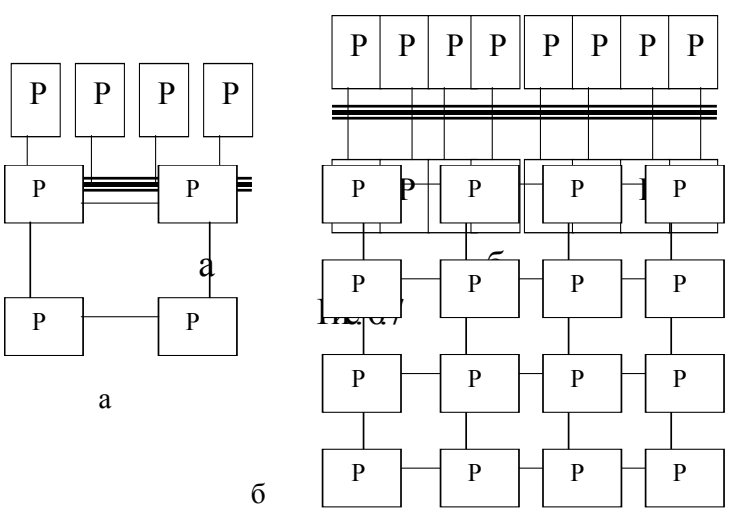
Закон Амдалаа определяет принципиально важные для параллельных вычислений положения:

- ускорение вычислений зависит как от потенциального параллелизма задачи (величина 1-*a*). так и от параметров аппаратуры (число процессоров n);
- предельное ускорение определяется свойствами задачи. Пусть, например, a=0.2, тогда ускорение не может превосходить 5 при любом числе процессоров.

Как достичь высокой производительности

Самый простой способ достижения высокой производительности – включать в систему дополнительные процессоры. Однако добавлять процессоры нужно таким образом, чтобы при этом не ограничивать повышение производительности системы. Система, к которой можно добавлять процессоры и получать соответствующее этому добавлению большую производительность, называется *расширяемой*.

Пусть у нас имеется 4 процессора, соединенные шиной (рис.6.7 а). Шина имеет некоторую пропускную способность, например, b Мбайт/с. Расширим систему до 16 процессоров (рис. 6.7 в). Тогда пропускная способность каждого процессора сократится в 4 раза. Такая система не является расширяемой.



Рассмотрим другую топологию межсоединений (рис. 6.8 а, б). В такой топологии при добавлении новых процессоров добавляются новые каналы, поэтому при расширении системы пропуск-

Рис. 6.8

ная способность на каждый процессор не снижается.

Пропускная способность – не единственный параметр. Добавления процессора к шине не увеличивает диаметр сети или время ожидания при отсутствии трафика, а добавление процессора к решетке – увеличивает. В такой конфигурации при увеличении числа процессоров в 4 раза влечет увеличение диаметра сети, а, следовательно, и среднее время ожидания в 2 раза.

В идеале расширяемая система при добавлении новых процессоров должна сохранять одну и ту среднюю же пропускную способность на каждый процессор и постоянное время ожидания. На практике сохранение достаточной пропускной способности на каждый процессор осуществимо, но время ожидания растет с увеличением размера сети.

Время ожидания часто является фатальным в производительности мелкомодульных и среднемодульных приложениях. Если программе требуются данные, которых нет в ее локальной памяти, то для получения их требуется тем больше времени, чем больше система. Системные разработчики применяют несколько различных технологий, которые позволяют сократить или, по крайней мере, скрыть время ожидания. Первая технология – *копирование данных*. Если копии данных хранить в нескольких местах, то можно сократить время доступа к ним.

Вторая технология – так называемая *упреждающая выборка*. Элемент данных вызывается еще до того, как он может понадобиться. Упреждающая выборка может быть автоматической, а может контролироваться программой. При реализации используются те же принципы, что и при работе с кэш-памятью.

Третья технология – это *многопоточная обработка*. В большинстве современных компьютерных систем поддерживается мультипрограммирование, при котором несколько процессов могут работать одновременно (или создавать иллюзию одновременной работы). Если переключение процессов происходит достаточно быстро, то когда один процесс блокируется, ожидая данные, аппаратное обеспечение быстро переключается на выполнение другого процесса.

Некоторые машины автоматически переключаются от процесса к процессу после каждой команды, чтобы скрыть длительное время ожидания. Такая идея была реализована в одном из первых суперкомпьютеров CDC 6600. Было объявлено, что он содержит 10 периферийных процессоров, которые работают параллельно. А на самом деле он содержал только один периферийный процессор, который моделировал работу 10 процессоров.

Четвертая технология – использование *неблокирующих записей*. Обычно при выполнении команды записи в память процессор ждет, пока она не закончится и только потом продолжает работать. При наличии неблокирующих записей процессор продолжает работать, пока отрабатывается команда записи в память. Продолжать работы при выполнении операции считывания из памяти сложнее, однако реализуемо, если применить исполнение с изменением последовательности.

6.1.5. Программное обеспечение

Без программного обеспечения с параллельной обработкой параллельное аппаратное обеспечение не принесет никакой пользы.

Существует 4 подхода к разработке программного обеспечения для параллельных компьютеров. Первый подход – добавление специальных библиотек численного анализа к обычным последовательным языкам. Например библиотечная процедура для решения некоторой численной задачи может быть вызвана из последовательной программы, после чего она будет выполняться на параллельном процессоре, а программист может даже не знать об этом. Недостаток – параллелизм может применяться только в нескольких процедурах, а основная часть программы останется скалярной.

Второй подход – добавление специальных библиотек, содержащих примитивы коммуникации и управления. Здесь программист сам создает процесс параллелизма и управляет им, используя дополнительные примитивы.

Третий подход – добавление нескольких специальных конструкций к существующим языкам программирования, позволяющих, например, легко порождать новые параллельные процессы. Такой подход очень широко используется и во многие языки программирования включены элементы параллелизма.

Четвертый подход – ввести совершенно новый язык специально для параллельной обработки. Достоинства очевидны, а недостаток в том, что программист должен учить новый язык программирования.

Существует достаточно много библиотек, расширений языков программирования и новых языков, которые дают достаточно широкий спектр возможностей, но их трудно классифицировать. Основные вопросы, которые формируют основу программного обеспечения для компьютеров параллельного действия:

1. Модели управления.
2. Степень распараллеливания процессов.
3. Вычислительные парадигмы.
4. Методы коммуникации.
5. Базисные элементы коммуникации.

Модели управления

Основной вопрос в работе программного обеспечения – сколько будет потоков управления, один или несколько.

В первой модели существует одна программа и один счетчик команд, но несколько наборов данных. Каждая команда выполняется над всеми

наборами данных одновременно разными обрабатывающими элементами. Такая модель имеет очень большое значение для аппаратного обеспечения. Это значит, что каждый обрабатывающий элемент представляет собой АЛУ и память, без схемы декодирования команд. Один центральный процессор вызывает команды и сообщает всем АЛУ, что делать дальше.

Альтернативная модель предполагает несколько потоков управления, каждый из которых содержит свой счетчик команд, регистры и локальные переменные. Каждый поток управления выполняет свою собственную программу над своими данными, при этом он время от времени может взаимодействовать с другими потоками управления.

Степень распараллеливания процессов

Параллелизм управления можно вводить на разных уровнях. На самом низком уровне элементы параллелизма могут содержаться в отдельных машинных командах (например, IA-64). Программисты обычно не знают о существовании такого параллелизма – он управляется компилятором или аппаратным обеспечением.

На более высоком уровне мы приходим к параллелизму на уровне блоков, который позволяет программисту самому контролировать, какие блоки будут выполняться последовательно, а какие параллельно.

Другая форма параллелизма – создание или порождение для каждого процесса нескольких потоков, каждый из которых работает в адресном пространстве этого процесса. Каждый поток имеет свой счетчик команд, свои регистры и стек, но разделяет все остальное адресное пространство (а также все глобальные переменные) со всеми другими потоками. *(Замечание: в отличие от потоков, разные процессы не разделяют общее адресное пространство.)* Потоки работают независимо друг от друга, иногда на разных процессорах. В одних системах ОС располагает информацией обо всех потоках и осуществляет планирование потоков; в других системах каждый пользовательский процесс сам выполняет планирование потоков и управляет потоками.

Наконец, параллелизм на уровне еще более крупных структурных единиц – несколько независимых процессов, которые вместе работают над решением одной задачи. В отличие от потоков, независимые процессы не разделяют общее адресное пространство, поэтому задача должна быть разделена на достаточно большие куски, каждый из которых создает процесс.

Вычислительные парадигмы

В большинстве параллельных программ, особенно тех, которые содержат большое число потоков или независимых процессов, используется некоторая парадигма для структуризации их работы. Существует достаточно много парадигм, остановимся на самых популярных.

Первая парадигма – *SPMD (Single Program Multiple Data – одно программа, несколько потоков данных).* Хотя система состоит из нескольких

независимых процессов, они все выполняют одну и туже программу, но над разными наборами данных, все процессы выполняют одни и те же вычисления, только каждый в своем пространстве.

Вторая парадигма – конвейер с тремя процессами (рис. 6.9 а). Данные поступают в первый процесс, который трансформирует их и передает второму процессу для чтения и т.д. Если поток данных длинный, все процедуры могут быть заняты одновременно. Так работают конвейеры в системе UNIX.

Следующая парадигма – фазированное вычисление (рис. 6.9 б), когда работа разделяется на фазы, например, повторения цикла. Во время каждой фазы насколько процессоров работают параллельно, но если один из процессов закончит свою работу, он должен ждать до тех пор, пока остальные процессы не завершат работу.

Четвертая парадигма – *разделяй и властвуй* (рис. 6.9.в). В этом случае один процесс запускается, а затем порождает другие процессы, которым он может передать часть работы.

Пятая парадигма – replicated worker (рис. 6.9.г). Здесь существует центральная очередь и рабочие процессы получают задачи из этой очереди. Каждый раз, когда рабочий процесс завершает задачу, он получает из очереди следующую.

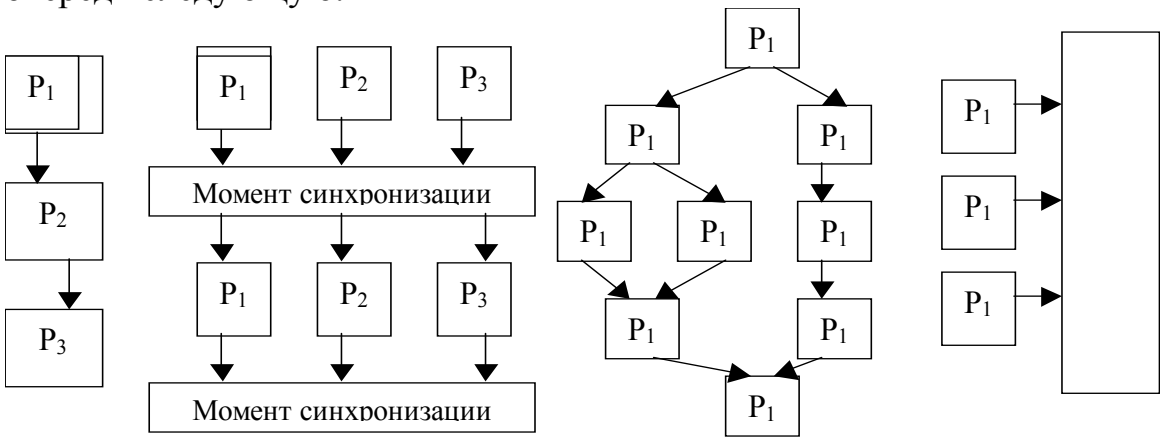


Рис. 6.9

Методы коммутации

Если программа разделена на части (процессы), которые работают параллельно, то эти части (процессы) должны каким-то образом взаимодействовать. Такое взаимодействие можно осуществить одним из двух способов: с помощью общих переменных и с помощью передачи сообщений. В первом случае все процессы имеют доступ к общей логической памяти.

В мультипроцессоре переменные могут разделяться между несколькими процессами с помощью отображения одной и той же страницы в адресное пространство каждого процесса. Затем общие переменные можно считывать

и записывать с помощью обычных команд записи и считывания. Даде в мультикомпьютере без разделенной физической памяти возможно логическое разделение переменных. Существует возможность разделения одного адресного пространства на мультикомпьютере, а также возможность разбиения на странице в сети. Процессы могут взаимодействовать между через общие переменные как в мультипроцессорах, так и в мультикомпьютерах.

При взаимодействии через передачу сообщений используются примитивы ***send*** и ***receive***. Один процесс выполняет примитив send, называя другой процесс в качестве пункта назначения. Как только второй процесс совершает ***receive***, сообщение копируется в адресное пространство получателя.

Следующий вопрос при передаче сообщений – количество получателей. Самый простой случай – один отправитель и один получатель (двухточечная передача сообщений). Однако в некоторых случаях требуется отправить сообщение всем процессам (*широковещание*) или определенному набору процессов (*мультивещание*).

Базисные элементы синхронизации

Параллельные процессы должны не только взаимодействовать, но и синхронизировать свои действия. Если процессы используют общие переменные, нужно быть уверенным, что пока один процесс записывает что-либо в общую структуру данных, никакой другой процесс не считает эту структуру. Т.о. требуется некоторая форма взаимного исключения, чтобы несколько процессов не могли использовать одни и те же данные одновременно.

Существуют различные базисные элементы, которые можно использовать для взаимного исключения. Это семафоры, блокировки и т.д. Все они позволяют процессу использовать какой-либо ресурс, и при этом никакой другой процесс доступа к этому ресурсу не имеют.

Во многих параллельных программах существует такой вид примитивов (базисных элементов), которые блокируют все процессы до тех пор, пока определенная фаза работы не завершится (рис. 6.9 б). Наиболее распространенным примитивом подобного рода является *барьер*. Когда процесс встречает барьер, он блокируется до тех пор, пока все процессы не наткнутся на барьер. Когда все процессы наткнулись на барьер, все процессы одновременно освобождаются и продолжают работу.

6.1.6. Классификация компьютеров параллельного действия

Хорошей классификации компьютеров параллельного действия пока не существует. Наиболее распространенной является классификация Флинна (Flynn). В основе классификации лежит два понятия: потоки команд и потоки

данных. Поток команд соответствует счетчику команд. Система с n процессорами имеет n счетчиков команд и, следовательно, n потоков команд.

Поток данных состоит из набора операндов. Потоки команд и данных в какой-то степени независимы, поэтому существует 4 комбинации:

- *SISD* – это классический последовательный компьютер фон Неймана. Он содержит один поток команд и один поток данных и может выполнять только одно действие одномоментно.
- *SIMD* – содержит один блок управления, выдающий по одной команде, но при этом имеется несколько АЛУ, которые могут обрабатывать несколько наборов данных одновременно. ILLIAC IV – прототип машин SIMD.
- *MISD* – несколько команд оперируют над одними данными. Некоторые считают машинами MISD машины с конвейерами.
- *MIMD* – несколько независимых процессов работают как часть большой системы. В эту категорию попадает большинство параллельных процессоров и мультикомпьютеры и мультипроцессоры.

На рис. 6.10 приведена расширенная классификация Флинна.

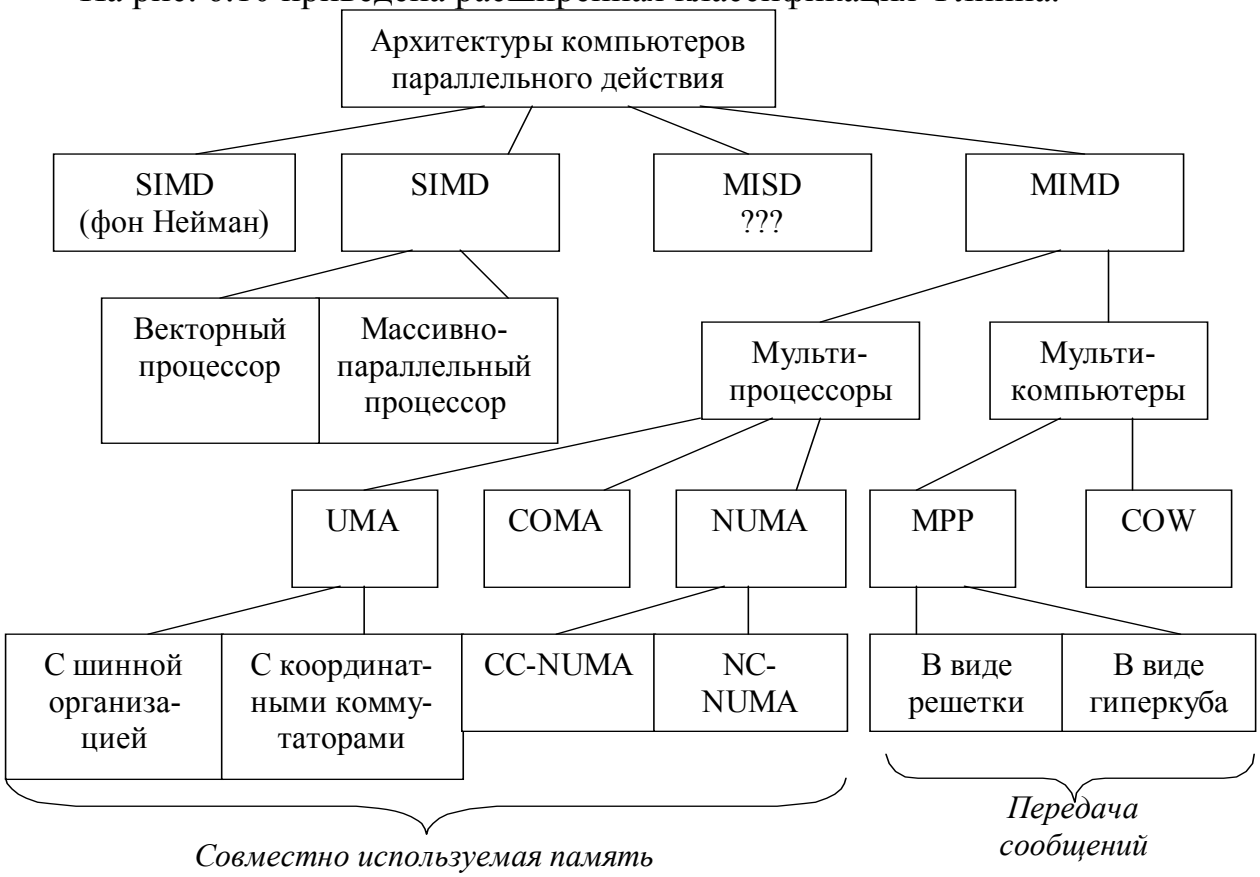


Рис. 6.10

Категория MIMD распалась на мультипроцессоры (машины с совместно используемой памятью) и мультикомпьютеры (машины с передачей

сообщений). Существует три типа мультипроцессоров. Они различаются по способу реализации памяти совместного использования. Это:

- *UMA (Uniform Memory Access)* – архитектура с однородным доступом к памяти.
- *NUMA (Non Uniform Memory Access)* – архитектура с необнородным доступом к памяти.
- *COMA (Cache Only Memory Access)* – архитектура с доступом только к кэш-памяти.

В машинах UMA каждый процессор имеет одно и тоже время доступа к памяти. Такая однородность делает производительность предсказуемой, а этот факт очень важен для написания эффективной программы.

Мультипроцессор NUMA, не обладает этим свойством. Обычно есть такой модуль памяти, доступ к которому осуществляется быстрее, чем к другим. Это позволяет поместить в «быстрый» модуль памяти наиболее часто используемые данные. Это позволяет повысить эффективность программ. Машины COMA тоже с неоднородным доступом.

Во вторую категорию MIMD попадают мультикомпьютеры и не имеют памяти совместного использования на архитектурном уровне. Поскольку мультикомпьютеры не имеют доступа к отдаленным модулям памяти, то они иногда называются машинами *NORMA (NO Remote Memory Access* – без доступа к отдаленным модулям памяти).

Мультикомпьютеры можно разделить на две категории.

- MPP (Massively Parallel Processor) – процессоры с массовым параллелизмом, дорогостоящие компьютеры, которые состоят из большого количества процессоров, связанных высокоскоростной коммуникационной сетью.
- NOW (Network of Workstations), COW (Cluster of Workstations) – сети рабочих станций и кластеры рабочих станций, которые связываются при помощи уже имеющихся соединений.

6.2. КОМПЬЮТЕРЫ SIMD

Компьютеры SIMD содержат один блок управления, который выполняет команды по одной, но каждая команда оперирует несколькими элементами данных.

6.2.1. Массивно-параллельные процессоры

Идея массивно-параллельных процессоров была предложена в конце 50-х годов, но реально такой процессор появился спустя 10 лет в Иллинойском университете и был он разработан для NASA. После этого другие компании создали несколько коммерческих компьютеров подобного типа, в том числе CM-2 и Maspar MP2, но они не пользовались популярностью на рынке.

В массивно-параллельном компьютере содержится один блок управления, который выдает команду нескольким обрабатывающим элементам. Каждый элемент состоит из процессора или усовершенствованного АЛУ и, как правило, локальной памяти.

Хотя все массивно-параллельные процессоры соответствуют общей модели, они могут отличаться друг от друга по следующим основным признакам.

1. Структура обрабатывающего элемента. Самые простые обрабатывающие элементы – 1-битные АЛУ (как в СМ-2). В такой машине каждое АЛУ получает два бита в качестве операнда из своей локальной памяти и бит из слова состояния программы (например, бит переноса). Результат операции – 1 бит данных и несколько флаговых битов. Чтобы сложить два 32-битных числа блоку управления надо транслировать команду у сложения 32 раза. Казалось бы быстродействие очень низкое, но количество обрабатывающих элементов очень велико (в СММ-2 – 2^{32}). Таким образом достигается высокое быстродействие.

   Обрабатывающим элементом может быть 8-битное, 32-битное или более мощное устройство, способное выполнять операции с плавающей точкой. Выбор типа обрабатывающего элемента определяется от типа целей машины.
2. Связь обрабатывающих элементов друг с другом. Здесь могут быть применены различные топологии. Однако наибольшее распространение получили прямоугольные решетки, поскольку они подходят для решения задач с матрицами и отображениями и хорошо применимы к большим размерам, поскольку при добавлении новых элементов пропускная способность автоматически увеличивается.
3. Какова локальная автономия обрабатывающих элементов. Блок управления сообщает, какую команду надо выполнить. Но во многих массивно-параллельных процессоров на основе анализа некоторых локальных данных может решать, выполнять эту команду или нет. Эта особенность существенно повышает гибкость системе в целом.

6.2.2. Векторные процессоры

Векторные процессоры более популярны на компьютерном рынке. Машины, разработанные Сеймуром Креем (Seymour Cray) для Cray Research (сейчас это часть Silicon Graphics) доминировали в научной сфере на протяжении десятилетий.

Напомним, что вектор – это одиночный массив, который образуется из многомерного массива, если один из индексов не фиксирован и пробегает все значения в диапазоне его изменения. Структура конвейерного процессора для векторной обработки приведена на рис. 6.11. Такая машина получает на входе два n-мерных вектора и обрабатывает соответствующие элементы

параллельно, используя векторное АЛУ, которое может оперировать с n элементами одновременно.

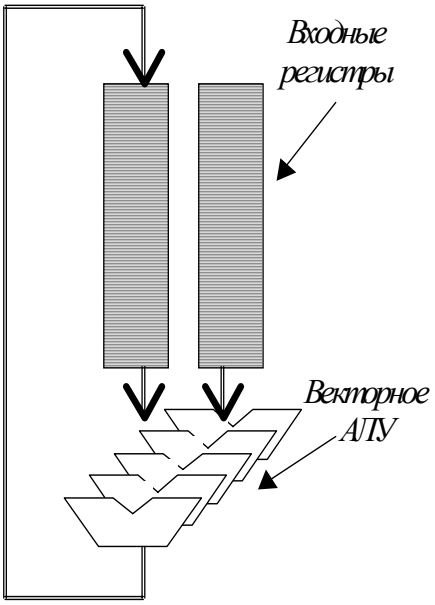


Рис. 6.11

На практике суперкомпьютеры редко строятся по схеме, приведенной на рис. 6.11. Причина этого не техническая (реализовать такую структуру можно), а экономическая. Создание компьютера с 64 высокоскоростными процессорами дорого обойдется.

Обычно векторные процессоры сочетаются с конвейерными. Операции с плавающей точкой достаточно сложны, они требуют выполнения нескольких шагов и для ускорения целесообразно использовать конвейер. Существенное различие между использованием конвейера для операций над векторами по сравнению с применением к обычным командам – в отсутствии скачков при работе с векторами. Каждый цикл используется полностью.

Векторный суперкомпьютер Crey-1

Векторный суперкомпьютер Crey-1 был построен в 1976 году. В настоящее время он не используется, но имеет достаточно простую архитектуру RISC, поэтому его удобно рассматривать как типичный пример, и к тому же его архитектуру можно встретить во многих современных компьютерах.

Машина Cray-1 (рис. 6.12) регистровая, большинство команд 16-битные, состоят из 7-битного кода операций и трех 3-битных номеров регистров для трех операндов. Имеется 5 типов регистров. Восемь 24-битных регистров A используются для обращения к памяти. 64 24-битных регистра B используются для хранения регистров A, когда они не нужны, чтобы сократить количество обращений в память. Восемь 64 регистров S предназначены для хранения скалярных величин (целых и с п.з.) Значения этих регистров можно использовать в качестве операндов. 64 64-битных регистров T – для хранения регистров S для сокращения операций в память.

Имеется группа из восьми векторных регистров. Каждый регистр может содержать 64-элементный вектор с п.з. В одной 16-битной команде можно сложить, вычесть или умножить два вектора. Векторные регистры могут загружаться из памяти, сохраняться в памяти, но такие перемещения выполнять очень невыгодно. Во всех векторных операциях используются регистровые команды.

Не всегда в суперкомпьютерах операнды расположены в регистрах. Машина Cyber 205 выполняла операции на векторами в памяти. Преимущество – нет ограничений на длину вектора. Недостаток – снижение быстродействия.

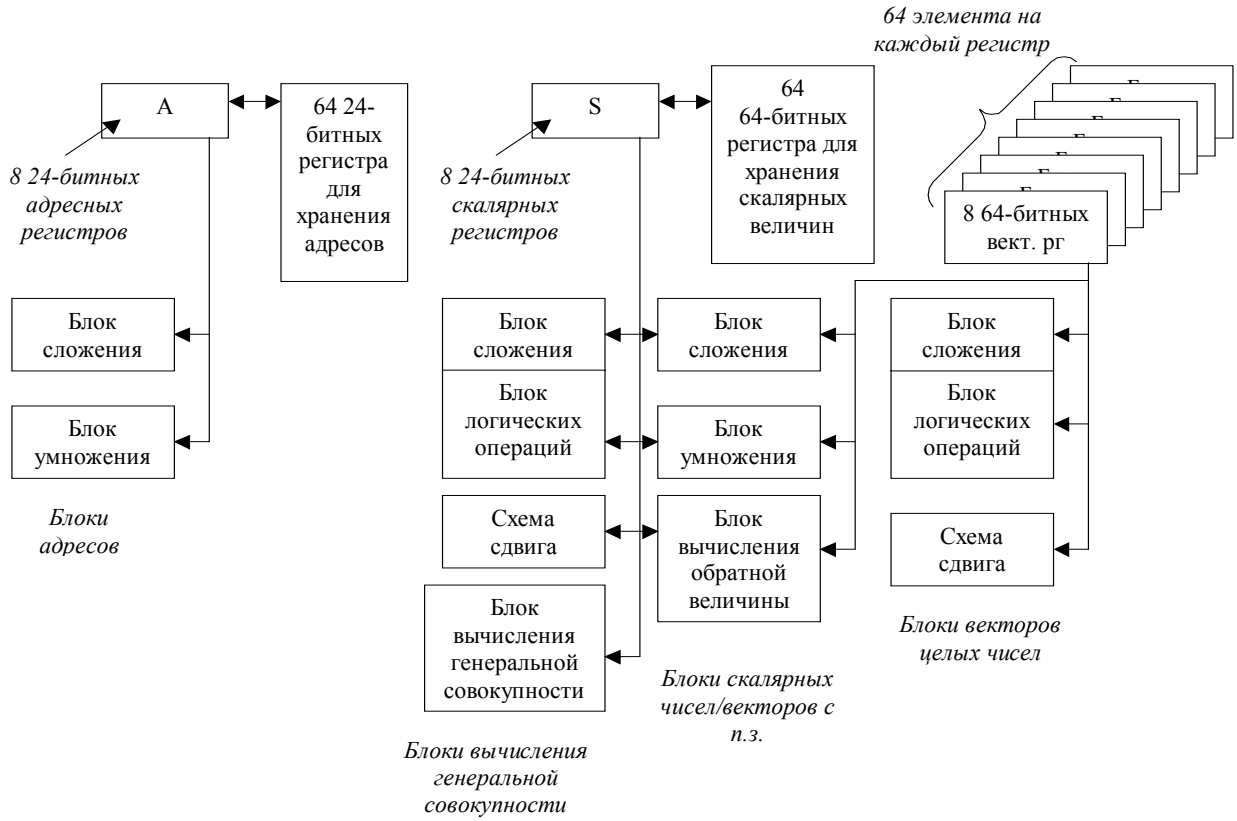
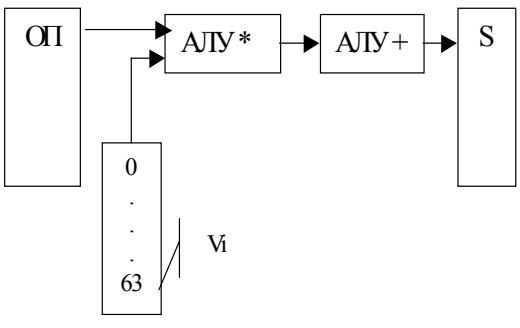


Рис. 6.12

Cray-1 содержит 12 функциональных блока. Два из них предназначены для арифметических действий с 24-битными адресами. Четыре – для операций с 64-битными целыми числами. Оставшиеся шесть – для работы над векторами, все они конвейеризированы. Cray-1 не имеет блока для целочисленного умножения.

Важной особенностью векторных конвейерных процессоров, используемой для ускорения вычислений, является механизм зацепления. Зацепление – такой способ вычислений, когда одновременно над вектором выполняется несколько операций. В частности, можно одновременно производить выборку вектора из памяти, умножение векторов, суммирование элементов вектора. Запишем:

LD L, Vi, A
ЗЦ Sn, Vi, B



Здесь команда ЗЦ задает одновременное выполнение операций в соответствии со схемой соединений (рис. 6.12). Использование механизма зацепления может повысить производи-

Рис. 6.13

тельность векторных ЭВМ в несколько раз.

Однако, высокое быстродействие достигается не на всех векторных операциях. Существуют операции, в которых ход дальнейших вычислений определяется в зависимости от результата каждой очередной элементарной операции над одним или парой операндов. К таким операциям относятся операции рассылки и сбора, которые можно определить следующими отрезками С-программ.

```
// рассылка                    //сбор
for (i=0; i<L; i++)            for (i=0; i<L; i++)
x[index[i]]=y[i];              y[i]=x[index[i]];
```

Целочисленный массив index содержит адреса операндов, разбросанных в произвольном порядке в памяти. Операция рассылки распределяет упорядоченный набор элементов y[i] по всей памяти в соответствии с комбинацией адресов в массиве index. Операция сбора действует наоборот. Подобного рода операции имеются в задачах сортировки, быстрого преобразования Фурье, при обработке графиков и др.

Быстродействие векторного процессора на таких операциях снижается до уровня быстродействия скалярного процессора.

Тактовый генератор Cray-1 работает с частотой 80 МГц, объем основной памяти – 8 Мбайт, а быстродействие Cray-1 достигало 80 млн. оп/с.

6.3. МУЛЬТИПРОЦЕССОРЫ С ПАМЯТЬЮ СОВМЕСТНОГО ИСПОЛЬЗОВАНИЯ

Мультипроцессор – компьютерная система, которая содержит несколько процессоров и одно адресное пространство, видимое для всех компьютеров. Он запускает одну копию операционной системы с одним набором таблиц, в том числе с таблицами, которые следят, какие страницы памяти заняты, а какие свободны. Когда процесс блокируется, его процессор сохраняет свое состояние в таблицах операционной системы, а затем, просматривает эти таблицы для нахождения другого процесса, который нужно запустить. Именно наличие одного отображения и отличает мультипроцессор от мультикомпьютера.

Если все процессоры имеют равный доступ ко всем модулям памяти и всем устройствам ввода-вывода и каждый процессор взаимозаменяем с другими процессорами, то такая система называется SMP (Symmetric MultiProcessor – симметричный мультипроцессор). Будем рассматривать именно такой тип системы.

6.3.1. Семантика памяти

Несмотря на то, что во всех мультипроцессорах процессорам предоставляется отображение одного адресного пространства, часто наряду с этим имеется множество модулей памяти, каждый из которых содержит какую-либо часть физической памяти. Процессоры и модули памяти

соединяются сложной коммуникационной сетью. Несколько процессов могут конфликтовать из-за обращения к одним и тем же переменным; некоторые сообщения могут быть доставлены не в том порядке, в котором были отправлены. Кроме того, существуют многочисленные копии одних и тех же блоков памяти и т.д.

Семантику памяти можно рассматривать как контракт между программным обеспечением и аппаратным обеспечением памяти. Если программное обеспечение соглашается следовать определенным правилам, то память соглашается выдавать определенные результаты. Основная проблема – каковы эти правила. Эти правила называются *моделями согласованности*. Было предложено и разработано множество таких правил.

Строгая согласованность

Самая простая модель – *строгая согласованность*. В такой системе при любом считывании из адреса X всегда возвращается самой последней записи в X. Программистам очень нравится такая модель, но она может быть реализована только следующим образом: должен быть один модуль памяти, который обслуживает все запросы по мере поступления, без кэш-памяти, без дублирования данных. А это очень сильно замедляет работу памяти.

Согласованность по последовательности

Согласованность по последовательности – при наличии нескольких запросов на чтение и запись аппаратное обеспечение определяет порядок всех запросов, но все процессоры наблюдают одну и ту последовательность запросов. Может возникнуть ситуация, при которой в одной и той же переменной обратятся несколько процессоров для записи и считывания. Т.е. возможна неоднозначность толкования последовательности обращений и процессы, обратившиеся к памяти практически одновременно (в течение одного цикла) могут считать различные результаты. Согласованность по последовательности гарантирует единый глобальный порядок записей, который виден всем процессорам. Если один процессор видит, что записано число 3, то и остальные процессоры видят то же самое.

Согласованность по последовательности очень полезна. Если несколько событий совершаются одновременно, существует определенный порядок, в котором эти события происходят, (сам порядок может определяться случайно) и все процессы наблюдают тот же самый порядок.

Однако есть модели согласованности, которые не гарантируют такого порядка.

Процессорная согласованность

Процессорная согласованность – более проигрышная модель, но зато ее легче реализовать на больших мультипроцессорах. Она имеет два свойства:

- Все процессоры воспринимают записи любого процессора в том порядке, в котором они начинаются;
- Все процессоры видят записи в слово памяти в том порядке, в котором они происходят.

В первом пункте говорится, что если процессор 1 начинает запись значений 1A, 1B, 1C в какое-то место в памяти именно в таком порядке, то все другие процессоры видят эти записи в таком же порядке. Второй пункт нужен, чтобы каждое слово в памяти имело определенное недвусмысленное значение после того, как процессор совершил несколько записей в это слово, а затем остановился. Все должны воспринимать последнее значение.

При таких ограничениях возможна ситуация, что процессор 2 начнет записи 2A, 2B, 2C одновременно с записями процессора 1. Другие процессоры, считывающие слова из памяти увидят последовательность из шести записей, например 1A, 2A, 1B, 2B, 1C, 2C или 1A, 1B, 1C, 2A, 2B, 2C и т.п. При процессорной согласованности не гарантируется, что каждый процессор видит одну и ту же последовательность. Единственное, что гарантируется совершенно точно, что никто не увидит последовательность 1B, 1A, …

Слабая согласованность

При использовании модели *слабой согласованности* записи, произведенные одним процессором, воспринимаются по порядку. Один процесс может увидеть 1A, 1B, а другой 1B, 1A. Чтобы внести порядок в этот хаос, в памяти содержатся элементы синхронизации или операции синхронизации. Когда выполняется синхронизация, все незаконченные записи завершаются, а новые не могут начаться пока не будут завершены все начатые и не будет проведена синхронизация. Синхронизация приводит память в стабильное состояние, когда не остается никаких незавершенных операций. Сами операции синхронизации согласованы по последовательности, т.е. все процессоры воспринимают один и тот же порядок.

При слабой согласованности время разделяется на последовательные периоды, разграниченные моментами синхронизации. Внутри периодов в последовательность может быть видна различными процессора по-разному, но в момент синхронизации происходит «выравнивание». Т.о. программное обеспечение вносит порядок в последовательность событий, хотя это и занимает некоторое время.

Свободная согласованность

Слабая согласованность не очень эффективный метод, поскольку он требует завершения всех операций памяти и задерживает выполнение новых операций до тех пор, пока старые не будут завершены. При *свободной согласованности* используется нечто похожее на критические секции программы. Идея состоит в следующем. Если процесс выходит за пределы критической области, это не значит, что все записи должны немедленно завершится. Требуется только, чтобы все записи были завершены до того, как процесс снова войдет в эту критическую область.

В такой модели операция синхронизации разделяется на две разные операции. Чтобы считать или записать общую переменную, процессор (т.е. его ПО) сначала должно выполнить операцию *acquire* над переменной синхронизации, чтобы показать получить монопольный доступ к общим разделяемым данным. Затем процессор может использовать эти данные по своему усмотрению, а затем процессор выполняет операцию *release* над переменной синхронизации, чтобы показать, что он завершил работу. Операция *release* не требует завершения незаконченных записей, однако она сама не может быть завершена, пока не закончатся все начатые записи. Новые операции памяти могут начаться сразу же.

Когда начинается новая операция *acquire,* производится проверка, все ли начатые операции *release* завершены. Если нет, то операции *acquire* будут задержаны. Это гарантирует, что все переменные в критической области будут обновлены. Такая схема сложнее, чем слабая согласованность, но она имеет преимущество: здесь не надо задерживать выполнение команд без необходимости.

6.3.2. Архитектуры UMA SMP с шинной организацией

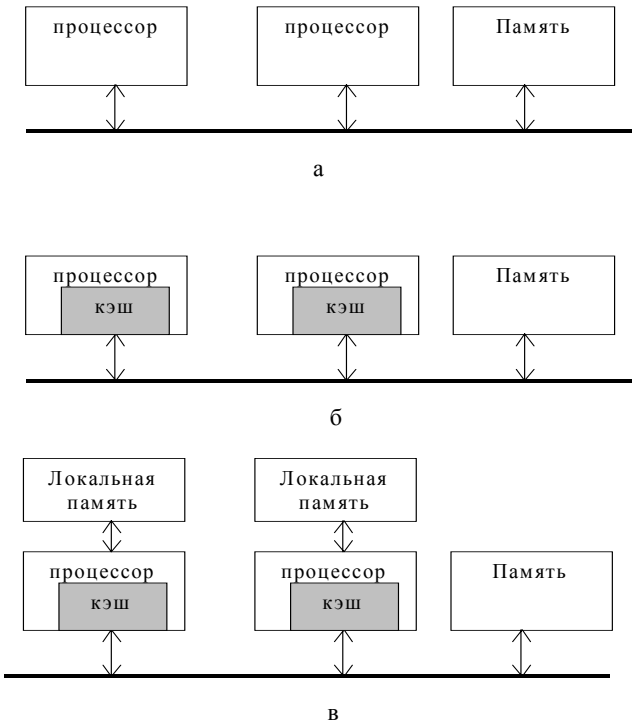


Рис. 6.14

В основе самых простых мультипроцессоров лежит одна шина (рис. 6.14 а). Если процессору нужно считать информацию из памяти, он проверяет, свободна ли шина.

Если шина занята, процессор ждет. При наличии двух или трех процессоров доступ к шине вполне управляем. При наличии большего числа процессоров (32, 64 и т.д.) производительность системы полностью ограничивается пропускной способностью шины, а большинство процессоров будут простаивать.

Для разрешения этой проблемы необходимо добавить кэш-память к

каждому процессору (рис. 6.14 б) Поскольку теперь слова можно из кэш-памяти, то и движения в шине будет меньше и система может поддержать большее число процессоров.

Следующая разработка – каждый процессор имеет не только кэш-память, но и свою локальную память, к которой он получает доступ через локальную шину (рис. 6.14 в). Чтобы оптимально использовать такую конфигурацию, компилятор должен поместить в локальные модули памяти весь текст программы, константы, данные, предназначенные только для чтения, стеки, локальные переменные. Общая разделенная память используется только для общих переменных. В большинстве случаев такое разумное размещение сильно сокращает количество данных, передаваемых по шине и не требует активного вмешательства со стороны компилятора.

Отслеживание изменений данных в кэш-памяти

Предположим, что память согласована по последовательности. Процессор 1 содержит в своей кэш-памяти строку. Процессор 2 пытается считать данные из этой же строки. Если не оговорено иное, процессор 2 получает в свою кэш-память копию строки. В этом нет ничего страшного, пока процессор 1 не внес изменения в строку кэш-памяти. В таком случае процессор 2 имеет устаревший данные, и нарушается принцип согласования данных по последовательности.

Такая проблема называется *непротиворечивостью кэшей*. Без решения этой проблемы нельзя использовать кэш-память в мультипроцессорных системах с общей шиной. Существуют алгоритмы, позволяющие разрешить эту проблему, которые называются *протоколом когерентности кэширования*. Эти протоколы различаются деталями, но все не допускают одновременного появления разных вариантов одной и той же строки в разных блоках кэш-памяти.

Во всех решениях контроллер кэш-памяти разрабатывается таким образом, чтобы кэш-память могла перехватить запросы на шине от других процессоров и кэш-памятей и предпринимала определенные действия в необходимых случаях. Такие устройства называются кэш-памятью с отслеживанием (snooping caches или snoopy caches).

Самый простой протокол когерентности кэширования называется сквозным кэшированием. Возможны 4 варианта, которые приведены в таблице.

Действие	Локальный запрос	Удаленный запрос
Промах при чтении	Вызов данных из памяти	
Попадание при чтении	Использование данных из локальной кэш-памяти	
Промах при записи	Обновление данных в основной памяти	
Попадание при записи	Обновление данных в кэш-памяти и основной памяти	Объявление элемента кэш-памяти недействительным

Суть протокола состоит в том, что в результате всех операций записи записываемое слово обязательно проходит через основную память.

Предположим, что кэш-1 записывает слово, которое присутствует в кэш-2. Если кэш-2 не произведен никаких действия, то в ней будут содержатся устаревшие данные. Поэтому, элемент, содержащий измененное слово просто помечается как недействительный (удаляется). Все кэш-памяти постоянно отслеживают изменения на шине и каждый раз, когда записывается слово, его обновляют в кэш-памяти инициатора, в основной памяти и удаляют из остальных кэш-памятей.

Возможны различные варианты реализации этого протокола. Можно не удалять измененное слово из остальных кэшей, а заменять новым. Т.е необходимо выбрать между *стратегией обновления* и *стратегией признания данных недействительными*. Сообщения об обновлении данных несут полезную нагрузку и, следовательно, они больше протокола о признании данных недействительными.

Протокол MESI

Один из популярных протоколов с обратной записью называется MESI (по первым буквам четырех состояний). В его основе лежит *протокол однократной записи*. Этот протокол используется в Pentium II и других процессорах для отслеживания шины. Каждый элемент кэш-памяти может находится в одном из четырех состояний:

1. Invalid – элемент кэш-памяти содержит недействительные данные.
2. Shared – несколько кэшей могут содержать данную строку, основная память обновлена.
3. Exclusive – никакой другой кэш не содержит эту строку, основная память обновлена.
4. Modified – элемент действителен, основная память недействительна, копий элемента не существует.

При загрузке процессора все элементы кэш-памяти помечаются как недействительные. При первом считывании из основной памяти нужная строка вызывается в кэш-память данного процессора и помечается как E. При последующих считываниях процессор использует эту строку, но не использует шину. Другой процессор может вызвать эту же строку, но при отслеживании исходный держатель (процессор 1) узнает, что он не единственный и объявляет, что у него есть копия. Обе копии помечаются состоянием S. При последующих чтениях кэшированных строк в состоянии S процессор не использует шину и не меняет состояние элемента.

Если процессор 2 произведет запись в строку кэш-памяти с состоянием S, то он помещает сигнал о недействительности на шину, который сообщает всем процессорам, что нужно отбросить все копии. Соответствующая строка переходит в состояние M. Эта строка не записывается в основную память.

Варианты:

- Если процессору 3 необходимо считать эту строку происходит следующее. Процессор 2, который содержит эту строку, знает, что копия в основной памяти недействительна, поэтому он передает на шину сигнал, который сообщает процессору 3 чтобы он подождал, пока процессор 2 запишет данные в основную память. Как только строка попала в основную память, процессор 3 вызывает из памяти копию строки и в обоих кэшах эта строка помечается как S.
- Если процессор 1 собрался записать слово в этой строке, то процессор 2 видит это и передает сигнал о том, что нужно подождать, пока строка не будет записана в основную память. Когда строка записана, процессор помечает собственную строку как недействительную.

6.3.3. Мультипроцессоры UMA с координатным коммутатором

Даже при всех возможных оптимизациях использование только одной шины ограничивает размер мультипроцессора UMA до 16 или 32 процессоров.

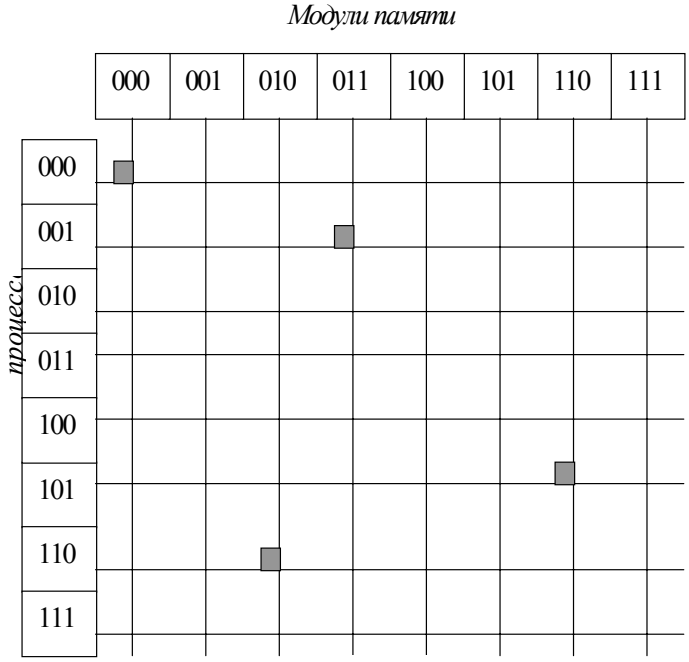


Рис. 6.15

Самая простая схема соединения процессоров с k блоками памяти – координатный коммутатор (рис 6.14). Координатный коммутатор представляет собой *неблокируемую сеть*. Это значит, что процессор всегда будет связан с нужным модулем памяти. Никакого предварительного планирования не требуется. Недостатком координатного коммутатора является то, что число узлов растет как n^2. Поэтому координатные коммутаторы приемлемы для систем средних размеров.

Sun Enterprise 10000

Эта система состоит из одного корпуса с 64 процессорами. Координатный коммутатор *Gigaplahe-XP* запакован в плату, содержащую 8 гнезд на каждой стороне. Каждое гнездо вмещает плату (400 х 500 мм),содержащую 4 процессора UltraSPARC II на 333 МГц и ОЗУ на 4 Гбайт. Благодаря жестким требованиям к синхронизации и малому времени ожидания доступ к памяти вне платы занимает столько же времени, что и доступ к памяти на плате.

Иметь только одну шину для взаимодействия всех процессоров и всех блоков памяти неудобно, поэтому в системе Enterprise 10000 применяется другая стратегия. Здесь координатный коммутатор 16 x 16 для перемещения данных между основной памятью и боками кэш-памяти. Длина строки кэш-памяти составляет 64 байта, а ширина канала связи – 16 байт, строка кэш-памяти перемещается за 4 цикла. Координатный коммутатор работает от точки к точке, поэтому его нельзя использовать для сохранения совместимости по кэш-памяти.

По этой причине помимо координатного коммутатора имеются 4 адресные шины, которые используются для отслеживания строк в кэш-памяти (рис. 6.16). Каждая шина использует ¼ часть физического адресного пространства. Для выбора шины используется 2 адресных бита. В случае промаха кэш-памяти при считывании процессор должен считать данные из основной памяти и тогда он обращается к соответствующей адресной шине, чтобы узнать, нет ли нужной строки в других блоках кэш-памяти. Все 16 плат отслеживают все 4 адресные шины одновременно, поэтому, если ответа нет, это означает, что требуемая строка отсутствует в кэш-памяти и следует обратиться в основную память.

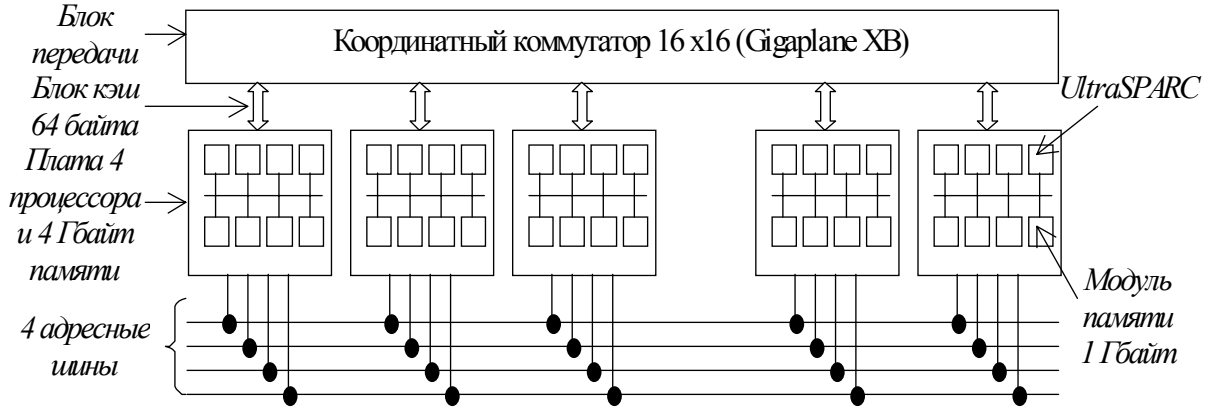


Рис. 6.16

Вызов из памяти происходит от точки к точке по координатному коммутатору по 16 байт. Цикл шины составляет 12 нс (83,3 МГц), и каждая адресная шина может отслеживаться в каждом цикле любой другой шины, т.е. возможно отслеживание 167 млн отслеживаний/с. Каждое отслеживание может потребовать передачи строки кэш-памяти в 64 байта, поэтому узел должен обеспечить передачу со скоростью 9.93 Гбайт/с. Строку кэш-памяти в 64 байта можно передать за 4 цикла (48 нс) при пропускной способности в 1.24 Гбайта/с за одну передачу. Поскольку узел может обрабатывать 16 передач одновременно, его максимальная пропускная способность – 19.87 Гбайт/с, а этого достаточно для поддержания скорости отслеживания, даже если учесть возможность возникновения конфликтной ситуации, при которой практическая пропускная способность снижается до 60% от теоретической.

Enterprise 10000 использует 4 отслеживающие шины параллельно и очень широкий коммутатор для передачи данных. Такая система преодолевает предел в 64 процессора. Но для существенного увеличения числа процессоров требуется иной подход.

6.3.4. Мультипроцессоры UMA с многоступенчатыми сетями

В основе другого подхода лежит коммутатор 2 x 2. Коммутатор содержит два входа и два выхода. Сообщения, приходящие на один из входов могут переключаться на любой выход. Коммутаторы 2 x 2 можно компоновать различными способами и получать *многоступенчатые сети*. Один из возможных вариантов – сеть omega (рис. 6.17). В этой сети соединены 8 процессоров и 8 модулей памяти, используя 12 коммутаторов. Для n процессоров и n модулей памяти потребуется $\log_2 n$ ступеней, n/2 коммутаторов на каждой ступени. Всего $(n\log_2 n)/2$ коммутаторов, что значительно лучше, чем n^2.

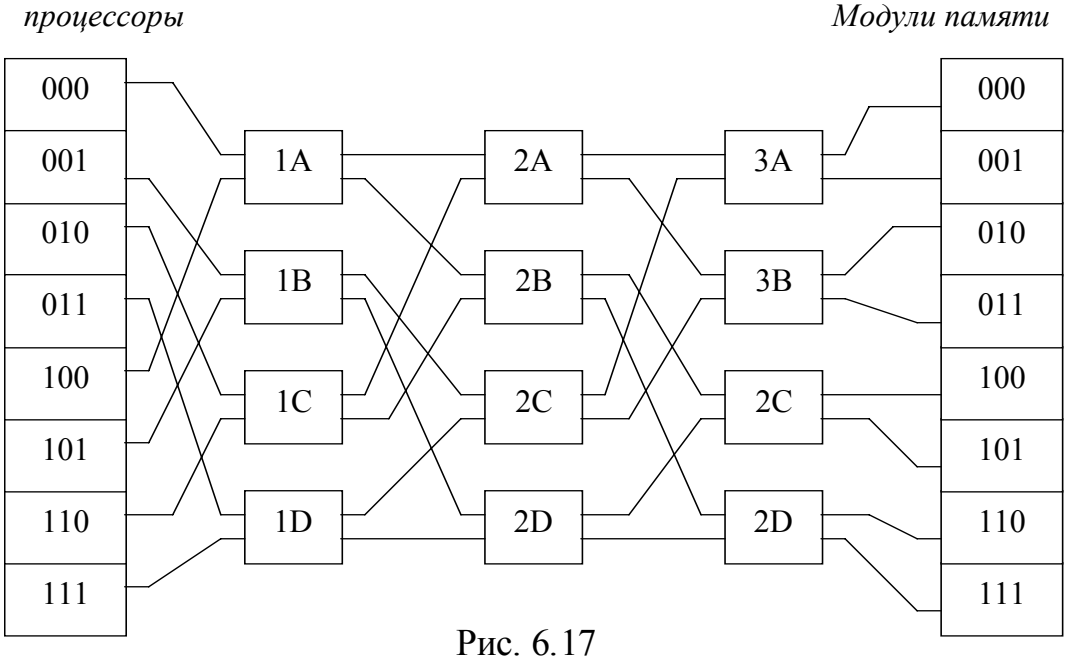


Рис. 6.17

Рисунок разводки сети omega называют *полным тасованием*. По мере прохождения сообщения по сети последовательно используются биты адреса получателя. Эти биты замещаются битами адреса отправителя, т.о. получатель знает, откуда пришел запрос.

Рассмотрим, что произойдет, если процессору 001 нужно записать обратиться в модуль памяти 001, одновременно процессор 000 обращается в модуль 000. Запрос может вступить в конфликт с запросом процессора 001 на коммутаторе 3A. Одному из запросов придется подождать. В отличии от координатного коммутатора, сеть omega – блокируемая сеть.

В такой сети желательно равномерно распределить обращения к памяти по модулям. Один из возможных способов – использовать младшие биты в качестве номера модуля памяти. В такой памяти, где последовательные слова располагаются в разных модулях памяти, называется *расслоенной*.

6.3.5. Мультипроцессоры NUMA

Для дальнейшего увеличения количества процессоров в системе необходимо находить дополнительные технические решения.

Мультипроцессоры NUMA (NonUniform Memory Access – с неоднородным доступом к памяти). Как и мультипроцессоры UMA, они обеспечивают единое адресное пространство для всех процессов, но в отличие от машин UMA, доступ к локальным модулям памяти происходит быстрее, чем к удаленным. Следовательно, все программы UMA будут работать без изменений на машинах NUMA, но производительность будет хуже, чем на машинах UMA с той же тактовой частотой.

Машины NUMA три ключевые характеристики, которыми все они обладают и которые в совокупности отличают их от других мультипроцессоров.

1. Существует одно адресное пространство, видимое для всех процессоров.
2. Доступ к удаленной памяти осуществляется с использованием команд load и store.
3. Доступ к удаленной памяти происходит медленнее, чем доступ к локальной памяти.

Если время доступа к удаленной памяти не скрыто (кэш-память отсутствует), то такая система называется NC-NUMA (No Caching NUMA – NUMA без кэширования. Если присутствуют согласованные кэши, то система называется CC-NUMA (Coherenet Cache NUMA – NUMA с согласованной кэш-памятью).

Упрощенная структурная схема системы NC-NUMA приведена на рис. 6.18. Запрос памяти приходит в блок управления памятью. Определяется, находятся ли требуемые данные в локальной памяти. Если да, то запрос отправляется по локальной шине. Если данных в локальной памяти нет, то запрос отправляется по системной шине к системе, к которой присутствуют данные. В первых таких системах (70-е годы) обращение к системной шине требовало времени в 10 раз большее, чем обращение к локальной памяти.

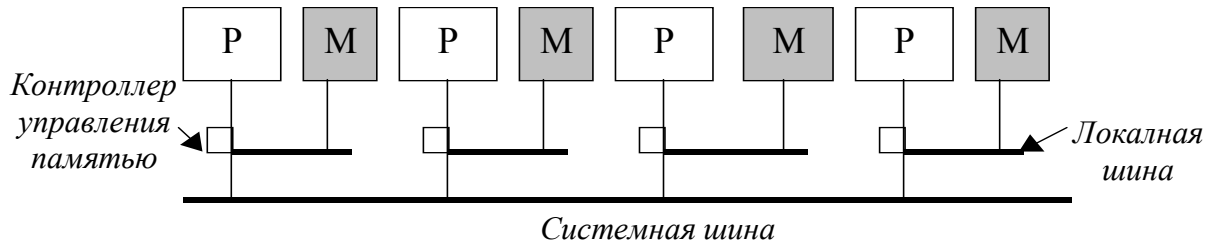


Рис. 6.18

Согласованность памяти гарантировало отсутствие кэш-памяти, т.к. каждое слово имеется в единственном экземпляре. Машина NC-NUMA использует сложное программное обеспечение для перемещения страниц, чтобы максимально увеличить производительность.

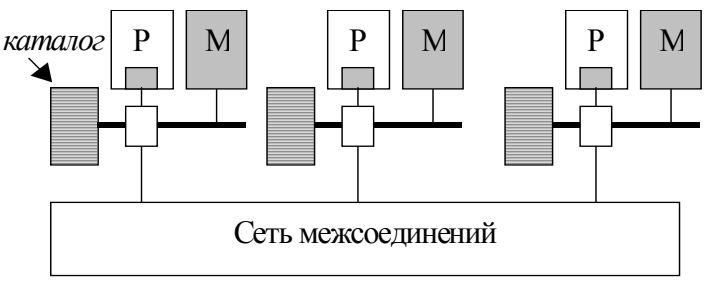
Обычно существует «сторожевой процесс» (демон), так называемый страничный сканер, который запускается каждые несколько секунд. Он должен следить за статистикой использования страниц и перемещать их таким образом, чтобы улучшить производительность. Было рассмотрено множество алгоритмов, но ни один из них не работает лучше других при лютых обстоятельствах.

6.3.6. Мультипроцессоры CC-NUMA

Мультипроцессоры, структура которых приведена на рис. 6.17 плохо расширяются, т.к. в них нет кэш-памяти. Добавление кэш-памяти порождает проблему совместимости кэшей. Один из вариантов – отслеживание системной шины. Технически это сделать несложно, но использование координатного переключателя ограничивает возможности расширения системы за счет резко возрастающих аппаратных затрат.

Самый популярный подход к построению больших мультипроцессоров CC-NUMA (Coherenet Cache NUMA – NUMA с согласованной кэш-памятью) – мультипроцессор на основе каталога. Основная идея состоит в сохранении базы данных, которая сообщает, где именно находится каждая строка кэш-памяти и каково ее состояние. При обращении к строке кэш-памяти из базы данных выявляется информация о том, где находится эта строка, и изменялась ли она. Поскольку обращение к базе данных происходит на каждой команде, которая обращается к памяти, то база данных должна находится в высокоскоростном специализированном программном обеспечении, которое способно выдавать информацию за долю цикла шины.

В качестве примера рассмотрим систему из 256 узлов, каждый из которых состоит из одного процессора и 16 Мбайт ОЗУ, связанного с процессором через локальную шину. Общий объем памяти – 2^{32} байт. Она разделена на 2^{26} строк кэш-памяти по 64 байта каждая. Память статически распределена по узлам: 0-16 М -- 0 узел, 16-32 – узел 1 и т.д. Узлы связаны через сеть (рис. 6.18). Сеть может быть реализована в виде решетки, гиперкуба или другой топологии. Каждый узел содержит элементы каталога для 2^{18} 64-битных строк кэш-памяти, составляя свою 2^{24}-битную память.



биты

Узел (8)	Блок (18)	Смещение (6)

Рис. 6.18

Проследим путь команды из процессора 20, который обращается к кэшированной строке. Процессор передает ее на в блок управления памятью, который переводит ее в

физический адрес, например 0x24000108. Блок управления разделяет этот адрес на три части. Получили: узел 36, строка 4, смещение 8. Слово находится в 36 узле, поэтому необходимо делать запрос через сеть, узнать есть ли строка 4 в кэш-памяти и где именно. Когда запрос прибывает в узел 36, он направляется в аппаратное обеспечение каталога. Аппаратное обеспечение индексирует таблицу в 2^{18} элементов (один элемент на строку) и извлекает элемент 4. Если строка отсутствует в кэш-памяти, то аппаратное обеспечение вызывает строку из локального ОЗУ, отправляет в узел 20 и обновляет элемент каталога 4, показывая, что эта строка размещена в кэш-памяти узла 20. Если строка присутствует в кэш-памяти, то аппаратное обеспечение обновляет элемент каталога, и объявляет элемент кэш-памяти в узле 36 недействительным.

Определим, какой объем памяти занимают каталоги. Каждый узел содержит 16 Мбайт ОЗУ и 2^{19} 9-битных элементов для слежения за этим ОЗУ. Таким образом непроизводственные затраты каталога составляют $9x2^{19}$ бит от 16 Мбайт или около 1,76%. Если расширить длину строки до 128 байт, то непроизводственные затраты снизятся до 1%.

Недостаток разработки в том, что строка может быть кэширована только в одном узле. Существуют различные варианты преодоления этого недостатка и все они связаны с дополнительными непроизводственными затратами памяти.

Мультипроцессор Stanford DASH

Первый мультипроцессор CC-NUMA на основе каталога DACH (Directory Architecture for Shared memory – архитектуры на основе каталога для памяти совместного использования) был разработан в Стенфордском университете как исследовательский проект. В ней применялся ряд промышленных изделий. Схема машины DACH в упрощенном варианте представлена на рис. 6.19. Она состоит из 16 кластеров, каждый из которых содержит шину, 4 процессора, 16 Мбайт глобальной памяти, а также некоторые устройства ввода/вывода (диски и т.п.), которые на схеме не показаны. Каждый процессор отслеживает только свою локальную шину. Локальная совместимость поддерживается с помощью отслеживания, для глобальной совместимости нужны иные механизмы, т.к. глобального отслеживания не существует.

Полный объем адресного пространства в данной системе равен 256 Мбайт. Адресное пространство разделено на 16 областей по 16 Мбайт каждая.

Каждый кластер содержит каталог, который следит за тем, какие кластеры в настоящий момент имеют копии своих строк. Поскольку каждый кластер содержит 1 М строк, то в каталоге содержится 1 М элементов, по одному на каждую строку. Каждый элемент содержит битовое отображение по одному биту на кластер, который показывает, имеется ли в данный

момент строка данного кластера в кэш-памяти. Кроме того, элемент содержит 2-х битное поле, которое сообщает о состоянии строки.

Получается, что объем каждого каталога превышает 2 Мбайта. При наличии 16 кластеров непроизводственные составляют 14 % от 256 Мбайт.

Если число процессоров на кластере возрастает, то это не приводит к увеличению объема памяти каталога. Большое число процессоров на кластер позволяет погашать стоимость памяти каталога, а также контроллера шины, сокращая стоимость на каждый процессор.

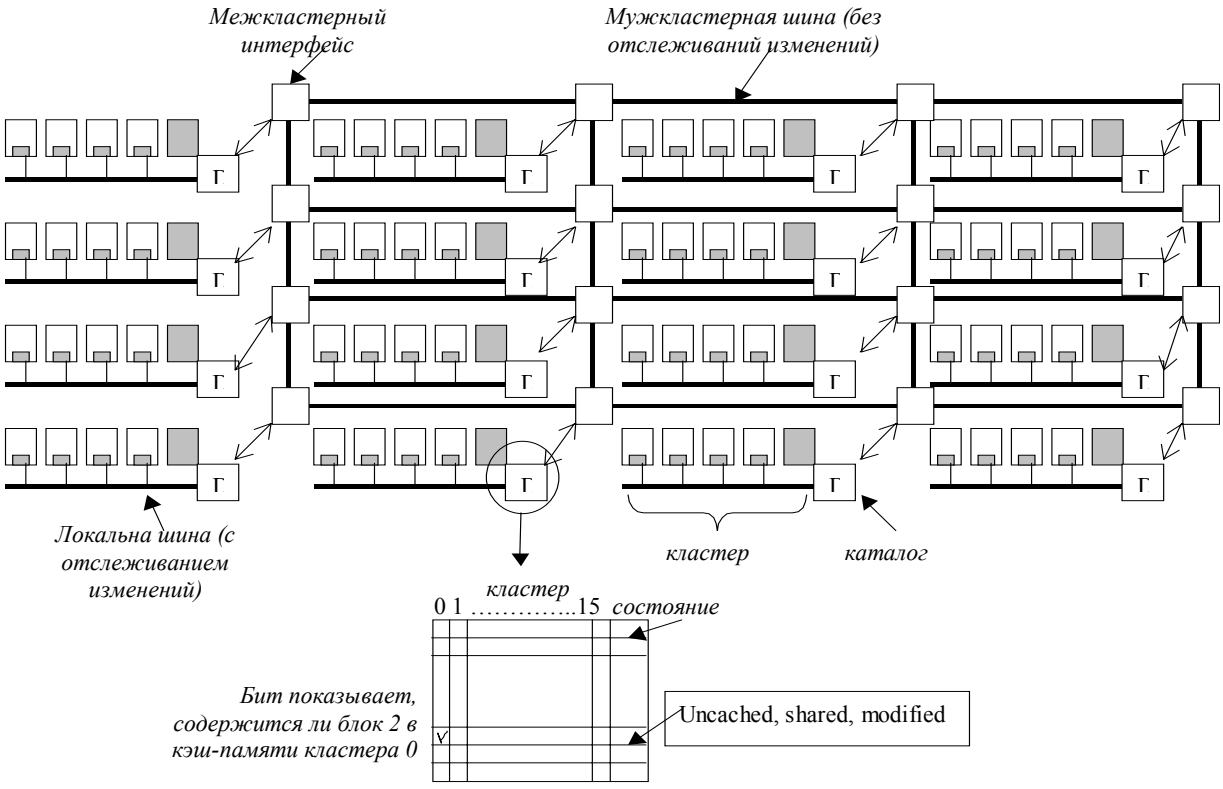


Рис. 6.19

Каждый кластер в DASH связан с интерфейсом, который дает возможность кластеру обмениваться информацией с другими кластерами. Интерфейсы связаны через межкластерные каналы в прямоугольную решетку. В системе использована маршрутизация «червоточина», поэтому первая часть пакета может быть направлена дальше еще до того, как получен весь пакет, что сокращает задержку на каждом транзитном участке. Существует два набора межкластерных пакетов: один для запрашиваемых, другой – для ответных. Межкластерные каналы не отслеживаются.

Каждая строка кэш-памяти может быть в одном из тех состояний:

1. UNCACHED (некэшированная) – строка находится в памяти.
2. SHARED (совместно используемая) – память содержит новейшие данные, строка может находится в нескольких блоках кэш-памяти.
3. MODIFIED (измененная) – строка, содержащаяся в памяти неправильная; данная строка находится только в одной кэш-памяти.

Состояние каждой строки кэш-памяти содержится в поле Состояние в соответствующем элементе каталога, как показано на рис. 6.19.

Сохранить согласованность памяти в системе DASH довольно трудно, и происходит это очень медленно. Для одного обращения в память иногда нужно отправлять несколько сообщений. Более того, чтобы память была согласована, доступ нельзя завершить, пока прием пакетов не будет подтвержден, а это снижает производительность. Для разрешения этих проблем в системе DASH (два набора межкластерных каналов, конвейеризация записи, использование свободной согласованности вместо согласованности по последовательности).

Мультипроцессор Sequent NUMA-Q

Машина DASH никогда не была коммерческим продуктом. Одно из коммерческих изделий машин этого класса -- Sequent NUMA-Q 2000. В ней используется интересный протокол когерентности кэширования *SCI (Scalable Coherent Interface* –масштабируемый когерентный интерфейс). Этот протокол стандартный (стандарт IEEE 1569) и используется в ряде других машин CC-NUMA.

В основе машины Sequent NUMA-Q лежит стандартная плата quard board производства Intel. Плата содержит 4 процессора Pentium-Pro и до 4 Гбайт ОЗУ. Каждый процессор содержит кэш-память первого и второго уровней. Непротиворечивость кэшей сохраняется благодаря отслеживанию локальной шины платы с использованием протокола MESI. Скорость передачи данных в локальной шине составляет 534 Мбайт/с. Размер строки кэш-памяти равен 64 байтам. Схема мультипроцессора Sequent NUMA-Q изображена на рис. 6.20.

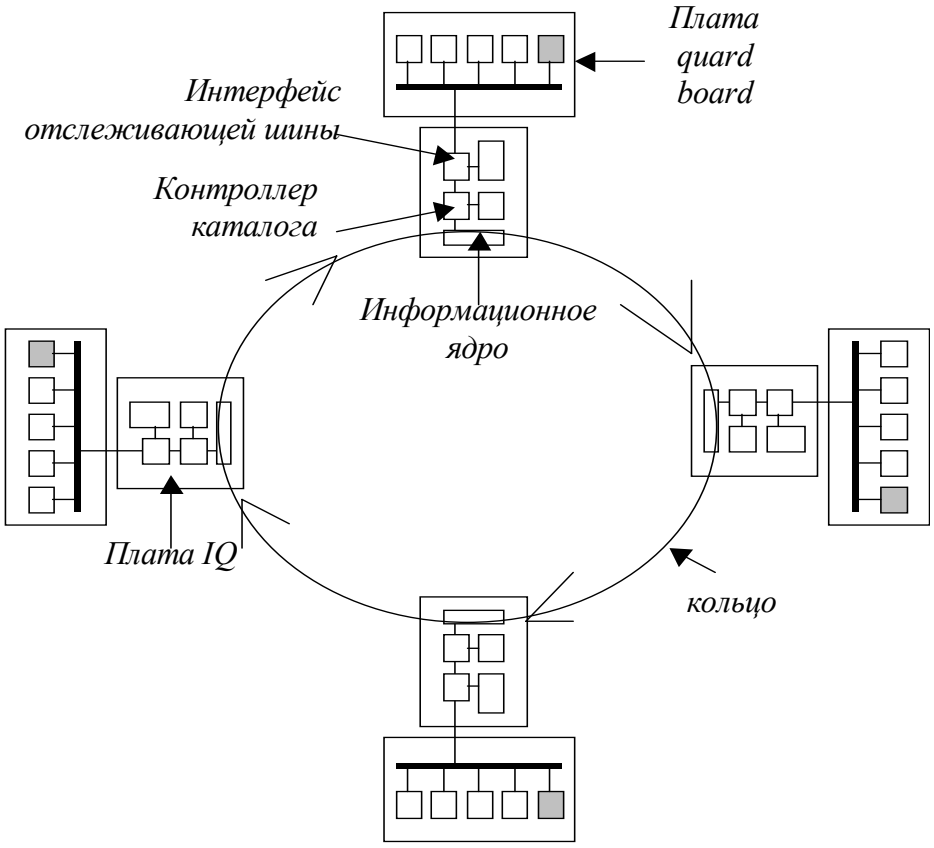


Рис. 6.20

Чтобы расширить систему надо вставить плату сетевого контроллера в гнездо сетевого контроллера на плате. Сетевой контроллер (плата IQ-Link) соединяет все платы в один мультипроцессор. Задача контроллера – реализовать протокол SCI. Каждая плата IQ-Link содержит 32 Мбайт кэш-памяти, каталог, который следит за там, что находится кэше, интерфейс с локальной шиной платы quard board и микросхему, называемую информационным ядром, соединяющим плату IQ-Link с другими платами IQ-Link. Эта микросхема подкачивает данные от входа к выходу, сохраняя те данные, которые направляются в ее узел и передавая прочие данные без изменений.

В этой разработке присутствуют два уровня протокола когерентности кэширования: протокол SCI поддерживает непротиворечивость всех кэшей, протокол MESI используется для сохранения непротиворечивости между четырьмя процессорами и кэш-памятью на 32 Мбайт в каждом узле.

Рассмотрим интерфейс SCI. Этот интерфейс был разработан для того, чтобы заменить шину в больших мультипроцессорах и мультикомпьютерах.

SCI поддерживает непротиворечивость кэшей, что важно для мультипроцессоров; и позволяет быстро передавать блоки, что актуально для мультикомпьютеров. SCI выдерживает нагрузку до 64 К узлов, адресное пространство каждого из которых может быть до 2^{48} байтов. Самая большая система NUMA-Q состоит из 63 плат, которые содержат 252 процессора и почти 2^{36} физической памяти.

Кольцо, соединяющее платы IQ-Link соответствует протоколу SCI и представляет собой не кольцо, а двухточечные кабели. (от платы к плате). Ширина кабеля – 18 бит (1 синхронизация, 1 флаг, 16 информационных). Тактовая частота – 500 Мгц, скорость передачи данных – 1 Гбайт/с. Передача осуществляется пакетами. Состав пакета:

- заголовок 14 байт;
- 0, 16, 64 или 256 байт данных;
- контрольная сумма 2 байта.

Физическая память в машине распределена по узлам, так что каждая страница памяти имеет свою собственную машину. Каждая плата quard board может иметь до 4 Гбайт ОЗУ. Размер строки кэш-памяти – 64 байта. Таким образом каждая плата содержит 2^{36} строк. Если строка не используется, она находится только в одном месте – в собственной памяти.

Для каждого узла существует *таблица локальной памяти* из 2^{36} элементов, по которой можно определить местоположение строк.

Все копии строки кэш-памяти собираются в дважды связанный список. Элемент в таблице локальной памяти показывает, в каком узле находится головная часть списка. В машине NUMA-Q 2000 достаточно 6-битного номера, поскольку может быть максимум 32 узла. Для системы SCI максимального размера достаточно 16-битного номера. Такая схема для больших систем лучше, чем битовое отображение. Поэтому SCI более расширяема, чем DASH.

Кроме таблицы локальной памяти каждая плата IQ-Link содержит каталог с одним элементом для каждой строки кэш-памяти. Поскольку размер кэша составляет 32 Мбайт, строка включает 64 байта, то каждая плата IQ-Link может содержать 2^{19} элементов. Если строка находится в одной кэш-памяти, то тот узел, в котором находится строка, указывается в таблице локальной памяти исходного узла. Если после этого данная строка появится в кэш-памяти другого узла, то в соответствии с новым протоколом исходный каталог будет указывать на новый элемент, который в свою очередь будет указывать на старый элемент. Таким образом, формируется двухэлементный список. Все новые узлы, содержащие эту строку, прибавляются в начало списка.

Каждый элемент каталога состоит из 36 бит. Шесть бит указывают на узел, который содержит предыдущую строчку цепочки. Следующие шесть – на узел, содержащий следующую строчку цепочки. 0 – конец цепочки, поэтому максимальный размер системы – 63 узла, а не 64. 7 бит предназначены для записи состояния строки. Последние 13 бит – тэг, который используется для идентификации строки. *(Замечание. Объем ОЗУ системы – 63 x $2^{32} \approx 2^{38}$, что соответствует 2^{32} строк кэш-памяти. 2^{32} строк отображаются на 2^{19} элементов кэш-памяти. Существует 2^{13} строк, отображаемые на каждый элемент. Следовательно, 13-битный тэг требуется для идентификации строки).*

Каждая строка имеет фиксированную позицию только в одном блоке памяти. Строки могут находится в одном из трех состояний:

- UNCACHED (некэшированная) – строка находится в памяти.
- SHARED (совместно используемая) – память содержит новейшие данные, строка может находится в нескольких блоках кэш-памяти.
- MODIFIED (измененная) – строка, содержащаяся в памяти неправильная; данная строка находится только в одной кэш-памяти.

В протоколе SCI определены 3 операции со списком: добавление узла к списку, удаление узла из списка, очистка всех узлов, кроме одного. Последняя операция нужна, если разделяемая строка изменена и становится единственной.

Протокол SCI имеет три варианта сложности. Протокол минимальной степени сложности разрешает иметь только одну копию каждой кэш-строки в памяти. В соответствии с протоколом средней степени сложности каждая строка может кэшироваться в неограниченном количестве узлов. Полный протокол различает особенности для увеличения производительности. Полностью протокол изложен в стандарте IEEE 1596.

6.3.7. Мультипроцессоры COMA

Машины NUMA и CC-NUMA имеют недостаток: обращения к удаленной памяти происходят гораздо медленнее, чем обращения к локальной памяти.

Машины UMA например, Sun Enterprise 10000 имеют очень высокую производительность, но ограничены в размерах и дорого стоят. Машины NUMA могут расширятся до больших размеров, но в таких системах трудно распределить страницы и предсказать, какие страницы понадобятся и страницы трудно перемещать из-за их значительных размеров. Это может привести к существенному сокращению производительности машины.

Существует процессор, в котором эти проблемы разрешаются за счет того, что основная память каждого процессора используется как кэш-память. Такая машина называется COMA (Cache Only Memory Access). При такой организации памяти страницы не имеют собственных фиксированных машин.

Вместо этого физическое адресное пространство делится на строки, которые перемещаются по системе в случае необходимости. Блоки памяти не имеют собственных машин. Память, которая привлекает строки по мере необходимости, называется *attraction memory*. Использование основной памяти в качестве большой кэш-памяти увеличивает частоту успешных обращений в кэш-память, а, следовательно, и производительность.

В результате появляются две новые проблемы:

1. как разместить строки кэш-памяти.
2. если строка удаляется из памяти, что произойдет, если это последняя копия.

Первая проблема возникает из-за того, что в такой системе строки не имеют постоянного местонахождения, а кочуют из машины в машину. Возникает необходимость постоянного отслеживания местонахождения

каждой строки, а в данной реализации это означает постоянный поиск по всей памяти. Для обеспечения приемлемого быстродействия необходимо разрабатывать новое аппаратное обеспечение.

Вторая проблема связана с удалением последней копии. Если происходит промах кэша, то строку надо откуда-то вызвать и какую-то удалить. А если удаляемая является последней копией? Тогда ее нужно каким-то образом сохранить. Таким образом, возникает необходимость учитывать и эту возможность.

Машина COMA обещает существенно увеличить производительности системы, но в настоящее время известно очень мало машин COMA (2 машины: KSR-1 и Data Diffusion Machine), и для объективной оценки недостаточно информации.

6.4. МУЛЬТИКОМПЬЮТЕРЫ С ПЕРЕДАЧЕЙ СООБЩЕНИЙ

Существует два типа параллельных процессоров MIMD: мультипроцессоры и мультикомпьютеры.

Из рассмотрения вариантов организации мультипроцессоров можно сделать вывод, что одной из основной проблем организации больших мультипроцессорных систем является использование общей памяти, что порождает проблемы согласования и контроля за использованием памяти.

Отличительная особенность мультикомпьютера в том, что каждый процессор имеет собственную память, в которую другие компьютеры не имеют прямого доступа. Программы на разных процессорах взаимодействуют друг с другом с помощью примитивов *send* и *receive*, которые используются для передачи сообщений. Это различие полностью меняет модель программирования.

Каждый узел в мультикомпьютере состоит из одного или нескольких процессоров, ОЗУ (общие для процессоров данного узла), диска и (или) других узлов ввода/вывода, а также процессора передачи данных. Процессоры передачи данных связаны между собой по высокоскоростной коммуникационной сети. Используется множество различных топологий, схем коммуникации и алгоритмов выбора маршрутов. Все мультикомпьютеры сходны в одном: когда программа выполняет примитив send, процессор передачи данных получает уведомление и передает блок данных в целевую машину (возможно, после предварительного запроса и получения разрешения). Схема мультипроцессора приведена на рис. 6.21.

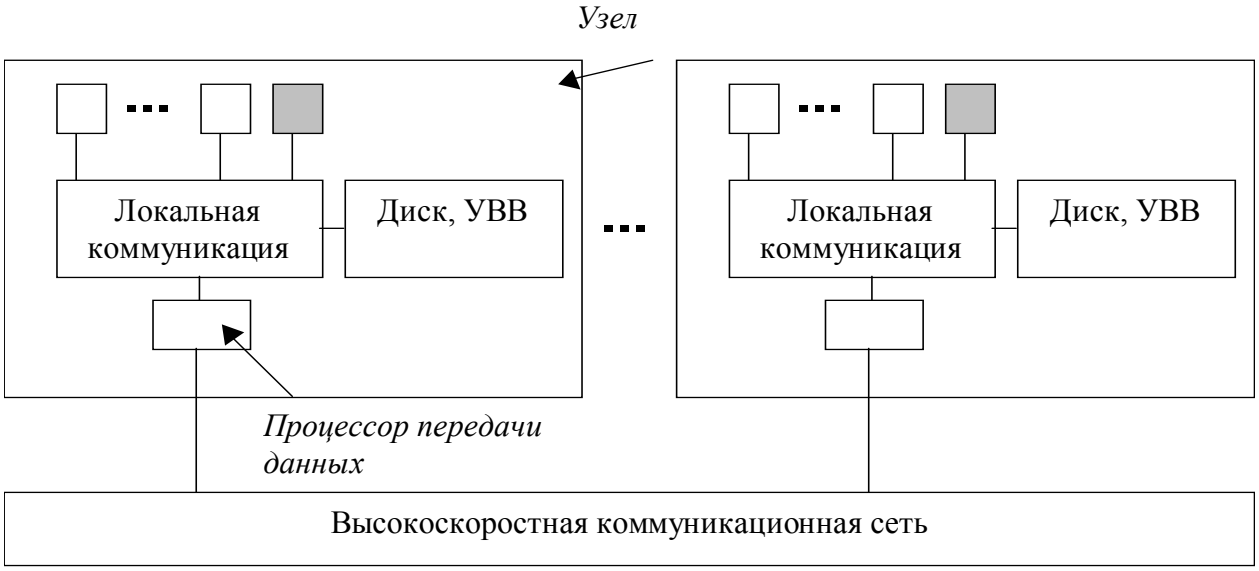


Рис. 6.21

6.4.1. MPP – процессоры с массовым параллелизмом

MPP (Massively Parallel Processor – процессоры с массовым параллелизмом) – это огромные компьютеры, стоимостью несколько миллионов долларов. Они используются в научных исследованиях и промышленности для выполнения сложных вычислений, для обработки большого числа тразакций, для хранения больших баз данных и управления ими.

В большинстве таких машин используются стандартные процессоры (Pentium, UltraSPARC, DEC Alpha и др.).

Особенности мультикомпьютеров:

- наличие высокопроизводительной сети передачи сообщений;
- высокая производительность процессов ввода-вывода (связана с необходиостью обработки огромных объемов данных);
- отказоустойчивость – система сохраняет работоспособность при отказе части процессоров;
- наличие специального аппаратного и программного обеспечения для контроля и диагностики системы, обнаружения и устранения неисправностей.

Crey T3E

В семейство T3E (последователя T3D) входят самые последние суперкомпьютеры. Различные модели – T3E, T3E-900 и T3E-1200 – идентичны с точки зрения архитектуры и различаются только ценой и производительностью (например, 600, 900 или 1200 мегафлопов на процессор). Мегафлоп – 1 млн. операций с п.з./с. (FLOP – Floating-point Operations – операции с плавающей точкой). В отличие от 6600 и Cray-1, в

которых очень мало параллелизма, эти машины могут содержать до 2048 процессоров.

В системе T3E используются процессоры DEC Alpha 21164. Это суперскалярный процессор RISC, способный выдавать 4 команды за цикл. Он работает с частотой 300, 450 и 600 Мгц в зависимости от модели. Тактовая частота – основное различие между моделями T3E. Процессор Alpha – 64-битная машина. Размер виртуальных адресов ограничен до 43 битов, а физических до 40. Таким образом, возможен доступ к 1 Тбайт физической памяти.

Каждый процессор Alpha имеет духуровневую кэш-память, встроенную в микросхему процессора. Кэш-память первого уровня содержит 8 Кбайт для команд и 8 Кбайт для данных. Кэш-память второго уровня смежная память на 96 Кбайт. Кэш-память обоих уровней содержит данные только из локального ОЗУ, которое может быть до 2 Гбайт на процессор. Таким образом, общий максимальный объем памяти 2048 x 2 = 4 Тбайт.

Каждый процессор Alpha заключен в особую схему, которая называется *оболочкой. (shell)*. Оболочка содержит память, процессор передачи данных, 512 регистров (т.н. E-регистров). Эти регистры могут загружаться адресами удаленной памяти с целью чтения или записи данных. Таким образом обеспечивается доступ к удаленной памяти, но он осуществляется не с помощью обычных команд записи/чтения. Непротиворечивость данных обеспечивается тем, что слова, считанные из удаленной памяти не попадают в кэш-память.

Узлы в машине T3E связаны двумя различными способами. Основная топология – дуплексный 3-мерный тор. Каждый узел в 3-мерном тор имеет 6 каналов связи. Скорость передачи данных в этих каналах 480 Мбайт/с в любом направлении.

Узлы также связаны одним или несколькими Giga Rings – подсистемами ввода/вывода коммуникационных пакетов, обладающими высокой пропускной способностью. Узлы используют эту систему для связи друг с другом, а также с сетями, дискам, периферийными устройствами. По ней посылают пакеты до 256 байт. Каждый Giga Rings состоит из пары колец шириной в 32 бита, которые соединяют узлы процессоров со специальными узлами ввода/вывода. Узлы ввода/вывода содержат специальные гнезда для сетевых карт, дисков и др. устройств.

Для повышения живучести системы в целом на каждые 128 пользовательских узлов содержится один запасной узел. Неисправный узел может быть заменен запасным во время работы системы без перезагрузки. Используемая операционная система – UNIX.

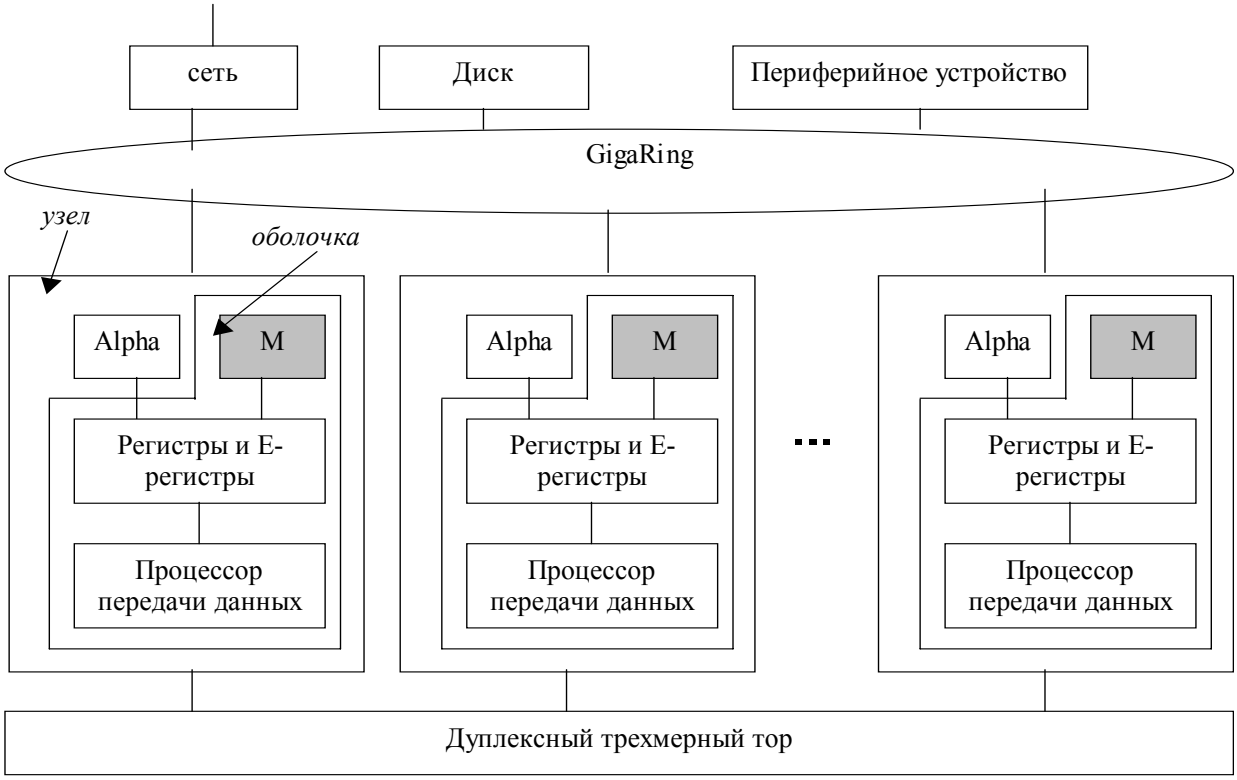


Рис. 6.22

Intel/Sandia Option Red

В середине 90-х годов департаменты обороны и энергетики США приступили к выполнению программы разработки 5 систем MPP, которые будут работать со скоростью 1, 3, 10, 30 и 100 терафлпов/с (терафлоп – 10^{14} операций с п.т./с).

В отличие от машины T3E, которую можно купить в магазине (за большие деньги), машины с такой производительностью не продаются. Эти машины предназначены для военных целей. Первые три машины получили названия Option Red (1996), Option Blue (1999), Option White. Будем рассматривать первую.

Option Red состоит из 4608 узлов, которые организованы в трехмерную сетку. Процессоры запакованы на платах двух разных типов. Платы *kestrel* используются в качестве вычислительных узлов, а платы *eagle* используются для сервисных, дисковых, сетевых узлов и узлов загрузки. Машина содержит 4536 вычислительных узлов, 32 сервисных узлов, 32 дисковых, 6 сетевых и 2 узла загрузки.

Плата *kestrel* (рис. 6.23) содержит 2 логических узла, каждый из которых включает 2 процессора Pentium Pro на 200 Мгц и разделенное ОЗУ на 64 Мбайт. Каждый узел *kestrel* содержит собственную 64-битную локальную шину и собственную микросхему *NIC (Network Interface Chip* – сетевой адаптер). Две микросхемы NIC соединены вместе, поэтому только одна из них подсоединена к сети, что делает систему более компактной. Платы *eagle* также содержат процессоры Pentium Pro, но всего 2 на плату.

Платы связаны в виде решетки 32 х 38 х 2 в виде двух взаимосвязанных плоскостей 32 х 38 (размер решетки продиктован целями компоновки, поэтому не во всех узлах решетки находятся платы). В каждом узле находится маршрутизатор с шестью каналами связи. Каждый канал связи может передавать информацию одновременно в обоих направлениях со скоростью 400 Мбайт/с.

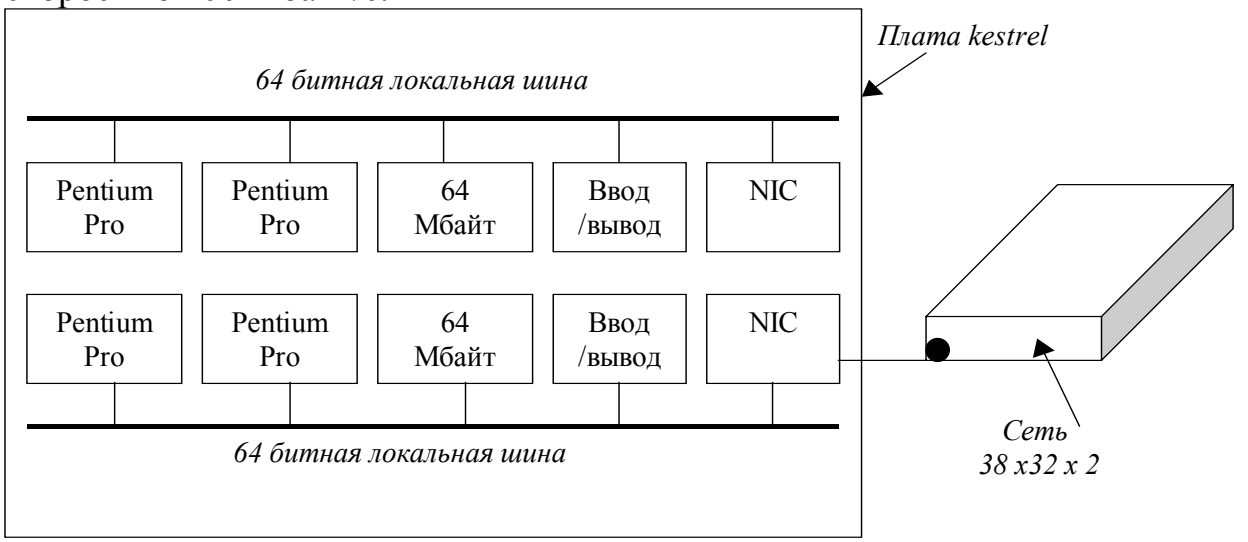


Рис. 6.23

Систему можно логически разделить на 4 части: сервис, вычисления, ввод/вывод и система. Сервисные узлы – это машины UNIX общего назначения с разделением времени, которые позволяют программистам писать и отлаживать свои программы. Вычислительные узлы запускают большие приложения. Они запускают не всю систему UNIX, а только микроядро, которое называется кугуаром (cougar – пума). Узлы ввода/вывода управляют 640 дисками, содержащими более 1 Тбайт данных. Есть два независимых набора узлов ввода-вывода. Узлы первого типа предназначены для секретной военной работы, а узлы второго типа – для несекретной. Эти два набора вводятся и удаляются из системы вручную, поэтому в каждый момент времени подсоединен только один набор узлов, что бы избежать утечки информации. Системные узлы используются для загрузки системы.

6.4.2. COW – Clusters of Workstation (кластеры рабочих станций)

Следующий тип мультикомпьютеров – системы COW или NOW (Network of Workstation – сеть рабочих станций). Обычно он состоит из нескольких сотен персональных компьютеров или рабочих станций, соединенных посредством сетевых плат. Различие между MPP и COW аналогично разнице между большой вычислительной машиной и персональным компьютером. Компоненты одинаковы, но различное быстродействие. Но они управляются и применяются по-разному.

Процессоры в MPP – это обычные процессоры, которые можно купить. Используются обычные динамические ОЗУ и ОС UNIX.

Исторически системы MPP отличались высокоскоростной сетью. Но с появлением коммерческих высокоскоростных серей это различие начало сглаживаться. В настоящее время существует высокоскоростная система COW, которая называется DAS (Distributed ASCII Supercomputer). Она состоит из 128 узлов, каждый из которых содержит процессор Pentium Pro yf 200 МГц и ОЗУ на 128 Мбайт (www.cs.vu.nl\~bal\das.html). Узлы организованы в 2-мерный тор. Скорость передачи информации по каналам – 160 Мбайт/с в обоих направлениях одновременно. Эти характеристики соизмеримы с характеристиками Option Red. Различие в том, что бюджет Option Red был значительно выше.

Преимущество систем COW над MPP в том, что COW полностью состоит из компонентов, которые можно купить и цены на которые постоянно падают.

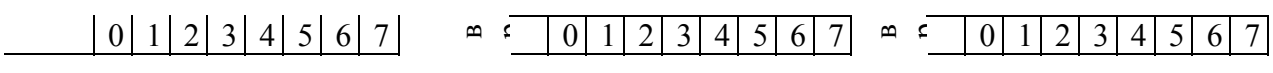
Существует множество различных видов COW, но доминируют два из них: централизованные и децентрализованные. Централизованные системы COW представляют собой кластер рабочих станций или персональных компьютеров, смонтированных в один блок в одном помещении. Иногда в таких системах применяется компоновка высокой плотности для сокращения длины кабелей. Как правило, такие машины не имеют развитой периферии за исключением сетевых карт. Такие системы еще называют *автономными рабочими станциями*.

Децентрализованная система COW состоит из рабочих станций или персональных компьютеров, которые территориально могут находиться в различных помещениях. Обычно они связаны через локальную сеть. Многие компьютеры имеют своих владельцев.

6.4.3. Планирование

Отличие децентрализованной системы COW от локальной сети связано с программным обеспечением и не связано с аппаратными средствами. В локальной сети пользователи работают с персональными машинами и используют их для своей работы. Децентрализованная система COW является общим ресурсом, которому пользователи могут поручить работу, требующую общих ресурсов. Чтобы система COW могла обрабатывать запросы от нескольких пользователей, каждому из которых нужно несколько процессоров, этой системе необходим планировщик заданий.

Рассмотрим самую простую систему планирования. Должно быть известно, сколько процессоров нужно для каждой работы (задачи). Тогда задачи выстраиваются в порядке FIFO (рис. 6.24 а). Когда первая задача начала выполнятся происходит проверка, есть ли достаточное количество процессоров для выполнения следующей задачи. Если достаточно, то она тоже принимается к выполнению. Если нет, то система ждет, пока не появится достаточное количество ресурсов.



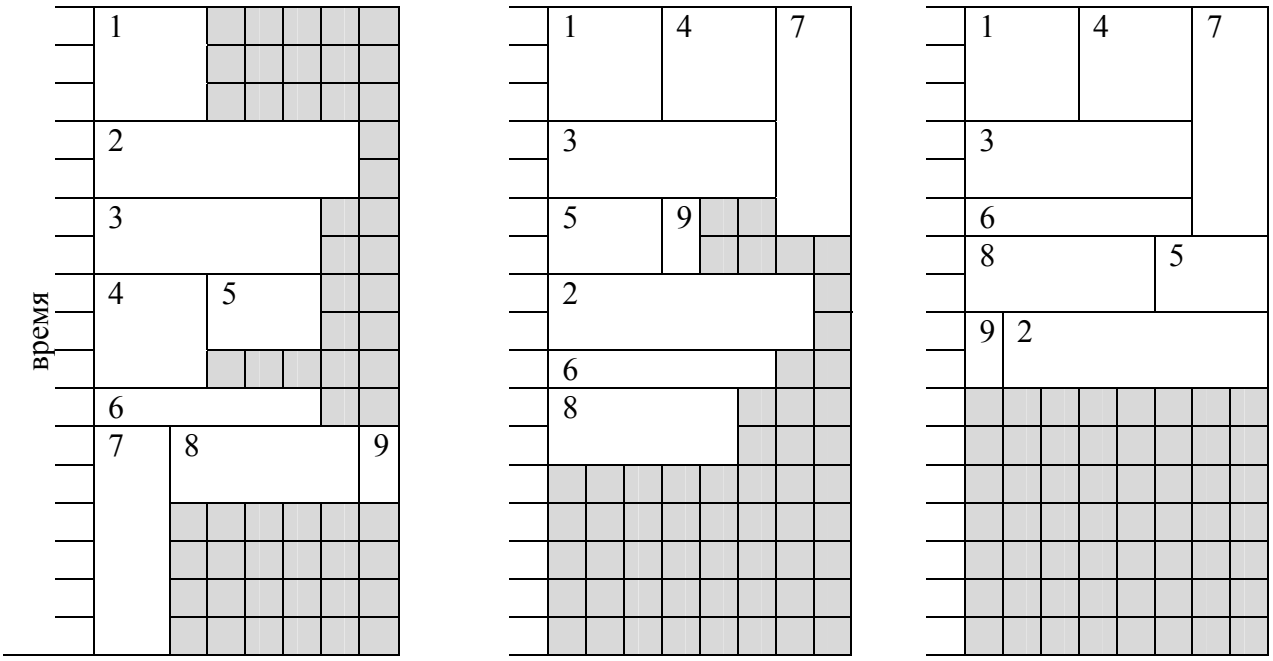


Рис. 6.24

В более сложном алгоритме, задачи, которые не могут быть приняты к выполнению пропускаются, и берется первая задача, для выполнения которой достаточно ресурсов. Всякий раз, когда задача завершается очередь просматривается с первой задачи (рис. 6.24 б).

Еще более сложный алгоритм требует не только знания необходимых ресурсов для каждой задачи, но и времени ее выполнения. Располагая такой информацией планировщик старается расположить задания наиболее эффективным способом (рис. 6.24 в).

Коммерческие сети межсоединений

Рассмотрим некоторые технологии связи.

Ethernet . Существует три версии этой системы*: classic Ethernet, fast Ethernet, gigabit Ethernet.* Они работают со скоростью 10, 100 и 1000 Мбит/с или 1,25, 12,5 и 125 Мбайт/с. (*Замечание. Эта скорость относится к тактовой частоте передачи символов. Учитывая протокол передачи сообщений, реальная скорость передачи информации составляет 800 – 900 Кбайт/с*).

Каждый компьютер в сети *Ethernet* содержит микросхему *Ethernet*, обычно на съемной плате. Схема подсоединения трех компьютеров приведена на рис. 8.25 а.

В соответствии с протоколом *Ethernet*, если машине нужно послать пакет, сначала она должна проверить, не совершает ли передачу в данный момент какая-либо другая машина. Если кабель свободен, машина просто посылает пакет. Если кабель занят, то машина ожидает окончания передачи и только после этого посылает пакет. Если две машины начинают передачу одновременно, происходит конфликтная ситуация. Разрешается она следующим образом. Обе машины определяют конфликтную ситуацию,

останавливают передачу, затем останавливаются на произвольный промежуток времени и повторяют попытку. Если конфликтная ситуация повторилась, они снова останавливаются и снова начинают передачу пакета, увеличив время среднее время ожидания в два раза с каждой конфликтной ситуацией.

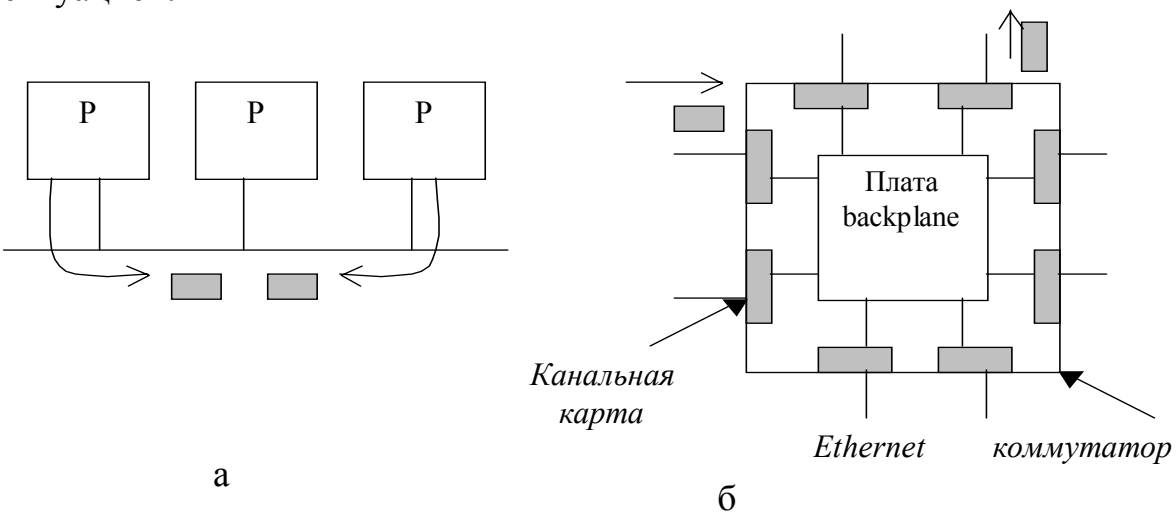


Рис. 6.25

Структура *Ethernet с использованием коммутатора* приведена на рис. 6.25 б. Здесь сетевой концентратор заменен устройством, содержащим высокоскоростную плату backplane, к которой можно подсоединять канальные карты. Каждая канальная карта принимает одну или несколько сетей *Ethernet* и разные карты могут воспринимать разные скорости, поэтому classic, fast и gigabit *Ethernet* могут быть связаны вместе. Когда пакет поступает в канальную карту, он временно сохраняется там в буфере, пока канальная карта не отправит запрос и не получит доступ к плате backplane, которая функционирует почти как шина. Если пакет был перемещен в канальную карту, к которой подключена целевая машина, он может направляться к этой машине. Если каждая канальная карта содержит только один *Ethernet* и этот *Ethernet* имеет только одну шину, то конфликтных ситуаций больше не возникнет, хотя пакет может быть потерян из-за переполнения буфера в канальной карте. *Gigabit Ethernet* с использованием коммутаторов и высокоскоростной платой Backplane имеет потенциальную производительность в 4 раза ниже, чем каналы связи в машине T3E, но стоит значительно дешевле (это по мнению разработчиков *Ethernet*). Однако при увеличении количества подключаемых машин вероятность возникновения конфликтных ситуаций возрастает.

Следующая технология связи ATM (Asynchronous Transfer Mode – асинхронный режим передачи). Технология ATM была разработана международным консорциумом телефонных компаний в качестве замены существующей телефонной системы на новую, полностью цифровую. Основная идея проекта состояла в том, чтбы каждый телефон и каждый компьютер в мире связать с помощью безошибочного цифрового битового канала со скоростью передачи данных 155 Мбит/с (потом появилась цифра

622 Мбит/с). Но практическая реализация вызвала сложности. Тем не менее многие компании сейчас выпускают съемные платы для персональных компьютеров с такой скоростью передачи данных. Вторая скорость подходит для мультикомпьютеров.

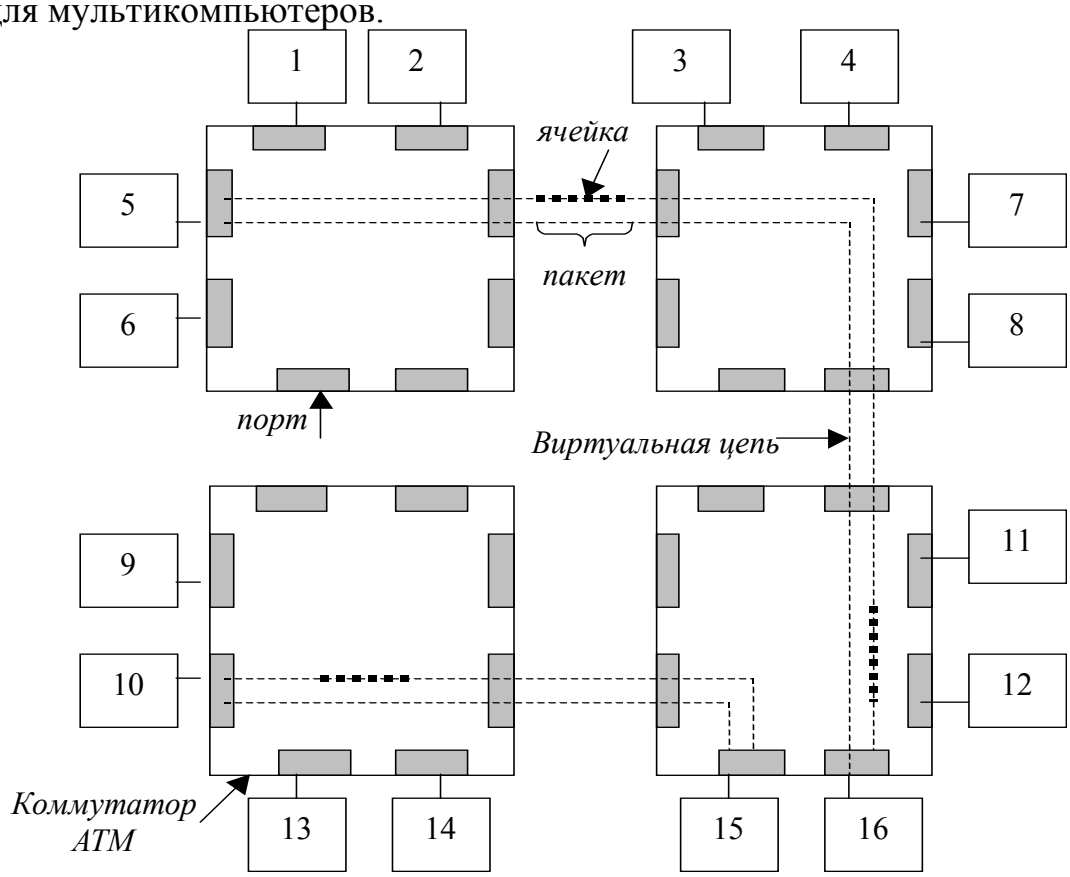


Рис. 6.26

Провод или стекловолокно, отходящее от плат ATM, перходит в переключатель ATM – устройство, похожее на коммутатор *Ethernet.* В него тоже поступают пакеты и сохраняются в буфере в канальных картах, затем поступают для передачи в пункт назначения. Различия между *Ethernet* и ATM следующие.

1. Поскольку ATM была разработана для замещения телефонной системы, то она представляет собой сеть с маршрутизацией информации. Перед отправлением пакета в пункт назначения исходная машина должна установить виртуальную цепь (рис. 6.26). В сети *Ethernet* такого нет. Пакеты, отправленные по виртуальной цепи всегда будут доставлены в правильном порядке, но буферы могут быть переполнены, поэтому доставка не гарантируется.
2. *Ethernet* может передавать целые пакеты до 1500 байт данных одним блоком. В ATM все пакеты разбиваются на ячейки по 53 байта. 5 байтов – поля заголовка, полезная нагрузка – 48 байт. Разбиение пакета и последующий сбор осуществляет аппаратное обеспечение.

Третий пример – сеть Myrinet – съемная плата, которая производится в Калифорнии и пользуется популярностью у разработчиков систем COW.

Здесь также используется модель со съемной платой, которая подсоединяется к коммутатор, а коммутаторы могут соединятся в любой топологии. Каналы сети дуплексные, скорость передачи информации 1,28 Гбит/с в обоих направлениях. Размер пакета не ограничен.

Myrinet пользуется популярностью у разработчиков систем COW, т.к. платы этой сети содержат программируемый процессор и большое ОЗУ. Хотя Myrinet появилась со своей операционной системой, многие исследовательские группы уже разработали свои собственные ОС. У них появились дополнительные функции и повысилась производительность.

6.4.4. Связное программное обеспечение для мультикомпьютеров

Для программирования мультикомпьютера требуется специальное программное обеспечение для обеспечения связей между процессорами и синхронизации.

В системах с передачей сообщений два и более процессор работают независимо друг от друга. Например, один из процессов может производить какие-то данные, а другой может потреблять их. Если у отправителя есть данные, нет никакой гарантии, что получатель готов их принять. В большинстве систем с передачей сообщений имеется два примитива *send* и *receive*. Три основных варианта:

1. Синхронная передача сообщений.
2. Буферная передача сообщений.
3. Неблокируемая передача сообщений.

Синхронная передача сообщений. Если отправитель выполняет операцию *send*, а получатель не еще выполнил операцию *receive*, то отправитель блокируется до тех пор, пока получатель не выполнит операцию receive, а в это время сообщение копируется. Когда к отправителю возвращается управление, он уже знает, что сообщение отправлено и получено. Этот метод имеет простую семантику и не требует буферизации. Недостаток: блокировка отправителя.

Буферная передача сообщений. Если сообщение посылается до того, как получатель готов его принять, это сообщение временно сохраняется в буфере (например, почтовый ящик), и хранится там до тех пор, пока получатель не возьмет его оттуда. Такая схема сокращает время ожидания. Однако отсутствуют гарантии, что сообщение получено.

Неблокируемая передача сообщений. Отправитель может продолжать работу сразу после вызова. Библиотека сообщает ОС, что она сделает вызов позднее, когда у нее будет время. Отправитель вообще не блокируется. Недостаток: отправитель после выполнения операции *send* не может снова использовать буфер сообщений, т.к. не исключена возможность, что сообщение еще не отправлено. Отправитель должен каким-то образом определять, когда он может использовать буфер. Все это усложняет ПО.

PVM – виртуальная машина параллельного действия

PVM(Parallel Virtual Machine – виртуальная машина параллельного действия) система с передачей сообщений, изначально разработана для машин COW с системой UNIX. Позднее она стала применяться в других машинах, в том числе и в машинах MPP. Это самодостаточная система с управлением процессами и системой ввода-вывода. PVM состоит из двух частей: библиотеки, вызываемой пользователем и «сторожевого» процесса, который работает постоянно на каждой машине в мультикомпьютере. Когда PVM начинает работу, она определяет, какие машины должны быть часть ее виртуального мультикомпьютера. Для этого она читает конфигурационный файл. «Сторожевой» процесс запускается на каждой из этих машин. Машины можно добавлять и убирать, вводя команды с консоли PVM.Взаимодействие в машине PVM осуществляется с помощью примитивов для передачи сообщений таким образом, чтобы взаимодействовать могли машины с разными системами счисления. Каждый процесс PVM в каждый момент времени имеет один активный пересылочный буфер и один приемный буфер. Отправляя сообщение процесс вызывает библиотечные процедуры, запаковывающие значения с самоописанием в активный пересылочный буфер, чтобы получатель мог узнать их и преобразовать в исходный формат. Когда сообщение скомпоновано, отправитель вызывает библиотечную процедуру *pmv_send* , которая представляет собой блокирующий сигнал *send.* Получатель может ответить по-разному.

Во-первых, он может вызвать процедуру *pmv_recv*, которая блокирует получателя до тех пор, пока не придет подходящее сообщение. Когда вызов возвратится, сообщение будет в активном приемном буфере. Оно может быть распаковано и преобразовано в подходящий формат с помощью набора распаковывающих процедур.

Во-вторых, получатель может вызвать процедуру pmv_trecv, которая блоктрует получателя на определенный промежуток времени, и если подходящего сообщения за это время не пришло, он разблокируется.

Третий вариант – процедура pmv_nrecv, которая сразу же возвращает значение – это может быть либо сообщение, либо указание на отсутствие сообщения. Вызов можно повторять, чтобы опрашивать входящие сообщения.

Помимо рассмотренных примитивов, PVM поддерживает широковещание (процедура pmv_bcast) и мультивещание (процедура pmv_mcast). Первая процедура отправляет сообщение всем процессам в группе, а вторая посылает сообщение только некоторым процессам, входящих в определенный список.

Синхронизация между процессами осуществляется с помощью процедуры pmv_barrier. Когда процесс вызывает эту процедуру, он блокируется до тех пор, пока определенное число других процессор не достигнет барьера, и они не вызовут ту же процедуру. Существуют другие процедуры для управления главной вычислительной машиной, группой, буферами, для передачи сигналов, проверки состояния и т.д. PVM – это

простой, легкий в применении пакет, имеющийся в наличии в большинстве компьютеров параллельного действия, что и объясняет его популярность.

MPI – интерфейс передачи сообщений

MPI (Message-Passing Interface – интерфейс передачи сообщений) – гораздо сложней, чем PVM. Он содержит намного больше библиотечных вызовов м много больше параметров на каждый вызов. Первая версия MPI, которая сейчас называется MPI-1 была дополнена второй версией, MPI-2, в 1997 году.

MPI-1, в отличие от PVM никак не связана с созданием процессов и управления процессами. Создавать процессы должен сам пользователь с помощью локальных системных вызовов. После создания процессы организуются в группы, которые уже не изменяются. С этими группами работает MPI.

В основе MPI лежит 4 понятия: коммуникаторы, типы передаваемых данных, операции коммуникации и виртуальные топологии. Коммуникатор – это группа процессов плюс контекст. Контекст – это метка, которая идентифицирует что-либо (например, фазу выполнения). В процессе отправки и получения сообщения контекст может быть использован для того, чтобы несвязанные сообщения не мешали друг другу. Сообщения могут быть разных типов: символьные, целочисленные, с обычной или удвоенной точностью и т.д. Можно образовывать новые типы сообщений из уже существующих.

MPI поддерживает 4 основных типа коммуникации. Первый тип синхронный. В нем отправитель не может начать передачу данных, пока получатель не вызовет процедуру *MPI_Recv.* Второй тип – коммуникация с использованием буфера. Здесь отсутствуют ограничения для первого типа. Третий тип стандартный. Он зависит от реализации и может быть либо синхронным, либо с буфером. Четвертый тип сходен с первым. Здесь отправитель требует, чтобы получатель был доступен, но без проверки. Каждый из этих примитивов бывает двух видов: блокирующим и неблокирующим, что обеспечивает в общей сложности 8 примитивов. Получение может быть только двух типов – блокирующим и неблокирующим.

MPI поддерживает коллективную коммуникацию – широковещание, распределение и сбор данных, обмен данными, агрегацию и барьер. При любых формах коллективной коммуникации все процессы в группе должны делать вызов, причем с совместимыми параметрами.

Четвертое основное понятие в MPI – виртуальная топология, когда процессы могут быть организованы в дерево, кольцо, решетку и т.д. Такая организация процессор обеспечивает способ наименования каналов связи и облегчает коммуникацию.

В MPI-2 были добавлены динамические процессы, доступ к удаленной памяти, неблокирующая коллективная коммуникация, расширяемая поддержка процессов ввода-вывода, обработка в PMB и другое.

В научном сообществе идет война между лагерями MPI и PVM. Сторонники PVM утверждают, что эту систему легче изучать и поддерживать. Сторонники MPI утверждают, что их система поддерживает больше функций и, кроме того, она стандартная, что подтверждается документами.

6.4.5. Совместно используемая память на прикладном уровне

Из рассмотренных ранее вариантов построения больших систем можно сделать вывод, что мультикомпьютеры можно расширять до гораздо больших размеров, чем мультипроцессоры. Максимальное число процессоров 64 (Sun Enterprise 1000) или 256 (NUMA-Q). А мультикомпьютер Option Red содержит 9416 процессоров.

Однако, мультикомпьютеры не имеют совместно используемой памяти на архитектурном уровне. Это привело к появлению таких систем с передачей сообщений как PVM и MPI. Большинство программистов предпочитают иллюзию совместно используемой памяти, даже если ее не существует.

Многие исследователи пришли к выводу, что общая память на архитектурном уровне может быть нерасширяемой, но существуют другие способы достижения иллюзии наличия совместно используемой памяти. Совместно используемая память может существовать и на других уровнях системы. Рассмотрим, как это может быть реализовано.

Распределенная совместно используемая память

Один из классов систем с общей памятью на прикладном уровне – это системы со страничной организацией памяти. Такая система называется DSM (Distributed Shared Memory – распределенная совместно используемая память).

В этом мультикомпьютере ряд процессоров разделяет общее виртуальное адресное пространство со страничной организацией. Самый простой вариант – каждая страница содержится в ОЗУ ровно для одного процессора. Когда процессор обращается к странице на своем ОЗУ, чтение и запись происходят без задержки. При обращении к странице на другом ОЗУ, происходит ошибка из-за отсутствия страницы. В отличие от однопроцессорных систем, отсутствующая страница берется не с диска. ОС посылает сообщение в узел, в котором находится данная страница, чтобы преобразовать ее и отправить к процессору. После получения она преобразовывается в исходное состояние, а приостановленная команда выполняется заново.

Впервые идея была реализована в машине IVY. В результате в мультикомпьютере появилась память совместного использования, согласованная по последовательности. В целях улучшения производительности возможны оптимизации. Первая оптимизация, появившаяся в IVY, -- страницы, предназначенные только для чтения, могли присутствовать в разных ОЗУ.

Но даже при такой оптимизации трудно достичь высокой производительности, т.к. в случае записи информации в одну и ту же страницу разными процессорами, страница должна постоянно перемещаться от процессору к процессору. Такая ситуация называется *ложным совместным использованием*.

Проблему ложного совместного использования можно разрешить различными способами.

Например, можно отказаться от согласованности по последовательности в пользу свободной согласованности. Потенциально записываемые страницы могут присутствовать в нескольких узлах одновременно, но перед записью процесс должен совершить операцию *acquire*, чтобы сообщить о своем намерении. В этот момент все копии, кроме последней, объявляются недействительными. Нельзя делать никаких копий до тех пор, пока не будет выполнена операция *release* и после этого страница снова станет общей.

Второй способ оптимизации – изначально преобразовать каждую записываемую страницу в режим «только для чтения». Когда в страницу запись производится впервые, система создает копию страницы, называемую *двойником*. Затем страница преобразовывается в формат «для чтения и записи», и последующие записи могут производиться на полной скорости. Если потом возникнет необходимость доставки страницы к другому процессору, между текущей страницей и двойником происходит пословное сравнение. Пересылаются только те слова, которые были изменены, что сокращает объем пересылаемой информации.

При возникновении ошибки из-за отсутствия страницы, необходимо определить ее местонахождение. Используются различные способы, в том числе и каталоги, как в машинах NUMA и COMA.

Фактически, система DSM представляет собой программную реализацию машин NUMA или COMA, в которой каждая страница трактуется как строка кэш-памяти.

Linda

Системы DSM со страничной организацией памяти используют блок управления памятью, для того, чтобы закрывать доступ к отсутствующим страницам. Хотя пересылка только отдельных слов вместо всей страницы и влияет на производительность системы, страницы не удобны для совместного использования, поэтому применяются иные подходы.

Один из таких подходов реализован в системе Linda. Здесь процессы на нескольких машинах снабжены структурированной распределенной памятью

совместного использования. Доступ к такой памяти осуществляется с помощью небольшого набора примитивных операций, которые можно добавить к существующим языкам (например, C), в результате чего получаются параллельные языки программирования, например C-Linda.

В основе системы Linda лежит понятие абстрактного *пространства кортежей*, которое глобально по отношению ко всей системе и доступно всем процессам этой системы. Пространство кортежей похоже на глобальную память совместного использования, только с определенной внутренней структурой. Пространство содержит ряд кортежей, каждый из которых состоит из одного или нескольких полей. Для C- Linda поля могут содержать целые числа, длинные целые, в плавающей точкой, а также массивы и структуры (но не другие кортежи.

Над кортежами можно совершать 4 операции:

- Out – помещает кортеж в пространство кортежей;
- In – извлекает кортеж из пространства кортежей (с удалением);
- Read – извлекает без удаления;
- Eval – параллельно оценивает свои параметры, а полученный в результате кортеж копируется в пространство кортежей.

Основной принцип программирования в системе Linda – модель *replicated worker*. В основе этой модели лежит понятие пакета задач (*task bag*), которые нужно выполнить. Идея состоит в том, что задачи выдаются в пространство кортежей. Каждый рабочий процесс начинает работу с получения кортежа с описанием задачи. После выполнения задачи процесс получает следующую задачу. Таким образом, работа динамически распределяется среди рабочих процессов и каждый рабочий процесс занят постоянно.

Orca

Несколько другой подход к совместно используемой памяти на прикладном уровне в мультикомпьютере – в качестве общей совместно используемой единицы памяти использовать полные объекты, а не просто кортежи. Объекты состоят из внутреннего (скрытого) состояния и процедур для оперирования этим состоянием. Поскольку программист не имеет прямого доступа к состоянию, появляется возможность совместного использования ресурсов машинами, которые не имеют общей физической памяти.

Одна из таких систем называется Orca. Orca—это традиционный язык программирования (основанный на Modula 2), к которому добавлены две особенности: объекты и возможность создавать новые процессы. Объекты – это стандартный тип данных. Объект заключает в себе внутренние структуры данных и написанные пользователем процедуры, которые называются *операциями*. Объекты пассивны, т.е. они не содержат потоков, которым можно посылать сообщения. Процессы получают доступ к внутренним данным объекта путем вызова их процедур.

В системе *Orca* данные совместного использования совмещаются с синхронизацией не так, как в системах со страничной организацией памяти. В программах с параллельной обработкой нужны два вида синхронизации. Первый вид – взаимное исключение. Этот метод не позволяет двум процессам одновременно выполнять одну и ту же критическую область. В системе *Orca* каждая операция над объектом похожа на критическую область, поскольку система гарантирует, что конечный результат будет таким же, как если бы все критические области выполнялись последовательно одна за другой. В этом отношении объект *Orca* похож на распределенное контролирующее устройство.

Второй вид синхронизации – условная синхронизация, при которой процесс блокируется и ждет выполнения определенного условия. В системе Orca условная синхронизация осуществляется при помощи предохранителей.

В системе Orca допускается копирование объектов, миграция и вызов операций. Каждый объект может находиться в одном из двух состояний: он может быть единственным, а может быть продублирован. В первом случае объект существует только на одной машине и все запросы адресуются туда. Продублированный объект присутствует на всех машинах, которые содержат процесс, использующий этот объект. Это упрощает операцию чтения (ее можно производить локально), но усложняет процесс обновления. При выполнении операции, которая изменяет продублированный объект, сначала нужно получить от центрального процессора порядковый номер. Затем в каждую машину, содержащую копию объекта, отправляются сообщения о необходимости выполнить эту операцию. Поскольку все такие обновления обладают порядковыми номерами, все машины просто выполняют операции в порядке номеров, что гарантирует согласованность по последовательности.

Globe

Большинство систем DSM, Linda, Orca работают в пределах одного здания или предприятия. Однако можно построить систему с совместно используемой памятью на прикладном уровне, которая может распространяться на весь мир. В системах Globe объект может находиться в адресном пространстве нескольких процессоров одновременно, может быть даже на разных континентах. Чтобы получить доступ к данным общего объекта, пользовательские процессы должны пройти через его процедуры, поэтому для разных объектов возможны различные способы реализации. Например, можно иметь один экземпляр данных, который запрашивается по мере необходимости (этот вариант удобен для данных, часто обновляемых одним владельцем). Другой вариант – все данные находятся в каждой копии объекта, а сигнал об обновлении посылаются каждой копии в соответствии с надежным протоколом широковещания.

Цель системы Globe – работать на миллиард пользователей и содержать триллион объектов – делает эту систему амбициозной. Ключевым моментом является размещение объектов, управление ими, а также расширение

системы. Система Globe содержит общую сеть, в которой каждый объект может иметь собственную стратегию дублирования, стратегию защиты и т.д.

Среди других широкомасштабных систем можно назвать Globus и Legion, но они, в отличие от Globe, не создают иллюзию совместного использования памяти.

7. СУПЕРКОМПЬТЕР СКИФ

7.1. ПРИНЦИПЫ ПОСТРОЕНИЯ СУПЕРКОМПЬЮТЕРОВ СЕМЕЙСТВА СКИФ

Основополагающими архитектурными принципами создания суперкомпьютерных конфигураций СКИФ являются:

- **Базовая кластерная архитектура;**
- **Иерархичные кластерные конфигурации (метакластеры);**
- **Универсальная двухуровневая архитектура.**

7.1.1. Базовая кластерная архитектура

Концепция создания моделей семейства суперкомпьютеров СКИФ базируется на масштабируемой кластерной архитектуре, реализуемой на *классических кластерах из вычислительных узлов (рис. 7.1) на основе компонентов широкого применения*.

Кластерный архитектурный уровень – это тесно связанная сеть (кластер) вычислительных узлов, работающих под управлением ОС Linux.

Для организации параллельного выполнения прикладных задач на данном уровне используется:

- Оригинальная система поддержки параллельных вычислений Т-система, реализующая автоматическое динамическое распараллеливание программ;
- Классические системы поддержки параллельных вычислений:MPI, PVM и др. В семействе суперкомпьютеров СКИФ в качестве базовой классической системы параллельных вычислений выбран MPI, но не исключается использование других средств.

На кластерном уровне с использованием Т-системы и MPI эффективно реализуются фрагменты со сложной логикой вычисления и с крупноблочным (явным статическим или динамическим) параллелизмом. Фрагменты же с простой логикой выполнения, конвейерным или суперскалярным параллелизмом реализуются менее эффективно.

Кластерная архитектура является открытой и масштабируемой, т.е. не накладывают жестких ограничений к программно-аппаратной платформе узлов кластера, топологии вычислительной сети и конфигурации.

Для организации взаимодействия вычислительных узлов суперкомпьютера в его составе используются различные сетевые (аппаратные и программные средства), в совокупности оьразующие две системы передачи данных:

- *Системная сеть кластера (СС),* или *System Area Network (SAN)*, объединяет узлы кластерного уровня. Данная система поддерживает масштабируемость кластерного уровня, а также пересылку когерентных данных во всех вычислительных узлах кластерного уровня суперкомпьютера. Системная сеть строится на основе специализированных высокоскоростных линков класса SCI, Myrinet и др., предназначенных для эффективной поддержки кластерных вычислений и соответствующей программной поддержки на уровне ОС Linux и систем организации параллельных вычислений (T-система, MPI).
- *Вспомогательная сеть (ВС) с протоколами TCP/IP* объединяет узлы кластерного уровня в обычную локальную сеть. Сеть может быть реализована на основе широко используемых сетевых технологий класса Fast Ethernet, Gigabit Ethernet и др. Данная сеть предназначена для управления системой, подключения рабочих мест пользователей, интеграции суперкомпьютера локальную сеть предприятия или глобальные сети.

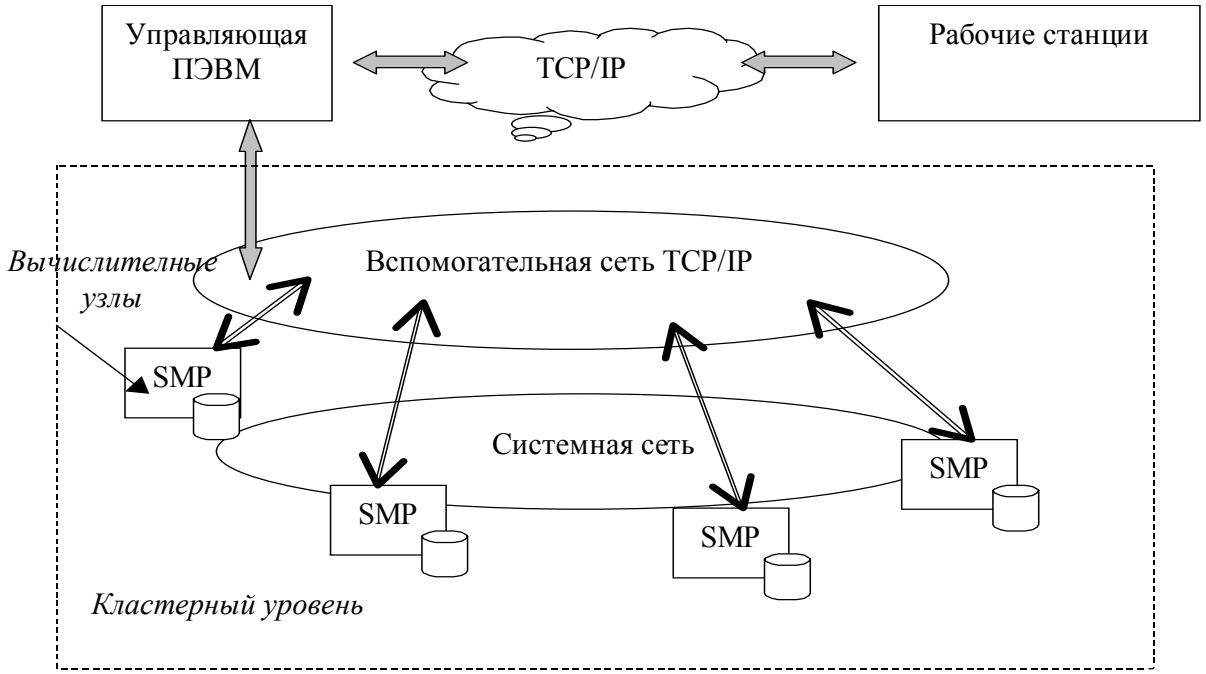


Рис. 7.1

Кластерные конфигурации на базе вспомогательной сети TCP/IP без использования дорогостоящих высокоскоростных линков могут быть реализованы в виде изделий *TCP/IP кластеры.* Такой подход расширяет область применения системы суперкомпьютеров СКИФ. Кластерные конфигурации на базе только вспомогательной сети могут быть реализованы как на базовых конструктивах СКИФ, так и путем

кластеризации имеющихся у пользователей ПЭВМ (персональные кластеры или супер ПЭВМ).

7.1.2. Базовое (системное) программное обеспечение

В качестве базовой ос используется операционная система Linux.

7.1.3. Иерархические кластерные конфигурации (метакластеры)

Отдельные кластеры могут быть объединены в единую кластерную конфигурацию – кластер высшего уровня *метакластер.* Метакластерный принцип позволяет создавать распределенные метакластерные конфигурации на базе локальных и глобальных сетей передачи данных.

Системное программное обеспечение метакластера обеспечивает возможность реализации *гетрогенных* систем, включающих подкластеры различной архитектуры на различных программно-аппаратных платформах. Рекомендуемый интерфейс передачи сообщений – *MPI.*

7.1.4. Универсальная двухуровневая архитектура

Концепция предусматривает возможность реализации универсальная двухуровневой архитектуры, приведенной на рис. 7.2

Концепция предусматривает реализацию потокового архитектурного уровня как на базе однородной вычислительной среды (ОВР) с использованием оригинальных СБИС ОВС, так и на базе других структурных и технических решений (XILINX, ALTERA и др.) По сути, вычислительные модули потокового уровня являются сопроцессорами вычислительных ресурсов кластерной конфигурации.

7.1.5. Особенности архитектуры семейства суперкомпьютеров СКИФ

Т-системы – обеспечивается автоматическое динамическое распараллеливание программ, что освобождает программиста от большинства трудоемких аспектов разработки параллельных, свойственных различным системам ручного статического распараллеливания:

- Обнаружение готовых к выполнению фрагментов задачи (процессов);
- Распределение по процессорам;
- Синхронизацию по данным.

По сравнению с использованием распараллеливающих компиляторов, Т-система обеспечивает более глубокий уровень параллелизма во врея выполнения программы и более полное использование вычислительных

ресурсов мультипроцессора. Принципиальная особенность Т-системы – это динамическое распараллеливание, в то время как распараллеливающий компилятор выполняет статическое распараллеливание.

В реализации Т-системы используются следующие технологические находки, не имеющие аналогов в мире:

- Реализация понятия «неготовое значение» и поддержка корректного выполнения некоторых операций над неготовыми значениями.
- Оригинальный алгоритм динамического автоматического распределения процессов по процессорам. По сравнению с известными алгоритмами, алгоритм Т-системы имеет существенно более низкий трафик межпроцессорных передач.

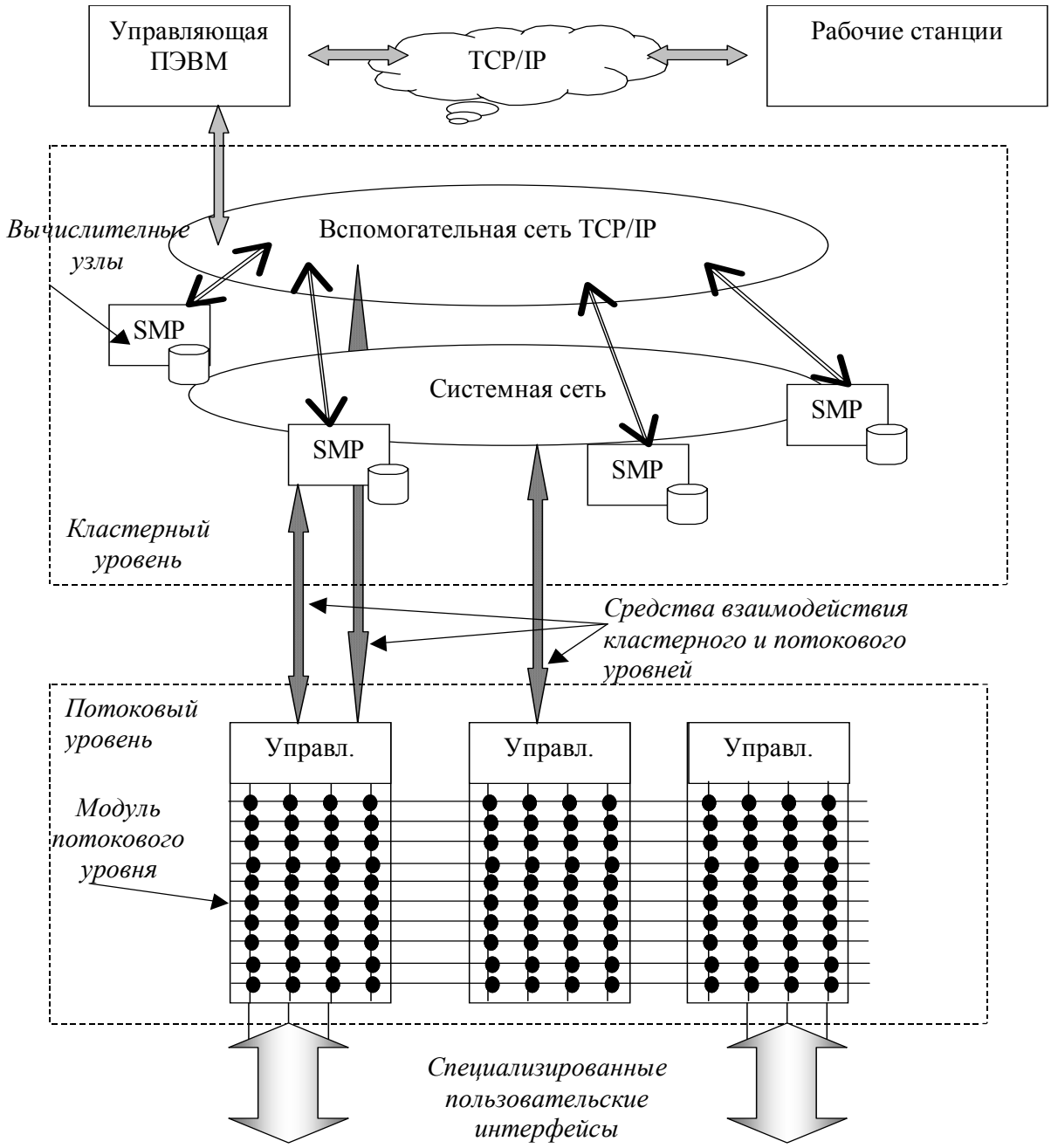


Рис. 7.1

- Архитектура вычислительных модулей потокового уровня позволяет использовать естественный параллелизм решаемой задачи. Фактически, при решении конкретной функции или самостоятельной задачи используется спецпроцессор, реализующую решаемую функцию или задачу. На матрице модулей потокового управления могут параллельно решаться несколько независимых задач. Вычислительные модули потокового уровня позволяют создавать системы с высоким уровнем надежности.

7.2. МОДЕЛИ СУПЕРКОМПЬЮТЕРОВ СКИФ РЯДА 2

7.3.1. Суперкомпьютер СКИФ К-500

Кластер К-500 создан в 2003 году для отработки принципов построения моделей суперкомпьютеров.

Технические характеристики:

Предельная пиковая (реальная на задаче Linpack) производительность	716 (475,3) Гфлопс
Тип процессора	Intel Xeon 2,8 ГГц
Число вычислительных узлов/процессоров	64/128
Оперативная память узла/установки	64 x 2 =128 Гб
Дисковая память установки	64 x 60 = 3840
Тип системной сети	3D-тор, SCI, D336
Тип вспомогательной сети	GB Ethernet

Структурная схема приведена на рис. 7.3.

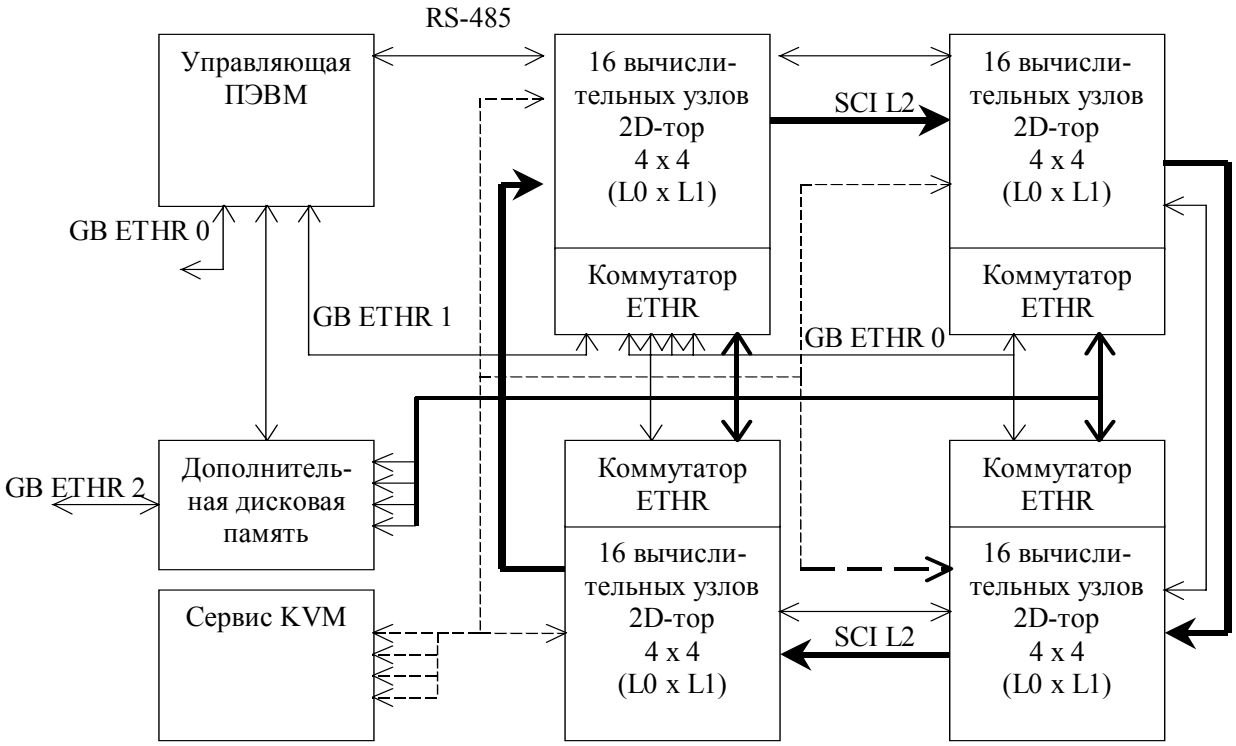


Рис. 7.3

Пиковая производительность каждого узла кластера СКИФ К-500 – 11,2 Гфлопс. SCI-интерфейс обеспечивает задержку при передаче сообщений в соответствии со стандартом MPI – 3 мкс, скорость обмена узел-узел – 263 Мб/с.

Вычислительные узлы соединяются высокоскоростной сетью SCI и образуют трехмерный тор 4 x 4 x 4. Блоки по 16 узлов в отдельных шкафах соединены в друхмерный тор. Тор 3Dобразуется с помощью 16 колец

внешних связей SCI интерфейса между соответствующими вычислительными узлами разных шкафов по линку L2.

Вспомогательный сетевой интерфейс GB Ethernet 0 предназначен для загрузки программ, управления и мониторинга. Он имеет звездообразную топологию, в которой к коммутатору, соединенному с управляющей ЭВМ , подключено три коммутатора остальных шкафов. Вычислительные узлы одного шкафа подключаются к своему коммутатору.

GB Ethernet 1 предназначен для доступа из внешней локальной сети к управляющей ЭВМ.

Сервисная сеть RS-485 предназначена для управления питанием вычислительного узла, взаимодействия с вычислительным узлом в консольном режиме.

KVM обеспечивает подключение любого из вычислительных узлов каждого шкафа к общей клавиатуре, мыши и т.д. для сервисного обслуживания.

В состав программного обеспечения кластера входят:

- Операционная система LINUX RED HAT с поддержкой SPM;
- Система управления, администрирования кластером и поддержка стандарта MPI 1.2 фирмы Scali (SSP 3.0.1)$
- Компиляторы C. C++;
- Система программирования и управления выполнением вычислений – T-система.

7.3.2. Суперкомпьютер СКИФ К-1000

Суперкомпьютер СКИФ К-1000 создается с целью комплексной реализации принципов построения моделей суперкомпьютеров Ряда 2 кластерного уровня с массовым параллелизмом сверхвысокой производительности. Прогнозируется, что технические характеристики К-1000 должны обеспечить включение его в top-500 в 2004-2005 гг.

.