Министерство образования и науки РФ

Пензенская государственная технологическая академия

Р.А. Бикташев

Вычислительные системы. Архитектура высокопроизводительных процессоров и памяти

Учебное пособие

Пенза 2011

ББК

УДК 681.3

Бикташев Р.А. Вычислительные системы. Архитектура высокопроизводительных процессоров и памяти. Учебное пособие. – Пенза : Пенз. гос. технол. ак, 2011.

Рассматриваются теоретические основы параллельных вычислений, классификация параллельных вычислительных систем, уровни параллельной обработки, способы представления параллельных алгоритмов, метрики параллельных вычислений, организация памяти вычислительных систем, структурные методы повышения пропускной способности памяти, организация КЭШ, способы построения ассоциативной памяти, организация виртуальной памяти, способы повышения производительности процессоров, конвейерные, суперскалярные процессоры, векторные, матричные, VLIW – процессоры. Приведены примеры архитектур наиболее популярных процессоров.

Учебное пособие подготовлено на кафедре “Вычислительные машины и системы” Пензенской государственной технологической академии.

Р е ц е н з е н т ы:

ВВЕДЕНИЕ

В настоящее время идёт стремительное развитие технологии СБИС. Интеграция элементов и частота их синхронизации достигли фантастических результатов. Однако человеческая мысль не стоит на месте и технологические усовершенствования подкрепляются новыми структурными подходами к увеличению производительности вычислительных систем. Вопросам структурных методов повышения производительности и посвящено это учебное пособие. Исторически первыми появились методы, повышающие производительность процессоров и однопроцессорных систем. Конвейерные, матричные, векторные вычисления составляли основу скалярных и суперскалярных процессоров и суперЭВМ. Сейчас на первый план выходят многопроцессорные системы как симметричные (SMP - системы) с разделяемой памятью, так и с массовым параллелизмом (MPP - системы). Предложено множество различных вариантов построения таких систем и осуществлена их практическая реализация. В пособии авторы попытались обобщить многообразие публикаций в данной области и систематизировать материал.

1. ТЕОРЕТИЧЕСКИЕ ОСНОВЫ ПАРАЛЛЕЛЬНЫХ ВЫЧИСЛЕНИЙ

1.1. КЛАССИФИКАЦИЯ ВЫЧИСЛИТЕЛЬНЫХ СИСТЕМ

Первые три поколения ЭВМ строились на одном и том же принципе обработки данных. Суть этого принципа составлял последовательный метод управления вычислительным процессом и постоянную структуру вычислительной системы. Задачи, решаемые на однопроцессорной ВС, имели относительно небольшую сложность. На входе ВС они представлялись в виде заданий, которые реализовывались в ВС независимыми процессами.

Структура однопроцессорной ВС, если следовать классификации М. Флинна, представляет собой класс "Одиночный поток Команд - Одиночный поток Данных" (ОКОД) или Singl Instraction – Singl Date (SIMD), который для трех основных устройств машины - памяти, устройства обработки (УО) и устройства управления (УУ) - можно представить схемой, приведенной на рис. 1.1.

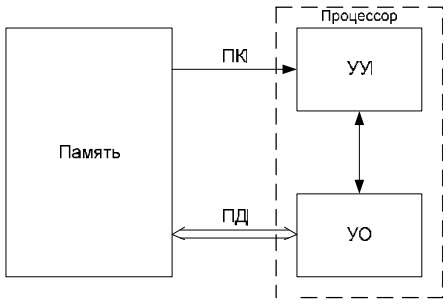


Рис. 1.1

Одинарная линия обозначает потоки команд (ПК), а двойная линия - потоки данных (ПД). Причем устройство управления и устройство обработки образуют в данном случае полноценный процессор. УУ процессора читает из памяти команды, дешифрирует их и направляет в УО (в обычном представлении - АЛУ). Устройство обработки выбирает из памяти данные (операнды) над которыми

выполняются операции, заданные текущей командой; результат возвращается в память.

Появление четвертого поколениия ВС связано с необходимостью решения более сложных задач, которые могут быть решены достаточно быстро лишь при условии применения вычислительных средств в виде некоторого коллектива вычислителей. Точно такие же потребности возникают и при решении задач в реальном масштабе времени (РМВ). При этом решение сложных задач предъявляет высокие требования к производительности (10^{12} - 10^{15}опер/с), надежности и живучести ВС, а также к качеству математического обеспечения и, прежде всего, к качеству операционной системы (ОС). Поскольку возможности по увеличению скорости переключения логических элементов ограничены, поэтому получение высокой производительности невозможно без включения параллелизма в структуру вычислительной системы. Кроме того, требуется динамическое подключение специализированных высокопроизводительных устройств обработки для выполнения трудоёмких операций (например, матричных) в ходе реализации вычислительного процесса.

В связи с изложенным, ВС должны обладать следующими свойствами:

1) автоматическим изменением структуры ВС в зависимости от структуры решаемых задач;

2) параллельно-последовательными алгоритмами управления параллельными вычислительными процессами;

3) возможностью наращивания вычислительной мощности;

4) достаточной надежностью и живучестью;

5) технической и программной модульностью;

6) аппаратной реализацией функций операционных систем.

Пятое поколение ВС связывается с еще более сложными задачами-системами, известными под общим названием "проблемы искусственного интеллекта".

Для удовлетворения этих требований ВС четвертого и пятого поколений строятся по принципам параллельной обработки. Параллельная обработка

информации представляет собой одновременное выполнение некоторого множества независимых программ или частей одной и той же программы на некотором множестве процессоров. Параллельную обработку следует отличать от мультипрограммной, основная цель которой, как известно, заключается в разделении времени оборудования между двумя или более программами, функционирующими квазиодновременно (т.е. создается иллюзия одновременности для пользователей) и размещенными полностью или частично в оперативной памяти. Следует отметить, что возможен мультипрограммный режим и при параллельной обработке.

Вычислительные системы параллельной обработки, или просто параллельные ВС, содержат два или более процессоров (компьютеров) и подразделяются в основном на три класса в соответствии с той же классификацией М. Флинна: "Одиночный поток Команд - Множественный поток Данных" (ОКМД) или Singl Instraction – Multiplle Date (SIMD), "Множественный поток Команд - Одиночный поток Данных" (МКОД) или Multiplle Instraction – Singl Date (MISD) и "Множественный поток Команд - Множественный поток Данных" (МКМД) или Multiplle Instraction – Multiplle Date (MIMD).

В структурах параллельных ВС (ПВС) с организацией типа ОКМД (рис. 1.2) одно устройство управления осуществляет управление работой множества процессорных модулей, так что каждый из них одновременно выполняет сначала одну команду, затем вторую и т.д. Другими словами, здесь реализуется синхронный параллельный вычислительный процесс. Часто такой вид параллелизма называют параллелизмом на уровне данных. К таким ВС относятся ассоциативные, матричные, векторные процессоры и ВС, а также процессоры с длинным командным словом (VLIW- процессоры).

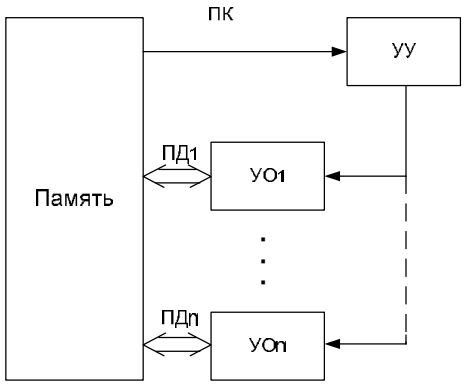


Рис. 1.2

Для структуры ПВС с МКОД процесс обработки разбивается на несколько этапов, каждому из которых соответствует своё УУ и УО (рис. 1.3). К таким ПВС относятся конвейерные ПВС, построенные по одному из трех принципов обработки: арифметико – конвейерному, командно – конвейерному или макро – конвейерному.

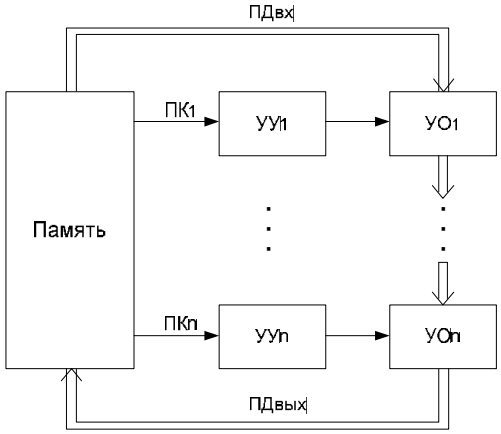


Рис. 1.3

В структурах ПВС с МКМД (рис. 1.4) несколько УУ осуществляют управление одновременным выполнением различных участков одной и той же программы, т.е. здесь реализуется асинхронный параллельный вычислительный процесс. К таким ПВС относятся прежде всего многопроцессорные и многомашинные ВС (МПВС и ММВС). Кроме того, сюда относятся распределенные ВС и вычислительные сети.

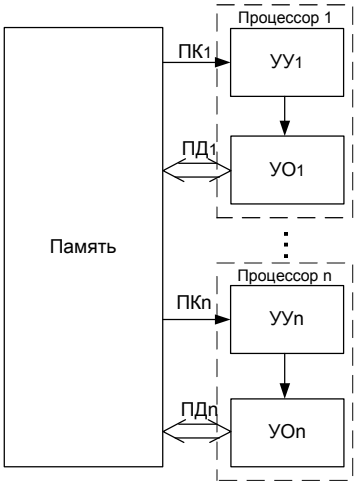


Рис.1.4

Классификационная схема М. Флинна, схема четырех классов сочетаний из различных ПК и ПД, не отражает все возможные структурные решения ПВС. Например, классификацию ПВС можно представить схемой, изображенной на рис. 1.5.

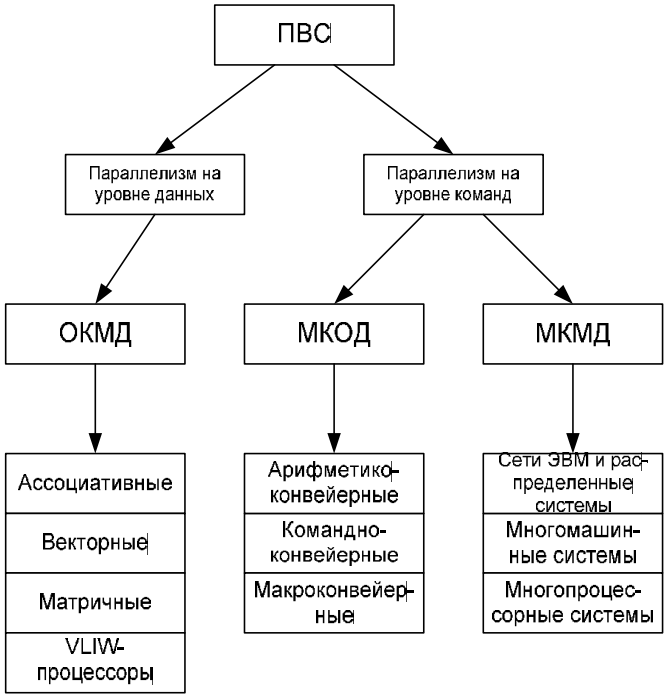


Рис.1.5

Существуют и другие классификации, фактически дополняющие и уточняющие классификацию М. Флинна [Воеводин].

Следует заметить, что в высокопроизводительных ВС и процессорах применяют не один принцип обработки, а совмещают сразу несколько. Так в суперскалярном процессоре Pentium 4 фирмы Intel процессор построен на базе 20-ти стадийного арифметического конвейера и блоков для выполнения целочисленных операций с фиксированной точкой. Для выполнения команд с плавающей точкой используются специализированные блоки FPU (Floating Point Unit). Кроме того в составе процессора имеются блоки расширения MMX и SSE с векторной технологией обработки данных. Все блоки обработки включаются в вычислительный процесс динамически, т.е. в момент появления соответствующей команды в программе. Если текущая команда выполняет операцию с плавающей точкой, то конвейер коммутируется на свободный блок FPU, если текущая команда выполняет 64 – х битную векторную операцию, то подключается блок MMX и т.д.

1.2. УРОВНИ ПАРАЛЛЕЛЬНОЙ ОБРАБОТКИ

Проблема параллельной обработки выдвигает четыре основных направления исследований:

1) формальная теория параллельных вычислений, исследующая общие свойства параллельных программ;

2) языки и методы параллельного программирования;

3) построение (в том числе и автоматическое) параллельных программ, заключающееся в выявлении сегментов (ветвей программ), пригодных для одновременного исполнения;

4) организация параллельных вычислительных процессов, в частности, распределение ветвей программы по процессорам.

Рассмотрим основные вопросы, касающиеся наиболее близкого к аппаратным средствам и интенсивно развивающегося в настоящее время третьего направления.

Виды параллелизма алгоритмов. Обычный процесс вычислений и решения задач на ЭВМ представляет собой последовательность операций, в результате которых получаются искомые данные. Вычислительная математика в течение многих лет создавала алгоритмы решения задач, которые соответствуют этой последовательности. Такие алгоритмы носят название последовательностных алгоритмов. Именно в соответствии с такими алгоритмами действует человек, решая задачу вручную, именно такой характер алгоритмов нашел отражение в структуре наиболее распространенных алгоритмических языков ПАСКАЛЬ, СИ, ФОРТРАН и др., применяемых в однопроцессорных системах.

Необходимость создания параллельных ВС выдвинуло проблему организации параллельных вычислений, которая в основном сводится к созданию методов разработки алгоритмов программ, пригодных для параллельного исполнения на нескольких процессорах. Поэтому в отличие от обычного последовательного программирования начали развиваться методы параллельного программирования.

Параллелизм, встречающийся в вычислительных задачах, можно разделить на четыре уровня: задачи, подзадачи, операции и микрооперации. Для первого уровня на множестве устройств обработки выполняется параллельно несколько независимых задач, для второго уровня - несколько подзадач одной задачи, для третьего - несколько операций и т.д.

Под независимыми понимаются задачи, результаты которых могут использоваться самостоятельно. К ним, например, можно отнести независимые задачи различных пользователей или одни и те же задачи, которые решаются параллельно в системах повышенной надежности. Распараллеливание на уровне задач (процессов) не требует специальных средств. Такие задачи пригодны для решения на многомашинных и многопроцессорных системах.

Суть параллелизма на уровне подзадач, или, как его еще называют, параллелизм независимых ветвей (потоков) состоит в том, что в программе решения задачи на тех или иных этапах могут быть выделены независимые части-ветви, которые при наличии в вычислительной системе соответствующих средств могут выполняться параллельно.

Понятие независимости включает в себя следующее. Ветвь В программы (подзадача В) не зависит от ветви А (подзадача А), если удовлетворяются следующие условия:

1) отсутствие связи между ветвями по данными и по управлению, т.е. в первом случае ни одна из входных переменных для ветви В не является выходной переменной ветви А (или другой ветви, от нее зависящей) и, во втором случае, условие выполнения ветви В не зависит от признаков, вырабатываемых при завершении работы по ветви А (или другой, от нее зависящей);

2) независимость в программном отношении (т.е. ветви должны выполняться по разным программам);

3) независимость по рабочим полям памяти (не должна производиться запись в одни и те же ячейки памяти для разных ветвей).

Для решения задач, распараллеливающихся на независимые ветви, используются в основном многопроцессорные системы. Только в некоторых случаях, когда решается крупная задача, которая имеет длинные независимые ветви и редкий взаимообмен между взаимодействующими ветвями, возможно использование многомашинных или распределенных ВС. Эффективность использования той или иной вычислительной системы для решения конкретной задачи определяется связностью подзадач (ветвей). Например, если в задаче одно из условий, перечисленных выше, не выполняется, то эффективность ее решения на МПВС снижается. Поэтому возникает необходимость использования других принципов обработки. Так, если не выполняется условие 2, т.е. ветви задачи выполняются по одной и той же программе, то будут возникать коллизии за доступ к единственному программному ресурсу, поэтому необходимо применять либо меры к устранению конфликтов (например, копирование программных блоков), что несомненно приведет к увеличению времени выполнения и возрастанию числа необходимых ресурсов ВС, либо использовать матричные или векторные вычисления.

Если не выполняется условие 3 (т.е. данные являются общими или их разделяемыми для множества процессов), то потребуется либо создание дополнительных копий в процессорных узлах (репликация программ или данных),

либо использование дополнительных программных или аппаратных средств для синхронизации процессов. Однако в первом случае также потребуются дополнительные аппаратные или программные средства для обеспечения когерентности (согласованности) разделяемых данных.

Примером задач, распараллеливающихся на ветви, выполняемые по одинаковой программе, являются задачи, которые сводятся к операциям над *n*-мерными векторами, над матрицами, решетчатыми функциями и др. Большинство операций, выполняемых в таких задачах, представляют собой последовательность из одних и тех же команд, которые одновременно выполняются над совокупностью различных данных.

Например, вычисление скалярного произведения для *n*-мерных векторов

$$A \bullet B = \sum_{i=1}^{n} a_i b_i = a_1 b_1 + a_2 b_2 + a_n b_n$$

состоит фактически из операций двух типов: вычисления *n* попарных произведений соответствующих компонент векторов, затем их суммирование. Программа представляет собой цикл, повторяющийся *n* раз, причем тело цикла будет состоять из операций присвоения и умножения. В однопроцессорной системе эта часть программы выполнится за *n* проходов тела цикла.

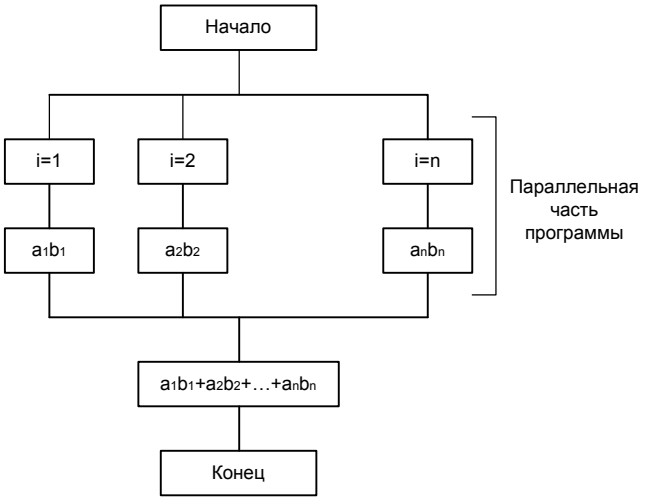


Рис. 1.6

Параллельный алгоритм вычисления произведения векторов A и B можно представить граф - схемой изображенной на рис. 1.6. Из граф – схемы видно, что ветви с 1-й по *n*- ю выполняются параллельно по одинаковой программе, состоящей из двух операций: присвоения и умножения, но с разными данными. Подобные задачи могут обрабатываться многопроцессорной ВС, но при этом каждому процессору желательно иметь копию параллельной программы. В системах типа ОКМД достаточно одной копии, хранящейся в общем устройстве управления. Устройство управления выбирает из памяти программ очередную команду, размножает ее и отправляет в каждое устройство обработки (элементарный процессор) на исполнение. Если число устройств обработки равно *n*, то для выполнения параллельной части программы потребуется время, достаточное для реализации одного прохода тела цикла.

Если в задаче не выполняется условие 1, то для ее решения возможно применение макроконвейерной системы типа МКОД, если выполнение одной и той же программы многократно повторяется. Возможность появляется в том случае, если условия выполнения и исходные данные для выполнения очередной (*i*+1)-й ветви возникают при исполнении *i*-й ветви, (*i*+2)-й ветви при исполнении(*i*+1)-й ветви и т.д. В этом случае можно совместить время исполнения *i*-й ветви с исполнением ветвей (*i*+1), (*i*+2), и т.д.

Вышесказанное иллюстрируется примером, изображенным на рис. 1.7.

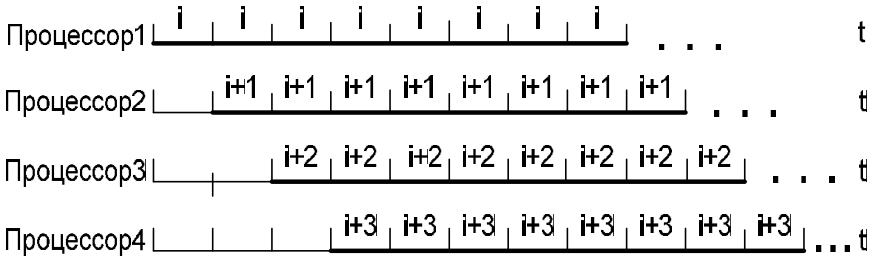


Рис. 1.7

Для упрощения принято, что задачи разбиваются на 4 ветви: i; (i+1); (i+2) (i+3). Условие выполнения (*i*+1) - й ветви возникает после выполнения *i*-й ветви и т.д. Поэтому, начиная с *i*-й ветви можно запускать в конвейер вычислений.

Процессорам 1,...,4 каждый раз после выполнения ветви вновь назначается та же ветвь (одной и той же циклически повторяющейся задачи), только с другими данными. Поэтому, начиная с (i+3) ветви, совмещается во времени выполнение всех ветвей задачи и вычисления производятся одновременно на 4-х процессорах.

Параллелизм на уровне операций и микроопераций (его называют параллелизмом смежных операций и микроопераций) в принципе не отличается от параллелизма ветвей. Отличие заключается в удельном весе, который они занимают в программе. Однако это отличие приводит к совершенно иным структурам процессоров или вычислительных систем. При параллелизме смежных операций используются командно - конвейерные, при параллелизме смежных микроопераций - арифметико-конвейерные процессоры. Однако при внешнем сходстве конвейерного выполнения, структурная организация разных уровней конвейеризации полностью отличаются.

К понятию уровня параллелизма тесно примыкает понятие гранулярности. Это мера отношения объема вычислений, выполненных в параллельной задаче, к объему коммуникаций (для обмена сообщениями). Степень гранулярности варьируется от мелкозернистой до крупнозернистой. Понятия крупнозернистого, среднезернистого и мелкозернистого параллелизма заключаются в следующем. При крупнозернистом параллелизме каждая ветвь параллельной программы достаточно независима от остальных, причем требуется относительно редкий обмен информацией между отдельными ветвями. Ветви могут включать сотни тысяч команд. Этот уровень параллелизма обеспечивается программистом.

При среднезернистом параллелизме параллельными ветвями могут являться вызываемые процедуры, включающие в себя несколько сотен команд. Обычно организуется как программистом, так и компилятором.

Мелкозернистый параллелизм состоит обычно из десятков команд. Распараллеливаемыми единицами являются элементы выражения или отдельные итерации цикла, имеющие небольшие зависимости по данным. Характерная особенность мелкозернистого параллелизма заключается в приблизительном равенстве интенсивности вычислений и интенсивности взаимодействия между

ветвями. Этот уровень параллелизма часто организуется распараллеливающим компилятором.

Эффективное параллельное исполнение параллельной программы зависит от уровня гранулярности. Так, например, задачи с крупной гранулярностью могут эффективно решаться на многопроцессорных и многомашинных системах, среднегранулярные – только на многопроцессорных. Мелкогранулярные задачи требуют применения матричных, векторных или командно-конвейерных процессоров.

Представление параллельных алгоритмов.

Для представления параллельных алгоритмов используются разные методы: язык *p*-схем, разработанный Э.В. Евреиновым и Ю.Г. Косаревым, аппарат ярусно-параллельных форм, предложенный Д.А. Поспеловым и др. Все эти методы в общем близки друг к другу.

Например, при использовании аппарата ярусно-параллельных форм программа представляется в виде "ярусов" на графе. Причем в 0-й ярус входят все те ветви программы, каждая из которых не зависит ни от одной ветви, в 1-й ярус - ветви, зависящие только от ветвей 0-го яруса, во 2-й ярус - ветви, зависящие от ветвей 1-го яруса и, может быть, ветвей 0-го яруса и т.д.

На рис. 1.8 в виде примера изображена некоторая программа, представленная графом в ярусно-параллельной форме. Кружками отмечены операторные вершины графа, обозначающие ветви программы, внутри операторов проставлены номера ветвей, а рядом - длины этих ветвей. Дугами показаны функциональные связи (связи по данным) между ветвями. Дуга, входящая в некоторый оператор, обозначает входную переменную для соответствующей ветви. Если дуга не выходит ни из какого другого оператора, то она соответствует входным данным программы. Дуга, выходящая из некоторого оператора, обозначает выходную переменную соответствующей ветви. Если эта дуга не входит ни в какой другой оператор, то она обозначает выходные данные программы

Разветвления дуг показывают, что одни и те же данные являются входными для двух или более ветвей.

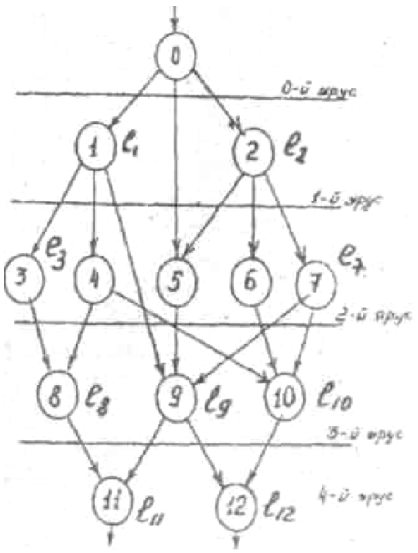


Рис. 1.8

Такое представление алгоритмов используют для оценки возможностей организации параллельных вычислений. Для этого вводят ряд количественных характеристик для программ, представленных в ярусно-параллельной форме: b_i - ширина i-го яруса (т.е. количество независимых ветвей в i-м ярусе); B- ширина графа (максимальная ширина яруса, т.е. $B=max\{bi\}$); l_i -длина яруса (обычно представляется трудоемкостью ветви, имеющей набольшую длину в этом ярусе); L - длина графа (максимальная длина пути, ведущего из нулевого в конечное состояние, называемая *критическим путем*); S_{ij} - связанность операторов или ветвей задачи, определяется количеством информации, передаваемой между i-й и j-й ветвями задачи в процессе ее решения; S - связанность графа (представляет связанность всех ветвей задачи) определяется матрицей связности $S=S_{ij}$.

Несвязанные, слабосвязанные и сильносвязанные задачи. Будем для упрощения считать, что сложная задача, решаемая ВС, состоит из набора простых задач (которые могут быть независимыми задачами или независимыми ветвями сложной задачи). Две или более простые задачи называются связанными, если данные одной задачи используются не только в собственном, но и в вычислительном процессе какой-нибудь задачи.

Сложная задача, состоящая из n простых задач $P_н = \{P_1, P_2,..., P_n\}$, называется несвязанной, если все элементы матрицы связности S равны нулю, т.е. $S_{ij} = 0$, $i,j=1,n$. Несвязанные задачи образуют простейший класс сложных задач, который является естественным обобщением класса одиночных задач. Одиночные простые задачи соединяются в некоторые наборы задач для их решения одними им теми же вычислительными средствами. Класс несвязанных задач является довольно распространенным. Они характерны для информационно-вычислительных систем, автоматизированных систем управления и т.п.

Сложная задача $P_c = \{P_1, P_2,..., P_n\}$ называется слабосвязанной, если не все элементы ее матрицы связности S равны нулю. Слабосвязанные задачи образуют обширный класс задач, различающихся как типом графа, описывающего связи между задачами так и объемом данных, передаваемых между простыми задачами. Считается, что общий объем обменных взаимодействий в слабосвязанных задачах много меньше объема вычислений одной простой задачи.

Наличие связанности в сложной задаче, несмотря на незначительный объем обменных взаимодействий, не позволяет эффективно решать задачи такого типа на одной ЭВМ или на совокупности не связанных между собой ЭВМ.

Сложная задача $P_k = \{P_1, P_2,..., P_n\}$ называется сильносвязанной, если все элементы ее матрицы связности S не равны нулю. В сильносвязанных задачах резко возрастает число обменных взаимодействий, причем они осуществляются не только после решения набора простых задач, составляющих сложную, но и в процессе решения простых задач, в предельном случае - на каждом шаге вычислений.

Наличие связности в сложных задачах налагает определенные требования на пропускную способность соединительной сети связи между модулями параллельной ВС (ЭВМ, процессорами, процессорными элементами и др.).

Средства определения параллелизма в программе.

При программировании задач на общепринятых алгоритмических языках, допускающих параллельное решение для обозначения независимых частей используются дополнительные операторы типа разветвление (FORK) и объединение (JOIN). При этом от программиста требуется, чтобы он сам расставил в тексте

программы операторы FORK<список ветвей>, обозначающие, что, начиная от данной точки программы, можно одновременно запускать ветви, перечисленные в списке, и операторы JOIN <список ветвей>, обозначающие, что дальнейшее продолжение программы возможно после выполнения всех ветвей, перечисленных в списке.

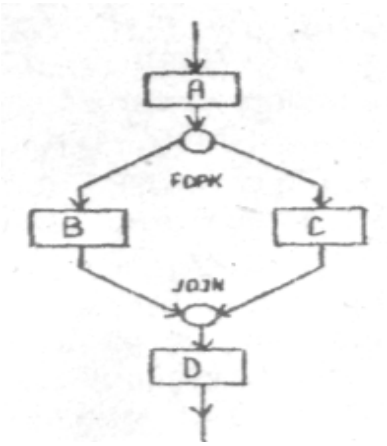


Рис. 1.9

Использование операторов FORK и JOIN поясняется на рис. 1.9. Допустим, что выполнение ветвей B и C некоторой программы может быть начато одновременно на разных процессорах, а обработка ветви D не может начаться раньше, чем закончится обработка ветвей B и C. Тогда к ветвям B и C применяют оператор FORK B, C. По окончании обработки применяют оператор JOIN B, C, объединяющий эти ветви.

Очевидно, что кроме средств разветвления и объединения в программе должны быть предусмотрены средства, которые могли бы определять момент окончания обработки наиболее трудоемкой ветви. Эти средства называют барьерной синхронизацией.

Метрики параллельных вычислений

При реализации задач на параллельных ВС очень важным является оценка эффективности параллельных вычислений. Для этого используют метрики, наиболее распространенные из которых рассматриваются ниже.

Профиль параллелизма программы

Число процессоров многопроцессорной системы, параллельно участвующих в выполнении программы в каждый момент времени *t*, определяют понятием *степень параллелизма D(t)*. Графическое представление параметра *D(t)* как функции

времени называют *профилем параллелизма программы*. Профиль параллелизма показывает уровень загрузки процессоров в течение времени решения задачи. Он зависит от многих факторов (алгоритма, доступных ресурсов, степени оптимизации, обеспечиваемой компилятором и т. д.). Пример профиль параллелизма для параллельного алгоритма показан на рис. 1.10. В интервалы времени t_1-t_3 и t_{23}-t_{26} загружен один процессор, в интервале t_3-t_5 загружены два процессора и т.д.

Пусть вычислительная система состоит из n одинаковых процессоров; максимальный параллелизм в профиле равен m, причем $n > m$. Производительность P одиночного процессора системы выражается в количестве операций, выполняемых в единицу времени и не учитывает издержек, связанных с обращением к памяти и пересылкой данных. Если за наблюдаемый период загружены i процессоров, то $D(t) = i$.

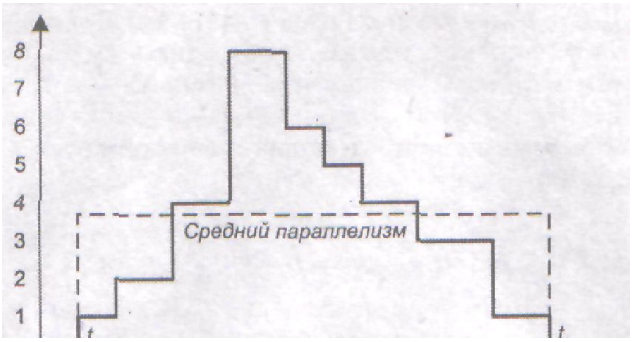


Рис. 1.10

Общий объем вычислительной работы θ (количество операций), выполненной с момента старта t_s до момента завершения t_e, пропорционален площади под кривой профиля параллелизма. Профиль параллелизма на рисунке за время наблюдения (t_s, t_e) возрастает от 1 до пикового значения m = 8, а затем спадает до 0. Средний параллелизм (загрузка процессоров) ρ = (1 x 5 + 2 x 3 + 3 x 4+4 x 6 + 5 x 2 + 6 x 2 + 8 x 3)/(5+ 3+4 + 6 + 2 + 2 + 3) = 93/25 = 3,72.

Фактически общая рабочая загрузка ρ и представляют собой нижнюю границу асимптотического ускорения.

Ускорение, эффективность, загрузка и качество

Предположим, что параллельная система состоит из n процессоров. Определим ускорение, которое создается параллельной программой. Пусть $T(n)$ — время выполнения параллельной программы на n- процессорной системе, а $T(1)$ - время выполнения этой же программы на однопроцессорной системе.

Ускорение - это отношение времени, требуемого для выполнения программы на одном процессоре и времени параллельного вычисления на n процессорах. Без учета различного рода издержек ускорение определяется как

$$\psi = \frac{T(1)}{T(n)}$$

Как правило, ускорение удовлетворяет условию $\psi < n$.

Эффективность n – процессорной системы — это ускорение, приходящееся на один процессор, определяемое выражением

$$\varepsilon = \frac{\psi}{n}$$

Эффективность отвечает условию $1/n \le \varepsilon_n \le 1$, что означает, что обычно часть мощности процессора теряется на решение системных проблем, связанных с конфигурированием в параллельную систем, т.е. организация вычислений на n процессорах связана с издержками вычислительного процесса. Поэтому имеет смысл ввести понятие потерь производительности, отражающее степень снижения производительности процессора в составе параллельной системы.

$$\pi = 1 - \varepsilon$$

Рассмотрим пример. Положим, что последовательный алгоритм выполняется за 12 с., а параллельный алгоритм реализуется на 8-ми процессорах за 2 с. Тогда:

$$\psi = 12/2 = 6$$

$$\varepsilon = 6/8 = 0{,}75$$

$$\pi = 1 - 0{,}75 = 0{,}25$$

Если ускорение, достигнутое на n процессорах, равно n, то говорят, что алгоритм показывает линейное ускорение. В исключительных ситуациях ускорение ψ может

быть больше, чем *n*. В этих случаях иногда применяют термин суперлинейное ускорение. Данное явление имеет шансы на возникновение в двух следующих случаях:

- Если последовательная программа и данные размещены в небольшом сегменте памяти, вызывая, таким образом, частый свопинг (подкачку данных с диска).
- Если кэш имеет малые размеры, что приводит к большему числу кэш-промахов.

В параллельных же программах ветви короче, поэтому зачастую используют много меньший набор данных.

Факторы, ограничивающие ускорение

- **Программные издержки.** Параллельные алгоритмы имеют дополнительные программные издержки по сравнению с последовательными даже если они выполняют одни и те же вычисления. Это связано в первую очередь с расширением функций операционных систем, например, функцией распределения задач по процессорам, которая отсутствует в однопроцессорных системах. Параллельные программы также могут вызывать различного рода конфликты при их исполнении, отсутствующие в алгоритмах последовательных, и требующие дополнительного программного кода для их разрешения. Такая ситуация может возникнуть, например, при доступе к программному коду операционной системы или его отдельным частям, которые разделяется между всеми процессорами.
- **Коммуникационные издержки.** Межпроцессорные коммуникации снижают ускорение, если обмен информацией и вычисления не перекрываются во времени. Поэтому важен уровень гранулярности, определяющий объем вычислительной работы, выполняемой между коммуникационными фазами алгоритма. Достичь уменьшения коммуникационных издержек можно в том случае, когда вычислительные гранулы являются достаточно крупными, а коммуникации имеет сравнительно небольшой объем.
- **Издержки из-за дисбаланса загрузки процессоров.** Между точками барьерной синхронизации ветви программ каждого из процессоров должны обладать одинаковой трудоемкостью, иначе часть процессоров будут ожидать, пока остальные завершат свои операции. Эта ситуация называется *дисбалансом*

загрузки. Таким образом, время выполнения программы, и, следовательно, ускорение определяется длиной критического пути на ярусно - параллельном графе алгоритма.

- **Издержки из-за последовательных участков программ** Параллельная вычислительная система обеспечивает значительное повышение скорости вычислений за счет разделения вычислительной нагрузки на множество независимо работающих процессоров. В лучшем случае система из n процессоров могла бы ускорить вычисления в n раз**.** В действительности достичь такого показателя по ряду причин не удается. Главная из этих причин заключается в невозможности полного распараллеливания задач. Как правило, в каждой программе имеется фрагмент кода, который принципиально должен выполняться последовательно и только одним из процессоров. Это может быть часть программы, например, отвечающая за распределение ветвей задачи по процессорам. Таким образом, добиться увеличения производительности прямо пропорционально числу процессоров ни коем образом не удастся.

Закон Амдала

Амдал предложил формулу, отражающую зависимость ускорения вычислений, достигаемого на параллельной ВС, от числа процессоров и соотношения между последовательной и параллельной частями программы. Пусть ускорение ψ - это отношение времени t_1, затрачиваемого на проведение вычислений на однопроцессорной ВС, ко времени t_n решения той же задачи на n – процессорной системе:

$$\psi = \frac{t_1}{t_n}$$

Определим ускорение, которое дает n – процессорная система с учетом последовательных участков программ (рисунок 1.11).Будем считать, что объем решаемой задачи остается неизменным независимо от числа процессоров, участвующих в ее решении. Программный код решаемой задачи состоит из двух частей: последовательной и параллельной. Обозначим долю операций, которые должны выполняться последовательно одним из процессоров, через f, где $0 \le f \le 1$ (под долей понимается отношение трудоемкости последовательной части

программы, измеряемой в машинных операциях, к трудоемкости всей программы в целом). Отсюда доля, приходящаяся на распараллеливаемую часть программы, составит $1 - f$. Если $f = 0$, то это соответствуют полностью параллельным, а $f = 1$, то полностью последовательным программам. Параллельная часть программы содержит одинаковые ветви и равномерно распределяется по всем процессорам вычислительной системы.

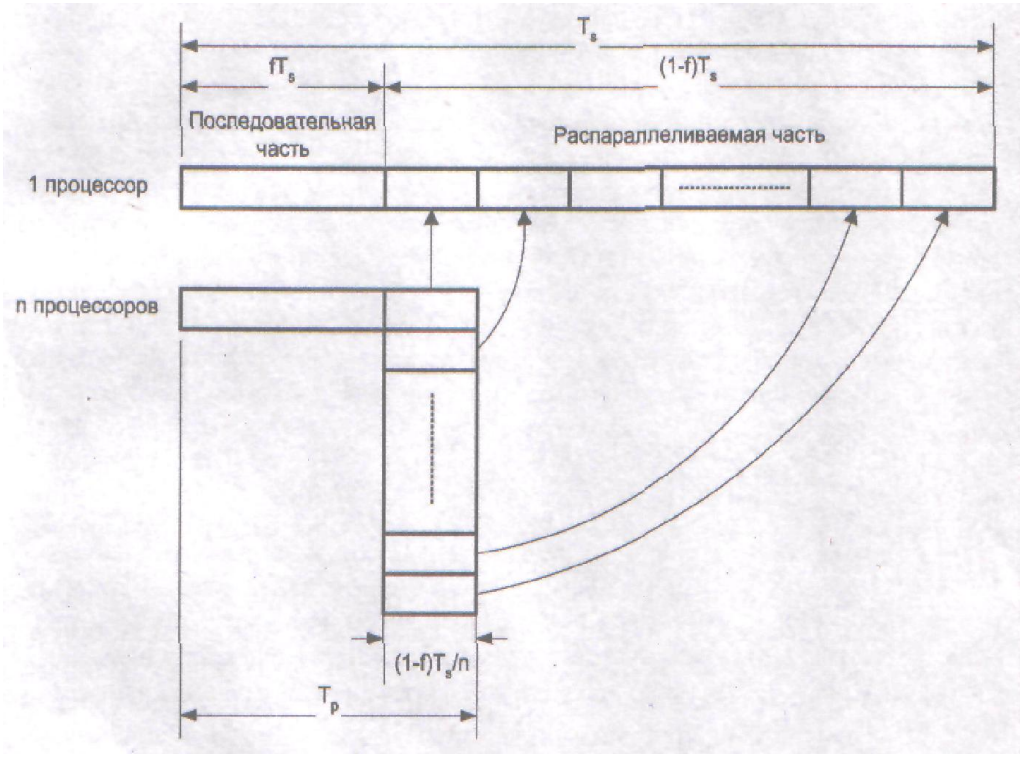


Рисунок 1.11- Определение величины ускорения в законе Амдала

$$t_n = ft_1 + \frac{(1-f)t_1}{n}$$

$$\psi = \frac{t_1}{t_n} = \frac{n}{1+f(n-1)}$$

Если устремить число процессоров к бесконечности, то в пределе получим:

$$\lim_{n\to\infty} \psi = \frac{1}{f} \quad .$$

Характер зависимости ускорения от числа процессоров и доли последовательной части программы показан на рис. 1.12.

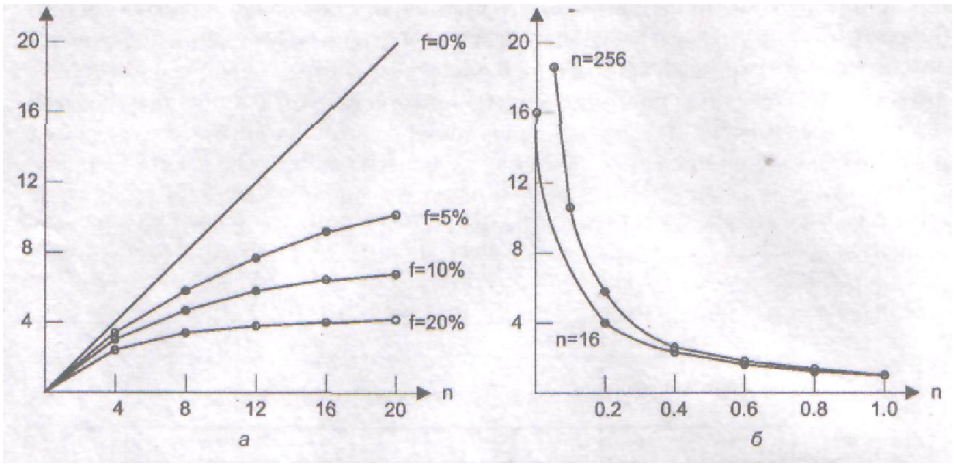


Рисунок 1.12 – Графики зависимости ускорения от доли последовательной части программы (а) и числа процессоров (б)

Закон Амдала показывает, что если в программе 10% последовательных операций (то есть $f = 0{,}1$), то, увеличения скорости работы программы более чем в десять раз нельзя получить, сколько бы процессоров не использовалось. Это верхняя оценка самого лучшего случая, когда никаких других видов издержек нет.

Закон Густафсона

Исследователям Густафсоном замечено, что при решении крупных задач на большой системе, состоящей из 1024 процессоров доля последовательного кода f лежала в пределах от 0,4 до 0,8 %, он получил значения ускорения по сравнению с однопроцессорным вариантом, равные более 1000. Согласно заколу Амдала для данного числа процессоров и диапазона f, ускорение не должно было превысить величины порядка 200. Густафсон пришел к выводу, что причина кроется в том, что в основе закона Амдала стоит повышение производительности ВС с увеличением числа процессоров при неизменном объеме задачи. Пользователь же обычно, получая в свое распоряжение более мощную ВС, стремится не сократить время вычислений, а

старается пропорционально мощности ВС увеличить объем решаемой задачи. И тут оказывается, что наращивание общего объема программы касается главным образом распараллеливаемой части программы. Это ведет к сокращению значения *f*. Примером может служить решение дифференциального уравнения в частных производных. Если доля последовательного кода составляет 10 % для 1000 узловых точек, то для 100 000 точек доля последовательного кода снизится до 0,1 %. Сказанное иллюстрирует рис. 1.13, который отражает тот факт, что, оставаясь практически неизменной, последовательная часть в общем объеме увеличенной программы имеет уже меньший удельный вес

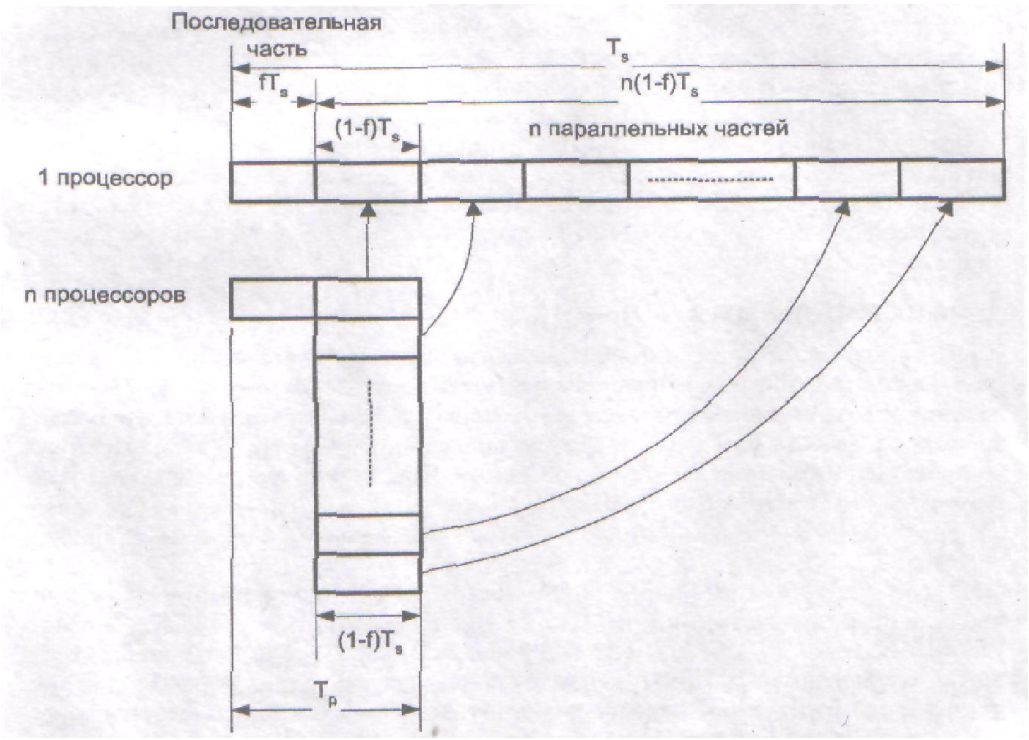


Рисунок 1.13- Определение величины ускорения в законе Густафсона

Было отмечено, что в первом приближении объем работы, которая может быть произведена параллельно, возрастает линейно с ростом числа процессоров в системе. Для того чтобы оценить возможность ускорения вычислений, когда объем последних увеличивается с ростом количества процессоров в системе (при постоянстве общего времени вычислений), Густафсон рекомендует использовать выражение, предложенное Е. Барсисом:

$$\psi = \frac{t_1}{t_n} = \frac{ft_1 + n(1-f)t_1}{ft_1 + (1-f)t_1} = n + (1-n)f.$$

Данное выражение известно как *закон масштабируемого ускорения* или *закон Густафсона* (иногда его называют также законом Густафсона-Барсиса). Закон Густафсона не противоречит закону Амдала. Различие состоит лишь в том, как используется дополнительная мощность вычислительной системы, возникающая при увеличении числа процессоров.

2. Организация памяти вычислительных систем

Иерархическая структура памяти

Желательно, чтобы память ЭВМ обладала как можно большей емкостью и как можно большим быстродействием. Трудно, однако, найти тип памяти, который бы удовлетворял этим противоречащим друг другу требованиям одновременно. Тенденция развития памяти такова, что более быстродействующие из них имеют и более высокую стоимость. Поэтому реализовать память, обладающую большой емкостью и высоким быстродействием, можно при совместном использовании дешевых запоминающих устройств (ЗУ) большой емкости и небольших по емкости, но быстродействующих ЗУ. В оптимальном сочетании таких ЗУ и состоит суть многоуровневой структуры памяти.

На рисунке 2.1 показан классический пример памяти с многоуровневой структурой. Емкость памяти на каждом из уровней этой памяти увеличивается в направлении от процессора в следующей последовательности: регистры процессора, быстродействующая буферная память, основная память, внешняя память, а повышение быстродействия этих устройств идет в обратном порядке.

С точки зрения программиста основная память рассматривается как ЗУ с произвольной выборкой и одномерной адресацией. Наличие буферного ЗУ лишь увеличивает эквивалентную скорость выборки из основного ЗУ, не внося никаких изменений в используемую программой систему адресации. Буферное ЗУ называется кэш-памятью.

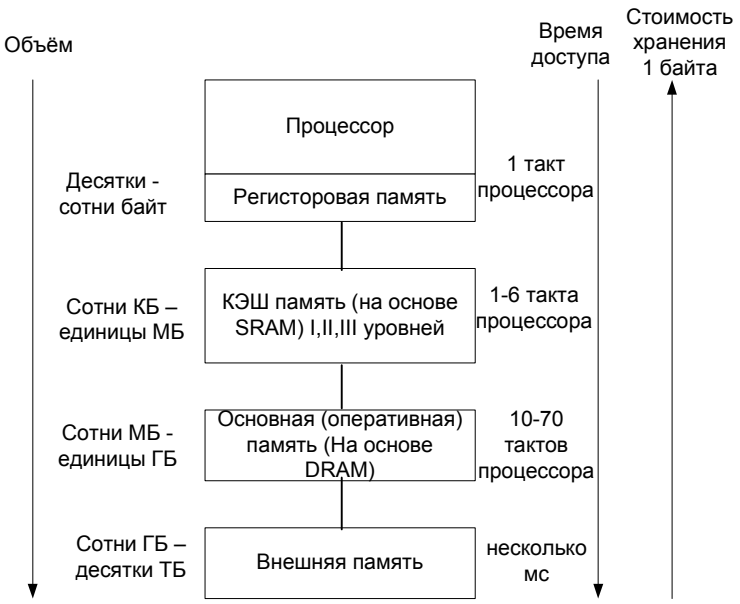


Рисунок 2.1- Многоуровневая структура памяти

Память, в организации которой используется механизм расширения ограниченной емкости основной памяти с помощью устройств памяти, например накопителей на магнитных дисках, называется виртуальной памятью. В виртуальной памяти без увеличения физической емкости основного ЗУ, образуется единое и расширенное, с точки зрения программиста, фиктивное адресное пространство, адресация которого не зависит от физических характеристик составляющих его устройств, и тем самым обеспечивается использование программой большой емкости памяти.

Между основным и внешним ЗУ можно также ввести буферное ЗУ. Оно называется дисковым КЭШем и предназначено для повышения эквивалентной скорости обращения к ЗУ на магнитных дисках (МД).

Как отмечалось выше, многоуровневая организация связана с передачей информации между разнотипными ЗУ и обеспечивает одновременно и быстродействие, и большую емкость памяти ЭВМ. Но в основе такой организации заложен принцип локального обращения к ЗУ (принцип локальности ссылок). Это означает, что область памяти, к которой происходят обращения работающих программ в течение некоторого

интервала времени, является, как правило, небольшой. Для этого программы и данные, обращение к которым происходит часто, размещаются в быстродействующих ЗУ, что способствует повышению быстродействия многоуровневой памяти в целом. При этом данные, обращение к которым происходит часто, размещаются в быстродействующих ЗУ, что способствует повышению быстродействия многоуровневой памяти в целом.

Рассмотрим в качестве модели многоуровневой памяти структуру на рис. 2.2, состоящую из ЗУ двух типов *M*1 и *M*2. Пусть их емкости равны соответственно n_1 и n_2, причем $n_2>>n1$, а времена выборки t_1 и t_2. Эквивалентное время выборки $t_Э$ многоуровневой памяти, состоящей из М1 и М2, определяется по формуле

$$t_{\acute{Y}} = t_1 + pt_2$$

где $p = 1/N$ (коэффициент промаха) - вероятность обращения к *M*2 из-за отсутствия данных в *M*1. Она показывает, что на *N* обращений к *M*1 в среднем приходится одно обращение к *M*2. Значение *N* увеличивается по мере увеличения n_1, причем стремятся обеспечить его больше 10 (обычно $N = 10 - 100$, что соответствует вероятности промаха $p = 0{,}1- 0{,}01$). В результате $t_Э$ принимает значение, близкое к t_1.

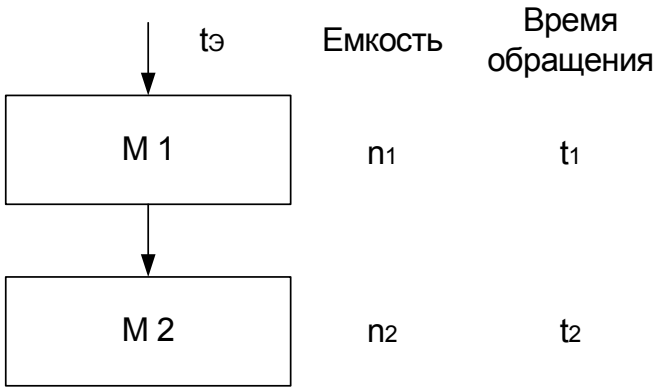


Рис.2.2. Модель двухуровневой памяти.

Расслоение памяти

Традиционно самым медленным устройством в подсистеме процессор-память является основная (оперативная) память. При обращении к основной памяти с целью ускорения этого процесса запись и считывание нескольких байтов могут осуществляться за один раз. При этом большей скорости доступа можно достичь за счет одновременного доступа ко многим независимым модулям памяти (независимыми являются модули, имеющие собственные схемы адресации и буферизации данных). С этой целью в большинстве современных ЭВМ основная память делится на несколько независимых модулей (блоков), т.е. применяется процедура *расслоения памяти*. В ее основе лежит так называемое *чередование адресов*, заключающееся в изменении системы распределения адресов между модулями памяти. Суть этой процедуры состоит в том, что при числе модулей n поступающий адрес a относится к модулю с номером $(a) \bmod n$.

Например, при четырех блоках памяти адресация выполняется в соответствии со структурой, показанной на рис. 2.3. Прием чередования адресов базируется на ранее рассмотренном свойстве локальности по обращению (локальности ссылок), согласно которому последовательный доступ в память обычно производится к ячейкам, имеющим смежные адреса. Иными словами, если в данный момент выполняется обращение к ячейке с адресом 4, то следующее обращение с большой вероятностью будет к ячейке с адресом 5, затем 6 и т. д. В самом деле, программы и данные располагаются в памяти обычно в последовательных адресах, что создает условия для высокоэффективной работы метода чередования адресов.

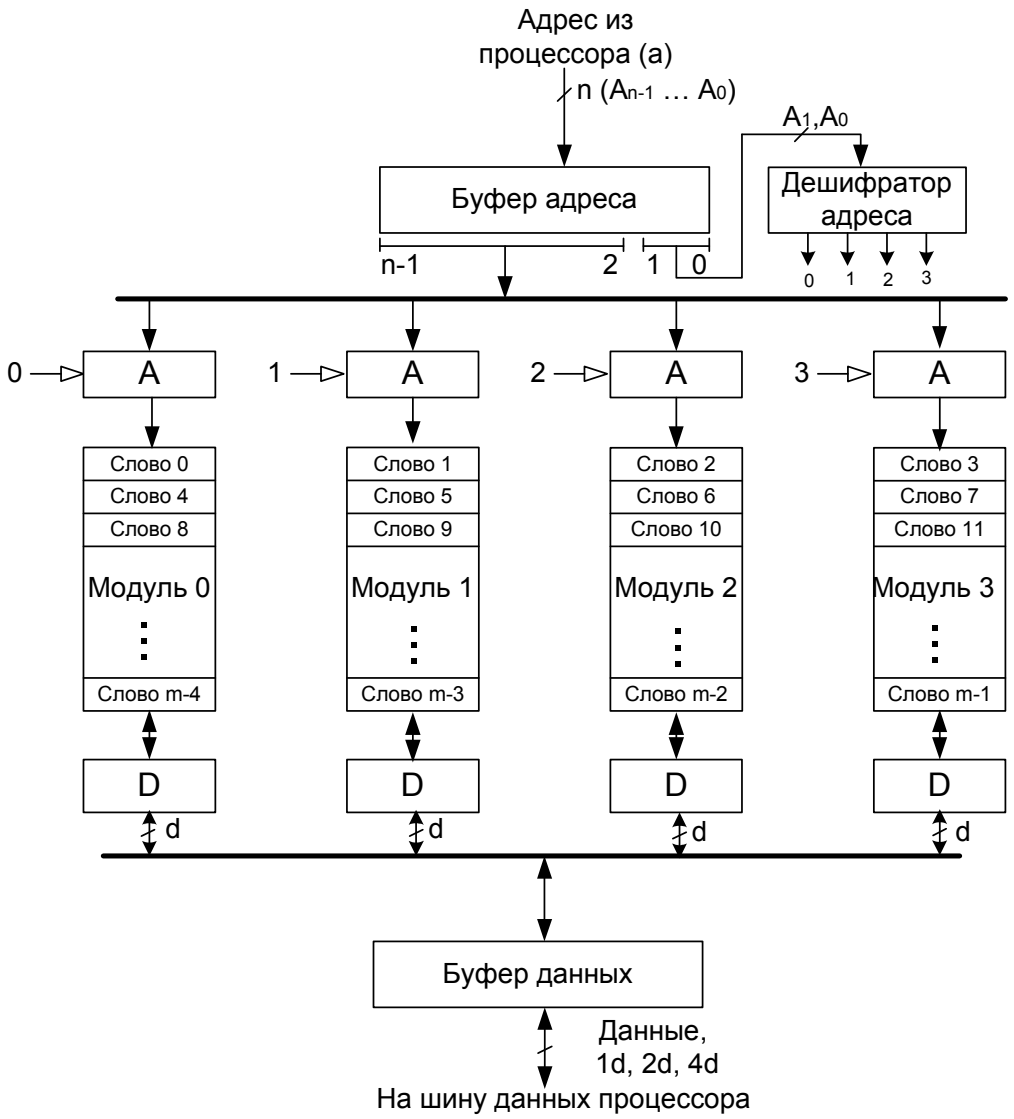


Рисунок 2.3- Схема памяти с расслоением.

Чередование адресов обеспечивается за счет циклического разбиения адреса. Для выбора модуля используются младшие разряды адреса. В приведенной на рисунке 2.3 схеме используются два младших разряда адреса A_1 и A_0 для выбора модуля памяти (см. рисунок 2.4), а для выбора ячейки в модуле - старшие разрядов (A_{n-1} - A_2).

A_1	A_0	№ модуля памяти
0	0	0
0	1	1

1	0	2
1	1	3

Рисунок 2.4- Таблица выбора модулей памяти по значению младших разрядов адреса

Хотя адреса памяти могут генерироваться с тактовой частотой процессора, но в каждом такте на шине адреса может присутствовать адрес только одной ячейки, т.е. параллельное обращение к нескольким модулям невозможно. Поэтому исполнительный адрес поступает на буфер адреса со смещением на один процессорный такт $t_т$. Адрес ячейки запоминается в индивидуальном регистре адреса А, на который указывает дешифратора адреса. Дальнейшие операции по доступу к ячейке в каждом модуле протекают независимо. В начальный момент времени процессор обращается к модулю 0 и запускается цикл памяти. Не дожидаясь окончания цикла памяти, производится обращение к модулю 1 и т.д. конвейерным методом. Следующее обращение к модулю памяти должно состояться не ранее чем закончится цикл этого модуля $t_ц$ (рисунок 2.5). При обращениях к последовательным соседним адресам можно добиться n- кратного увеличения скорости обращения за счет параллельного действия всех блоков. Подобная процедура распределения адресов по блокам называется n- расслоением обращений к памяти.

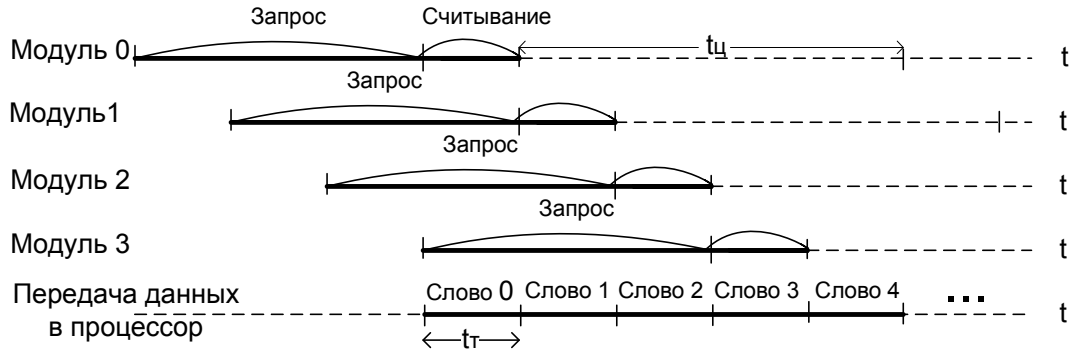


Рисунок 2.5 - Временная диаграмма работы памяти с расслоением.

Передача слов между регистрами D и буфером данных осуществляется через мультиплексор/демультиплексор (на рисунке 2.3 не показан). Прочитанные из памяти данные направляются в процессора с размерностью в одно слово (1d), двойного слова (2d) или четверного слова (4d). Передача данных в режиме записи производится пословно. Схема управления записью/ чтением данных (на рисунке не показана) является общей для всех блоков и функционирует в режиме разделения времени. Таким образом, конструктивно расслоенная память представляет собой единое устройство.

Если же запросы к одному и тому же модулю следуют друг за другом, каждый следующий запрос должен ожидать завершения обслуживания предыдущего. Такая ситуация называется *конфликтом по доступу*. При частом возникновении конфликтов по доступу метод чередования адресов становится неэффективным.

Устройство памяти с расслоением может применяться как в одно- так и многопроцессорных системах, где требуется очень высокая пропускная. Число модулей памяти *n* варьируется обычно в пределах 2—16. В некоторых случаях их число может достигать 64 или даже 128. В некоторых ВС возможен отказ от применения кэш, если используется память с высокой степенью расслоения. Это обстоятельство важно для многопроцессорных систем, где использование кэш в процессорных узлах наряду с положительными качествами создает проблемы, например, проблему обеспечения согласованности разделяемых (общих для всех процессоров) данных или кэш-когерентности (см. раздел ?).

Банковая организация памяти

Традиционные способы расслоения памяти хорошо работают в рамках одной задачи, для которой характерно свойство локальности. В многопроцессорных системах с общей памятью, где запросы на доступ к памяти

достаточно независимы, чаще применяют другой подход, который можно рассматривать как развитие идеи расслоения памяти. Для этого в систему включают несколько банков памяти со своими контроллерами, что позволяет отдельным модулям работать полностью автономно. Причем каждый модуль может быть выполнен с чередованием адресов, что обеспечивает дополнительное повышения пропускной способности памяти.

Обычно такая архитектура памяти применяется в многопроцессорных системах с общей шиной. Структурно-функциональная организация памяти показана на рисунке 2.6. Процессоры взаимодействуют с памятью через внешнюю общую шину (на схеме не показана), работающую в режиме временного мультиплексирования и расщепления транзакций (см. раздел?). Процессоры выставляют адрес памяти на шину адреса, причем старшие разряды используются для выбора модуля с помощью дешифратора адреса, а младшие - для адресации внутри модуля последовательно как в обычной памяти.

Эффективность банковой организации памяти напрямую зависит от частоты обращений процессоров к разным модулям. Уменьшение вероятности последовательных обращений к одному и тому же модулю памяти можно добиться при большом числе модулей. Так, в суперкомпьютере NEC SX/3 основная память состоит из 128 модулей. Кроме того, в целях сокращения конфликтов по доступу необходимо, чтобы компилятор или операционная система, учитывая архитектуру памяти, распределяли сегменты программ и данных в разные модули.

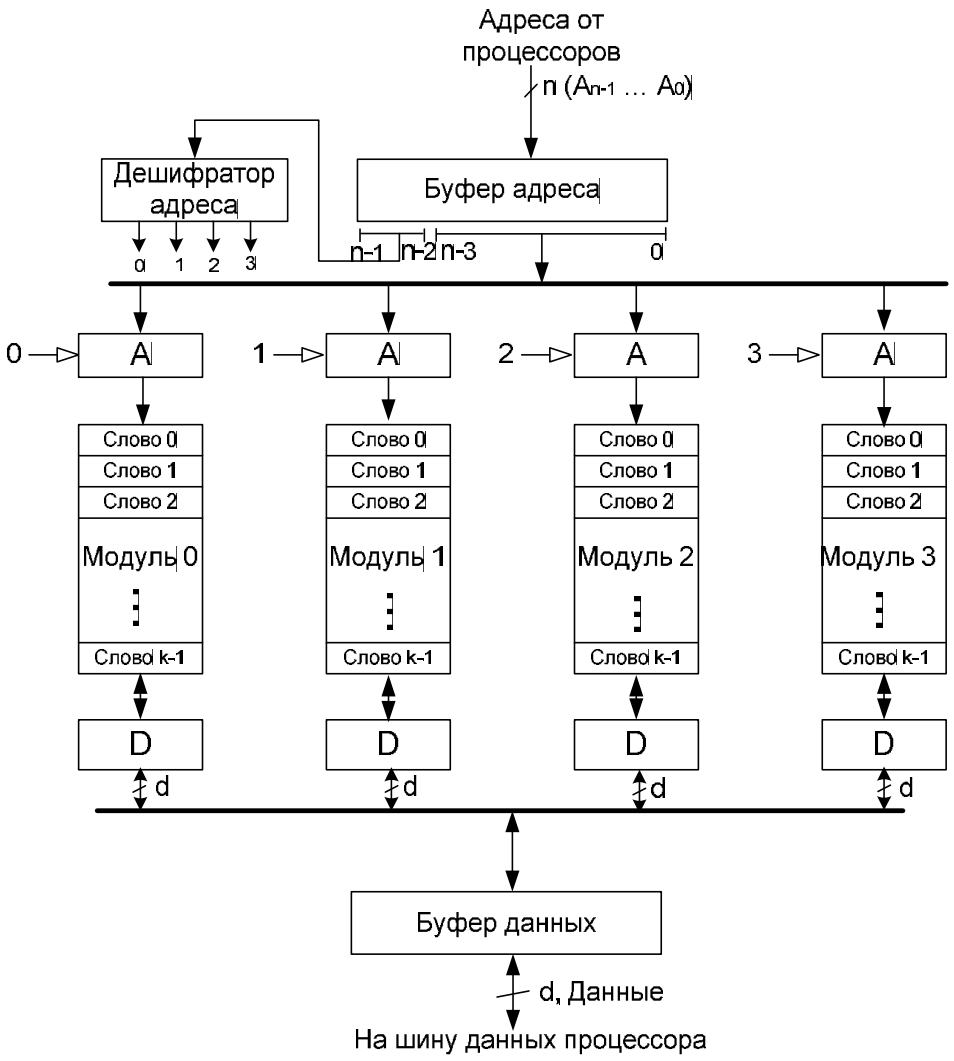


Рисунок 2.6- Схема банковой памяти

Многовходовая память.

Дуальная память. Часто процессоры работают с непрерывными потоками данных большой размерности. Поскольку время доступа к оперативной памяти составляет десятки процессорных тактов, то процессор вынужден часто простаивать, теряя реальную производительность. Особенно эта ситуация недопустима в системах реального времени, где должна быть задействована вся мощность процессора, чтобы успеть обработать быстротекущие процессы.

Один из способов повышения пропускной способности памяти заключается в использовании дуальной или переключающейся памяти.

Организуется такая память из двух или более буферных блоков сверхбыстродействующей памяти с произвольным доступом, которые соединяются так, что пока одни из буферов осуществляет обмен с оперативной памятью (ОП), другие остаются подключенным к центральному процессору (ЦП) и снабжают его данными. По окончании обменов ОП и процессора с подключенными к ним буферами, связи этих буферов взаимно переключаются. ЦП получает доступ к новой партии данных, а ОП получает возможность обмена данными с другими буферными модулями (рис.2.7).

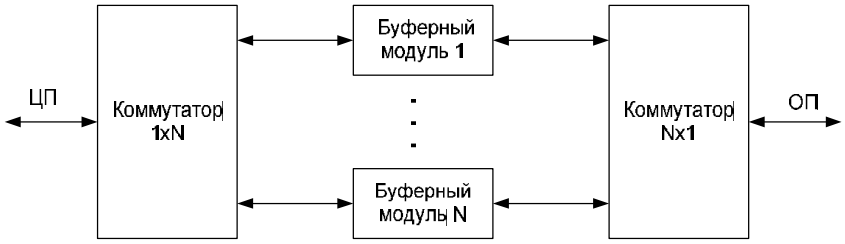


Рисунок 2.7- Структура дуальной памяти с N буферными модулями

Многопортовая память. Стандартная однопортовая память имеет по одной шине адреса, данных и управления и в каждый момент времени обеспечивает доступ к ячейке памяти только одному устройству. В отличие от стандартной в *n*-портовой памяти имеется *n* независимых наборов шин адреса, данных и управления, гарантирующих одновременный и независимый доступ к ОЗУ *n* устройствам. Данное свойство позволяет существенно упростить создание многопроцессорных, многомашинных и конвейерных вычислительных систем, где многопортовая память выступает в роли общей или совместно используемой памяти. В настоящее время серийно выпускаются двух- и четырехпортовые микросхемы, среди которых наиболее распространены первые. Поскольку архитектурные решения в обоих случаях схожи, дальнейшее изложение будет вестись применительно к двухпортовым ОЗУ.

Запоминающий элемент (ЗЭ) двухпортового ОЗУ (см. рис. 2.8) содержит шесть транзисторов, в отличие от стандартного ЗЭ, где число транзисторов

ограничивается четырьмя. Транзисторы Т3, Т4, Т5 и Т6 используются в качестве внутреннего коммутатора выходов запоминающего элемента с прямыми и инверсными линиями битов $P1$ ($\overline{P1}$) $P2$ и ($\overline{P2}$). Запоминающие элементы объединяются в матрицу, которую называют запоминающим массивом.

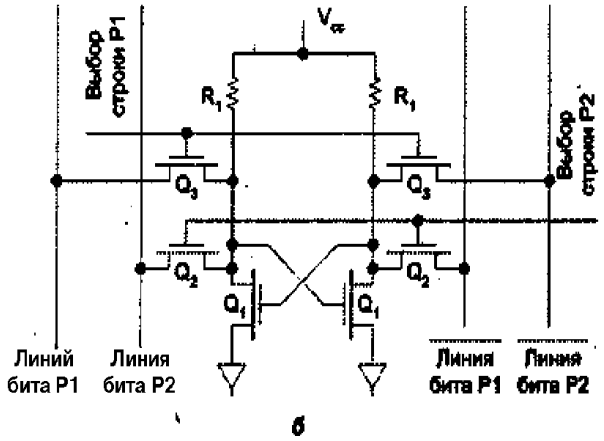


Рисунок 2.8- Запоминающий элемент двухпортового статического ОЗУ

Линии битов запоминающего массива образуют две шины данных модуля памяти, а линии выбора строк соответственно две шины адреса. Таким образом, в двухпортовой памяти имеются два набора адресных шин, шин данных и управляющих шин, каждая из которых обеспечивает доступ к общему массиву ЗЭ (рис. 2.9). Соответственно в ОЗУ с произвольным доступом имеется по два комплекта логических схем дешифрации адресов и буферных регистров данных (портов), которые обеспечивают одновременный доступ к её различным словам со стороны двух устройств (например, к видеопамяти со стороны процессора и видеоконтроллера) через два порта чтения/записи. Стоимость таких устройств высока, что ограничивает их практическое применение специализированными устройствами.

Поскольку двухпортовому ОЗУ свойственна симметричная структура, в дальнейшем наборы шин будем называть «левым» и «правым». В целом организация матрицы ЗЭ остается традиционной. Доступ к ячейкам возможен как через левую, так и через правую группу шин, причем, если адреса различны, никаких конфликтов не возникает. Проблемы потенциально возможны, когда левые и правые устройства одновременно обращаются по одному и тому же

адресу и хотя бы одно из этих устройств пытается выполнить операцию записи. В этом случае, если один из портов читает информацию, а другой производит запись в ту же ячейку, вероятно считывание недостоверной информации.

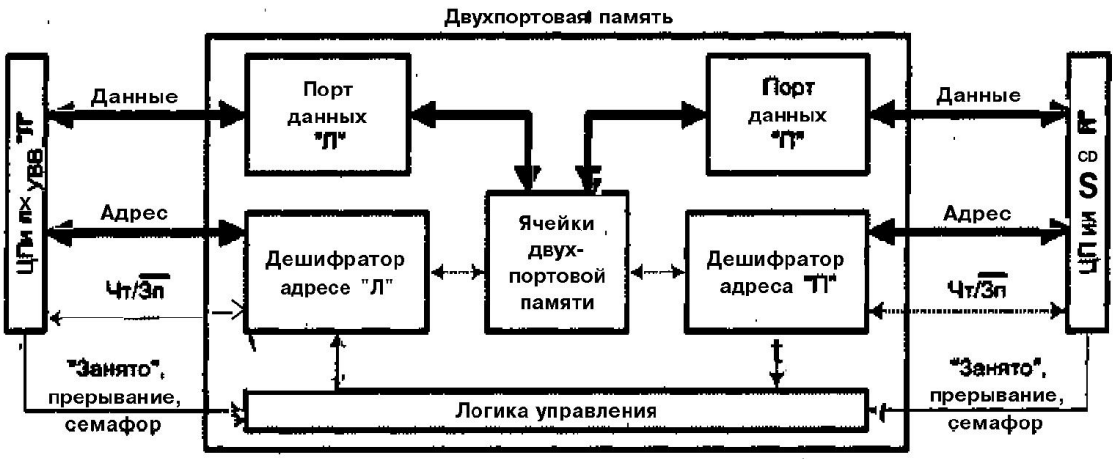


Рис. 5.13. Структура двухпортового ОЗУ

Рисунок 2.9- Функциональная организация двухпортовой памяти

При попытке единовременной записи в одну ячейку с двух направлений в нее может быть занесено неверное слово. Несмотря на то, что вероятность подобных ситуаций по оценкам не превышает 0,1%, такой вариант необходимо учитывать, для чего в двухпортовой памяти имеется схема арбитража.

Логика арбитража в микросхеме реализована аппаратными средствами. Схема обеспечивает формирование сигнала «Занято», запрещающего запись в ячейку со стороны другого порта, на котором адрес появится позже. Кроме того, арбитр производит выбор одного из входных портов при одновременном поступлении адресов одновременно на оба порта. Для этого арбитр снабжают двумя комплектами компараторов адреса, двумя буферами задержки (левые и правые) и триггером-защелкой для фиксации пары сигналов «блокировка» (левый и правый). Триггер-защелка выполняет роль битового семафора, обеспечивающего защиту общего ресурса - ячейки памяти - от одновременного доступа к ней со стороны двух запрашивающих устройств. Семафор при захвате ячейки одним из устройств, блокирует другое

запрашивающее устройство на время выполнения обмена с памятью первого устройства.

Ортогональная память.

Структурная организация памяти этого типа приведена на рис.2.10. В такой памяти данные, расположенные в запоминающих элементах (ЗЭ) строки, рассматриваются как элементы единого информационного сообщения - слова. Доступ к данным в ортогональной памяти, во-первых, обеспечивается указанием номера (адреса) строки. В этом случае обеспечивается параллельный доступ ко всем ЗЭ, расположенным в строке с указанным номером. Таким образом, доступ к данным в памяти выполняется последовательно по словам. Во-вторых, обеспечена возможность доступа к ЗЭ, расположенным в одном столбце. Запоминающие элементы являются битовыми и могут объединятся в байтовые ячейки. В первом случае данные столбца называют битовым срезом, во втором – байтовым срезом. Доступ к данным столбца обеспечивается указанием номера (адрес) столбца. В итоге, в памяти с ортогональной организацией выполняется высокоскоростной доступ к данным последовательно по словам или последовательно по срезам (битовым или байтовым).

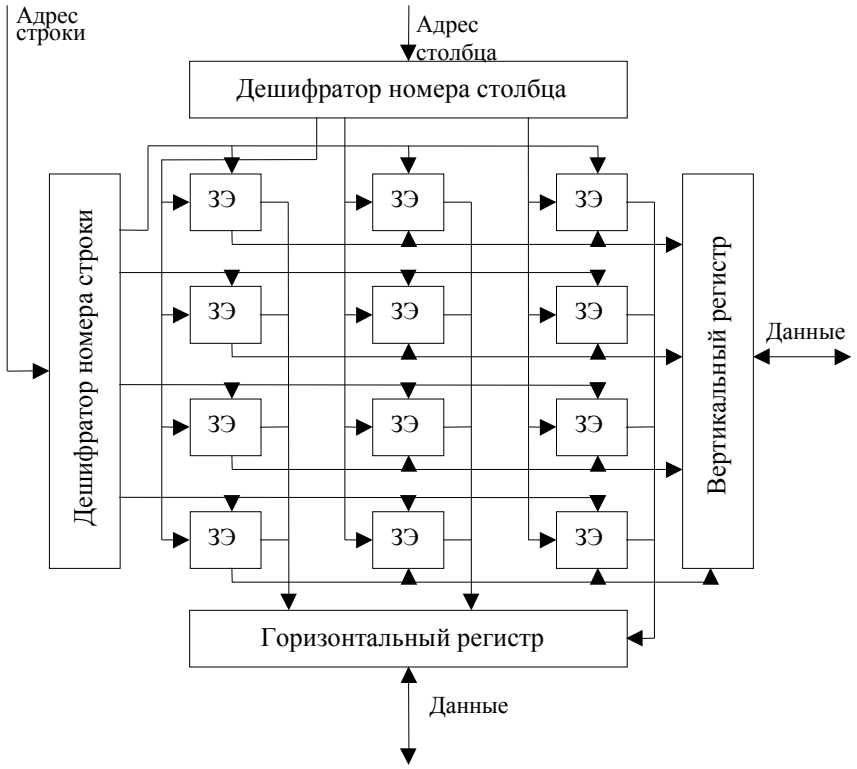


Рисунок 2.10 – Структура ортогональной памяти

Ассоциативная память.

При использовании адресной памяти операнды записываются в память и считываются из памяти поочередно (последовательно), в то время как при решении многих задач эти операции целесообразно выполнять одновременно (параллельно). С целью упрощения поиска операндов и обеспечения возможности одновременного доступа ко многим ячейкам памяти используется адресация по содержанию. Ввиду высокой стоимости и технической сложности память с такой адресацией в ВС используется в качестве локальной памяти и применяется для выполнение операций свертки, поиска и сортировки.

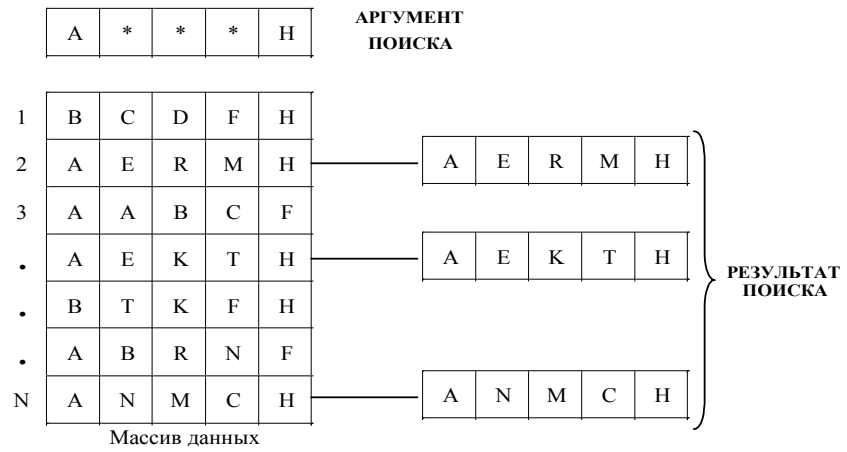


Рисунок 2.11- Принцип адресации по содержанию

Сущность принципа адресации по содержанию заключается в следующем (рис.2.11). Имеется массив данных емкостью N слов. Требуется найти в массиве все слова, которые начинаются с символа "А" и кончаются символом "Н". В этом случае аргументом поиска является слово А***Н, где значком * отмечены разряды, не влияющие на результат поиска. Запоминающий массив на аппаратном уровне строится таким образом, что на выходе ячеек памяти, содержимое которых совпадает со значением поступившего аргумента поиска, появляется сигнал - указатель совпадения. Далее по выработанным сигналам совпадения выполняется выборка ячеек памяти.

Реализация ассоциативного доступа к данным может быть организована одним из двух способов:

1) *программным, основанным на распределении памяти в зависимости от содержания данных и реализуемым с помощью программных средств (метод хеширования. При этом предполагается, что информация хранится в обычном запоминающем устройстве произвольного доступа со стандартной системой адресации;*
2) *аппаратным, опирающимся на применение специальных аппаратных средств, предназначенных для хранения и ассоциативного поиска элементов данных.*

В методах программного поиска доступ к любой информационной единице (записи, документу, массиву или элементу данных и т. п.) осуществляется непосредственно: либо по специальному ключу, либо по некоторому фрагменту самой информационной единицы. Такие методы называют методами хеширования (или другие названия их: адресация в разброс, адресация с перемешиванием, ключевые преобразования и т.п.).

Аппаратная реализация ассоциативной памяти должна обеспечить, как минимум, следующие функции:

- *осуществлять операцию поиска элемента по некоторому аргументу;*
- *выполнять считывание (запись) информации;*
- *возвращать в исходное состояние память фиксации реакций;*
- *анализировать многократность совпадения и осуществлять выборку по совпадению.*

В параллельных ассоциативных ЗУ (АЗУ) процесс поиска данных по содержанию организуется следующим образом. Каждая ячейка памяти модуля памяти АЗУ обеспечивает выполнение функций приема, хранения данных, сравнения хранимой информации с аргументом поиска и выработку сигналов о результате сравнения (рис.2.12).

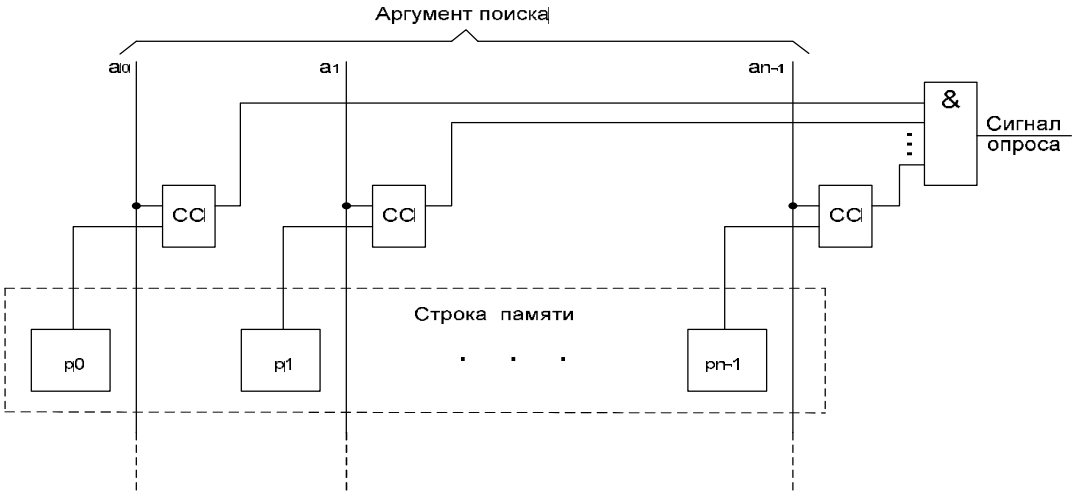


Рисунок 2.12- Упрощенная функциональная схема ячейки АЗУ: СС- схемы сравнения; $p_{0,}...,p_{n-1}$- разряды строки памяти, $a_0,...,a_{n-1}$- разряды аргумента поиска

Модуль памяти организован таким образом, что на каждом цикле работы аргумент поиска поступает параллельно во все ячейки АЗУ. В результате в модуле памяти АЗУ выполняется массовая операция сравнения содержимого ячейки памяти с аргументом поиска и установка - сигналов указателей о совпадении на выходе. Ассоциативную память иногда называют ассоциативным процессором, поскольку она не только хранит данные, но производит их обработку.

Как правило, ячейки памяти АЗУ, расположенные в одной строке, рассматриваются как слово. В операции сравнения могут участвовать не все разряды строки ($p_0,...,p_{n-1}$), а только та их часть, которая активизируется управляющим словом, которое называют вектором-маской или просто маской. Например, разряд ячейки памяти переводится в активное состояние, если соответствующий ей разряд маски равен единице, в противном случае этот разряд в операции сравнения не участвует.

АЗУ (рисунок 2.13) подключается к шинам адреса, данных центрального процессора обычным образом (в виде ускорителя для операций поиска). Перед выполнением операции в модуль памяти (МП) вводится

массив данных, для чего он переводится в режим прямоадресуемой памяти. Центральный процессор выставляет адрес ячейки на шину адреса, который используется для выбора (селектирования) соответствующего модуля памяти (если их несколько). Затем адрес дешифрируется логическими схемами АЗУ, указывая выходным сигналом дешифратора в какую ячейку следует осуществить запись. Слово данных по шине данных помещается в адресуемую ячейку по входной шине данных.

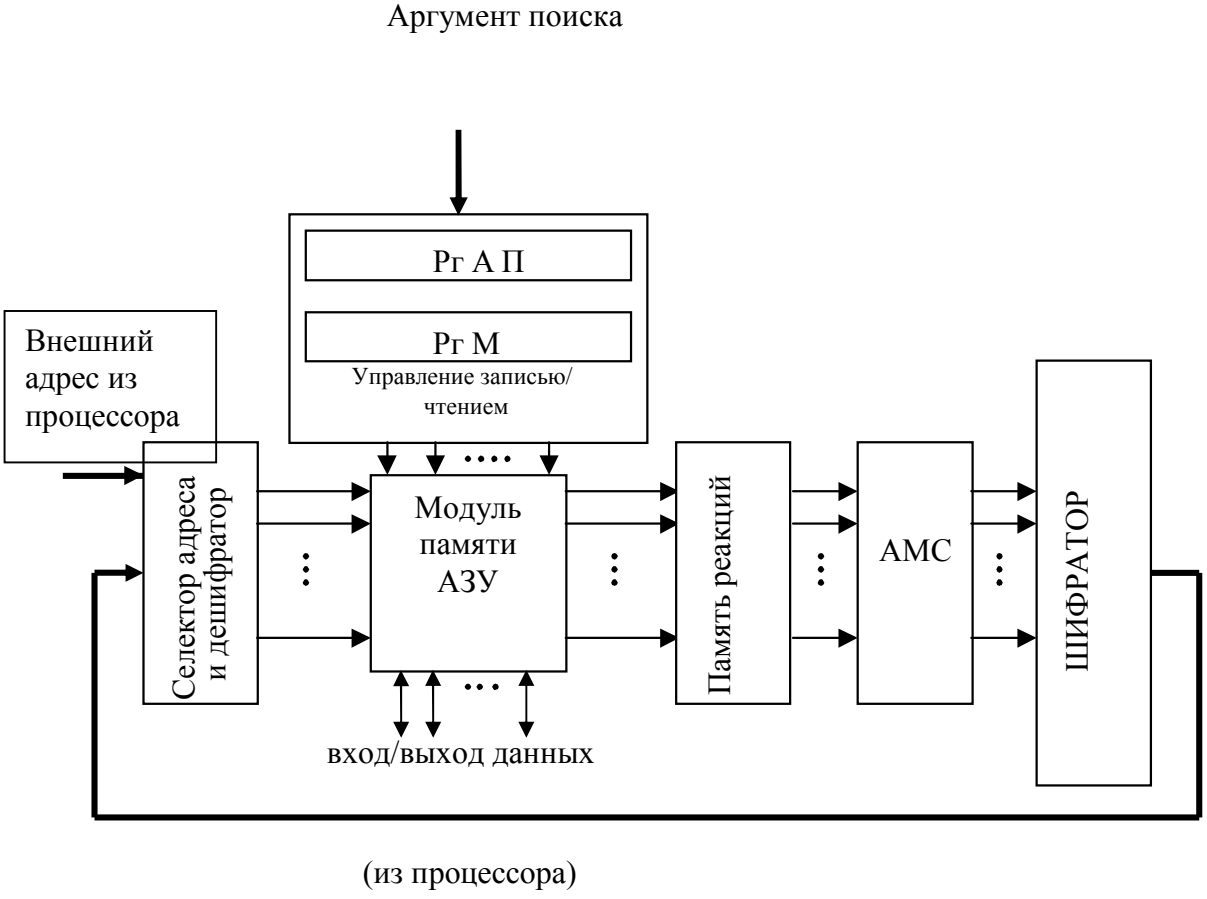


Рис.2.13- Структура параллельного ассоциативного ЗУ

При выборке данных из АЗУ с параллельной организацией в регистр аргумента поиска (РгАП) вводится аргумент поиска, в регистр маски (РгМ) – слово маски. Аргумент поиска и слово маски также формируются центральным процессором. Данные из РгАП и РгМ параллельно поступают в

ячейки памяти АЗУ, выполняется операция сравнения, по окончании которой на выходах тех строк модуля памяти АЗУ, в которых произошло совпадение с заданным аргументом поиска, вырабатываются сигналы опроса. Эти сигналы фиксируются в памяти реакций и в виде унитарного кода подаются в анализатор многократных совпадений (АМС). В АМС выполняется их приоритетный разбор, по результатам которого формируется последовательность выдачи сигналов в шифратор, который соответственно преобразует унитарный код в двоичный код адреса. Этот адрес используется для выборки данных из тех ячеек АЗУ, в которых произошло совпадение.

В принципе сигналы с АМС можно подавать непосредственно на адресные шины модуля памяти АЗУ, что обеспечило бы доступ ко всем "откликнувшимся" ячейкам одновременно. Однако на практике модуль памяти изготавливается из стандартных микросхем памяти с встроенными дешифраторами адреса. Последнее позволяет использовать АЗУ и в режиме адресуемой памяти для ввода массива данных из центрального процессора.

Ввод данных в АЗУ может производиться в «разбег» после предварительного поиска в модуле памяти АЗУ свободных ячеек памяти. С этой целью в ячейке предусматривается специальный бит - бит занятости, который устанавливается в единицу при вводе данных в неё слова данных. Если в модуле памяти АЗУ выявлено несколько свободных строк, ввод данных выполняется по приоритету, определяемому в АМС.

В последовательно-поразрядных или последовательно-побайтных АЗУ (рис. 2. 14) модуль памяти реализуется в виде ортогонального ОЗУ, т. е. с возможностью выборки как по строкам, так и по столбцам. В РгАП вводится слово аргумента поиска, в РгМ – слово маски. Далее выполняется опрос первого столбца модуля памяти АЗУ. Данные поступают на первые входы блока схем сравнения, на вторые входы которого поступает соответствующая часть аргумента поиска из РгАП и часть слова маски из РгМ. Результаты сравнения фиксируются в памяти реакций. Затем выполняется сдвиг

регистров РгАП и РгМ, после чего выполняется опрос второго столбца ячеек памяти.

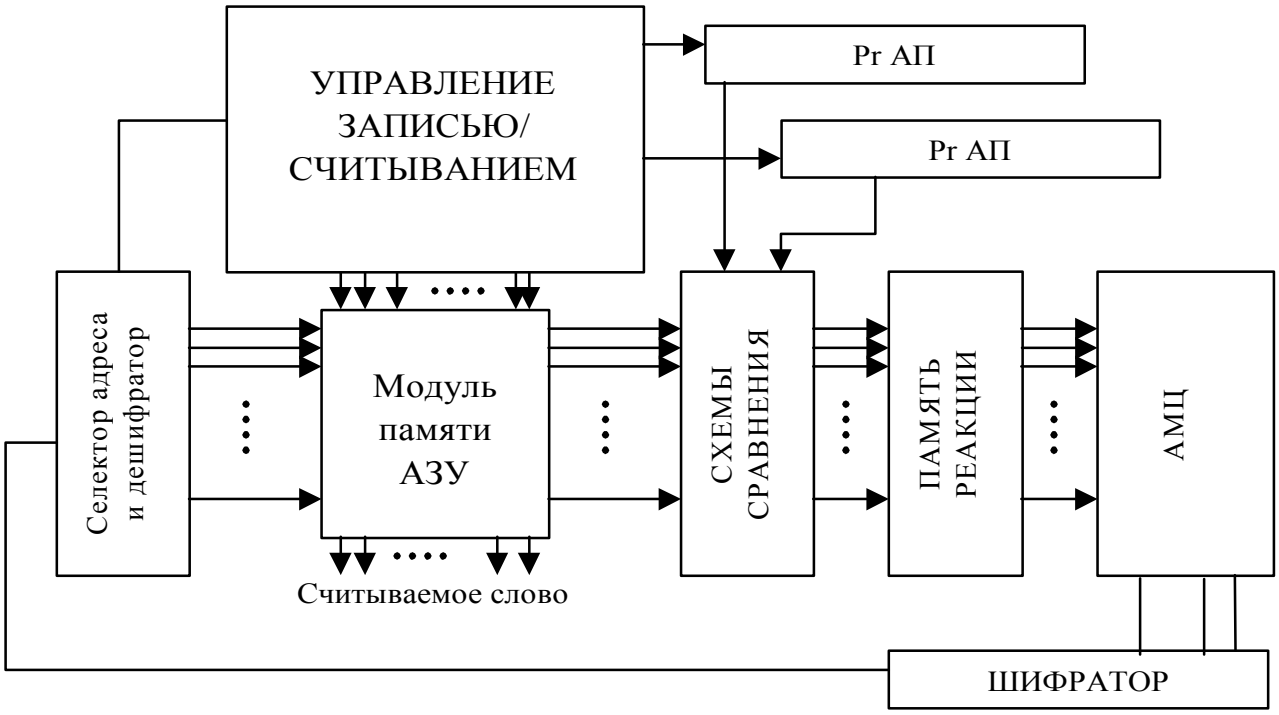


Рис.2.14

Сигналы совпадения фиксируются элементами памяти реакции только в том случае, если в эти же элементы поступали и сигналы о совпадении на предыдущем такте. В случае если на текущем такте сигнала совпадения нет, а на предыдущем он был, то соответствующий элемент памяти реакции фиксирует отсутствие совпадений. Описанный процесс повторяется до тех пор, пока не будет выполнен опрос последнего столбца модуля памяти АЗУ. В итоге в памяти реакций будут зафиксированы сигналы - указатели строк ячеек памяти, в которых произошло совпадение с аргументом поиска. В остальном АЗУ этого типа работает полностью аналогично параллельному АЗУ. Таким образом, параллельные АЗУ независимо от емкости обеспечивают выборку данных, удовлетворяющих заданному аргументу поиска, за один цикл памяти. Однако при этом требуется большой объем схемной логики в каждой ячейке памяти, что приводит к увеличению стоимости устройства. В АЗУ с последовательно-поразрядной организацией

выработка данных обеспечена за *n* циклов памяти, где *n* - число разрядов в строке модуля памяти. Но объем схемной логики для выполнения операций сравнения с аргументом поиска значительно меньше. На практике применяют оба типа АЗУ, причем выбор того или иного типа определяется требованиями быстродействия и стоимости.

Кэш-память

Кэш-память предназначена для уменьшения времени доступа к данным, размещенным в основной (оперативной) памяти (ОП). В отличие от оперативной памяти, где адреса ячеек фиксированы и располагаются в последовательных ячейках, кэш хранит данные в любом порядке, но вместе с их адресом.

В структуру кэш-памяти, как показано на рис.1.18, входят массив данных и справочник. Данные в кэш размещаются строками длиной в несколько слов (обычно 4 - 8). В строке хранится также начальный адрес данных, соответствующий адресу данных в оперативной памяти, и необходимыми управляющими битами (рисунок 1.19). Оперативная память также условно делится на строки размером, равным по величине размеру поля данных кэш. Чтение данных из памяти производится целой строкой за одну транзакцию. Запись в отличие от чтения производится обычно пословно. Для выполнения таких операций контроллер ОП и кэш имеют соответствующие аппаратные средства.

Процессор при выполнении команды обращается в первую очередь к кэш-памяти. Если данные обнаружены в кэш, т. е. адрес, выставленный процессором, обнаружен в кэш, то получен быстрый ответ. При отсутствии в кэш требуемой строки она копируются в кэш из основной памяти. Одновременно требуемое слово процессор из этой строки заносится в процессор (медленный ответ).

Эквивалентное время обращения к памяти t_{on}, определенное по формуле (1.1), равно

$$t_e = t_k + p_{miss} t_{m,} \qquad (1.2)$$

где t_k— время обращения к кэш, t_m— время обращения к основной памяти, p_{miss} — вероятность кэш – промаха, т.е. вероятность того, что требуемая информация отсутствует в кэш. Обычно t_k меньше t_m на порядок или более и при достаточном уменьшении p_{miss} можно достичь t_e приблизительно равным t_k. По мере повышения быстродействия логических элементов повышается быстродействие и процессора. При этом сокращается и время обращения к кэш.

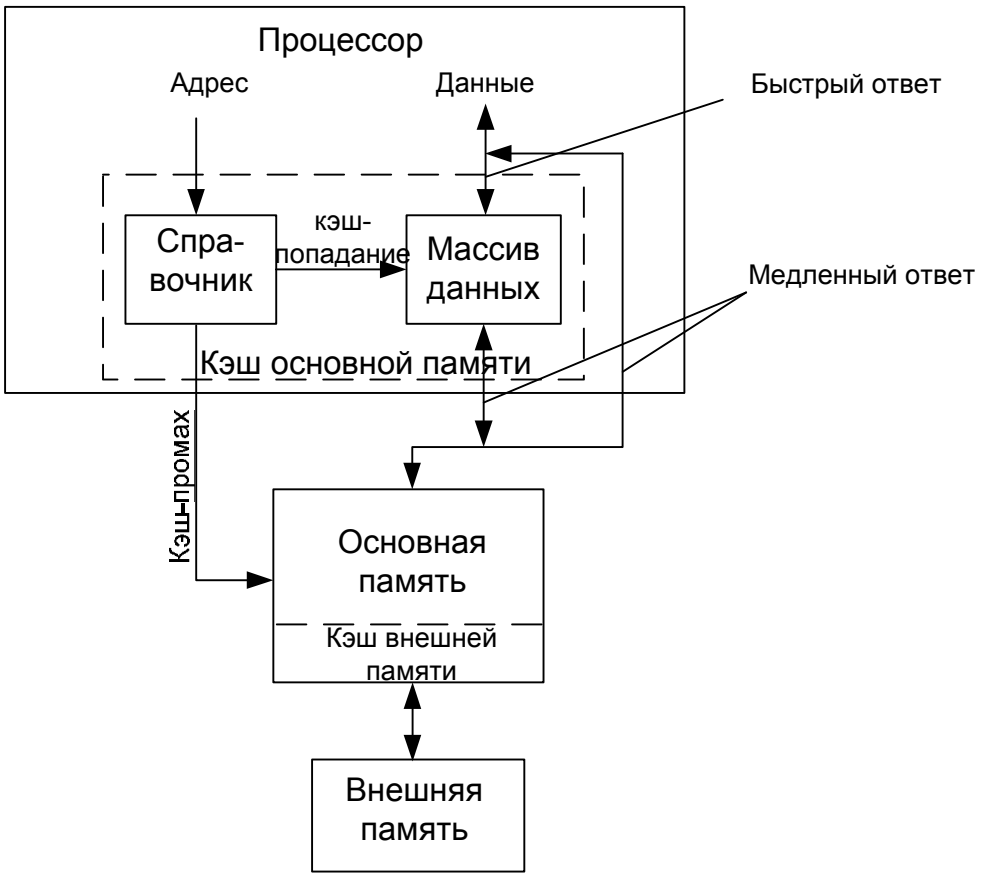


Рис.1.18. Структурная организация и место кэш в подсистеме памяти

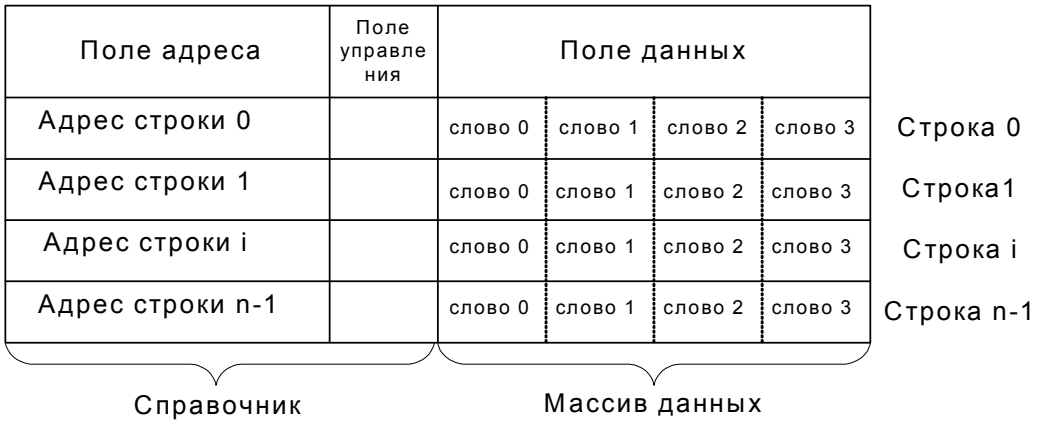


Рисунок 1.19– Информационная структура кэш

Однако в силу постоянно растущих требований к увеличению объема памяти время обращения к основной памяти существенно уменьшить трудно, и разница между t_k и t_m увеличивается, что приводит к снижению эффективности кэш-памяти. Преодолевается эта проблема за счет увеличения емкости кэш-памяти или применения многоуровневой структуры кэш.

Подобный подход к повышению пропускной способности может, например, использоваться для повышения скорости обмена данными между основной памятью и накопителями внешней памяти (ВП). Используя кэш - память емкостью несколько десятков мегабайт, реализованную в виде отдельных микросхем памяти или выделенную в специальной области оперативной памяти (электронный диск), можно увеличить эквивалентную скорость обращения к магнитным дискам примерно в 2—10 раз. Таким образом, кэширование является универсальным методом, пригодным для ускорения доступа к оперативной памяти, к накопителям на дисках и к другим видам запоминающих устройств.

Виртуальную память также можно считать одним из вариантов реализации принципа кэширования данных, при котором оперативная память выступает в роли кэш по отношению к внешней памяти — жесткому диску. И в этом случае кэширование используется не только для того, чтобы уменьшить время доступа к информации, но и для того, чтобы использовать диск для временного хранения неиспользуемых некоторое время программ и данных с

целью освобождения места для активных процессов. В результате наиболее интенсивно используемые программы и данные находятся в оперативной памяти, остальная же информация хранится в более объемной и менее дорогостоящей внешней памяти.

Одна из возможных схем кэширования представлена на рис.1.19. Содержимое кэш-памяти представляет собой совокупность записей обо всех загруженных в нее строках из основной памяти. Каждая запись о строке включает в себя три поля:

- адрес, который эта строка имеет в основной памяти;
- дополнительную информацию, которая используется для реализации алгоритма замещения данных в кэш и обычно включает признак модификации, признак действительности данных и др.
- значение элементов данных (слов);

Первое поле зачастую называют полем тегов, второе – полем управления, третье - полем данных.

При каждом обращении к основной памяти по физическому адресу просматривается содержимое кэш-памяти с целью определения, не находятся ли там нужные данные. Кэш-память не является адресуемой, поэтому поиск нужных данных осуществляется по содержимому, т.е. по взятому из запроса значению адреса слова в оперативной памяти. Далее возможен один из двух вариантов развития событий:

- если строка, содержащая слово данных обнаруживаются в кэш-памяти, т.е. произошло кэш-попадание (cache-hit), слово считывается из нее и результат передается процессору;
- если данные отсутствуют в кэш-памяти, то есть произошел кэш-промах (cache-miss), строка с запрашиваемыми данными считывается из основной памяти, копируется в кэш-память и одновременно с этим нужное слово передается процессору.

Среднее время доступа к данным в системе с кэш-памятью линейно зависит от вероятности попадания в кэш. Очевидно, что использование кэш-памяти имеет смысл только при высокой вероятности кэш-попадания.

Вероятность обнаружения данных в кэш зависит от разных факторов, таких, например, как объем кэш, объем кэшируемой памяти, алгоритм замещения данных в кэш, особенности выполняемой программы, время ее работы, уровень мультипрограммирования и других особенностей вычислительного процесса. Тем не менее, в большинстве реализаций кэш-памяти процент кэш-попадании оказывается весьма высоким — свыше 90 %. Такое высокое значение вероятности нахождения данных в кэш-памяти объясняется наличием у программ и данных объективных свойств, называемых локальностью ссылок, которая подразделяется на пространственную и временную локальность.

Временная локальность заключается в том, что если произошло обращение по некоторому адресу, то следующее обращение по тому же адресу с большой вероятностью произойдет в ближайшее время. Пространственная локальность заключается в том, что если произошло обращение по некоторому адресу, то с высокой степенью вероятности в ближайшее время произойдет обращение к соседним адресам.

Основываясь на свойстве временной локальности, программы и данные, только что считанные из основной памяти, размещают в запоминающем устройстве быстрого доступа, предполагая, что скоро они опять понадобятся. Вначале работы системы, когда кэш-память еще пуста, происходит процедура её заполнения. В этот промежуток времени практически каждый запрос процессора к основной памяти выполняется по следующей схеме: просмотр кэш, констатация промаха, чтение данных из основной памяти, передача результата источнику запроса с одновременным копированием данных в кэш. Затем, по мере заполнения кэш, в полном соответствии со свойством временной локальности возрастает вероятность обращения к данным, которые уже были использованы на предыдущем этапе работы системы, то есть к

данным, которые содержатся в кэш и могут быть считаны значительно быстрее, чем из основной памяти.

Свойство пространственной локальности также используется для увеличения вероятности кэш-попадания: как правило, в кэш-память считывается не одно слово, к которому произошло обращение, а целый блок данных, расположенный в основной памяти в непосредственной близости с данным словом. Поскольку при выполнении программы очень высока вероятность, что команды выбираются из памяти последовательно одна за другой из соседних ячеек, то имеет смысл загружать в кэш-память целый фрагмент программы. Аналогично если программа ведет обработку некоторого массива данных, то ее работу можно ускорить, загрузив в кэш часть или даже весь массив данных. При этом учитывается высокая вероятность того, что значительное число обращений к памяти будет выполняться к адресам массива данных.

Проблема согласования данных

В процессе функционирования вычислительной системы содержимое кэш-памяти постоянно обновляется. Время от времени устаревшие данные из нее вытесняются для освобождения места под новые данные. Реально вытеснение означает либо простое объявление свободной соответствующей области кэш-памяти (сброс бита действительности), если вытесняемые данные за время нахождения в кэш не были изменены, либо в дополнение к этому копирование данных в основную память, если они были модифицированы. Алгоритм замены данных в кэш-памяти существенно влияет на ее эффективность. Алгоритм должен, во-первых, быть максимально быстрым, чтобы не замедлять работу кэш-памяти, а во-вторых, обеспечивать максимально возможную вероятность кэш-попаданий. Поскольку из-за непредсказуемости вычислительного процесса ни один алгоритм замещения данных в кэш-памяти не может гарантировать идеальный результат,

разработчики ограничиваются рациональными решениями, которые, по крайней мере, не сильно замедляют работу кэш.

Наличие в ЭВМ двух копий данных - в основной памяти и в кэш - порождает проблему согласования данных. Если происходит запись в основную память по некоторому адресу, а содержимое этой ячейки находится в кэш, то в результате соответствующая запись в кэш становится недостоверной. Рассмотрим два подхода к решению этой проблемы.

Сквозная запись (write through).

При каждом запросе к основной памяти, в том числе и при записи, просматривается кэш. Если строка с запрашиваемым адресом отсутствует в кэш (для этого просматривают поле тегов), то запись выполняется только в основную память. Если же адрес, по которому выполняется обращение, находится и в кэш, то запись производится одновременно в кэш и основную память. Такая операция по времени равносильна обращению в основную (оперативную) память и не дает большого выигрыша от применения кэш-памяти.

Обратная запись (write back).

Аналогично при возникновении запроса к памяти выполняется просмотр кэш, и если строка с запрашиваемым адресом там отсутствует, то запись выполняется только в основную память. В противном же случае запись производится только в кэш-память, при этом в поле управления делается специальная отметка (признак модификации), которая указывает на то, что при вытеснении этих данных из кэш необходимо переписать их в подходящий момент в основную память, чтобы обновить устаревшее содержимое основной памяти.

Очевидно, что в этом случае в соответствии с принципами временной и пространственной локальности, обращения будут происходить преимущественно к кэш-памяти, а малая их доля упадёт на основную память. Поэтому временная

эффективность метода обратной записи высокая, хотя требует применения более сложных алгоритмов согласования данных. Контроллер кэш-памяти осуществляет копирование модифицированной строки при любых замещениях строки. Модифицированные данные могут выгружаться не только при освобождении места в кэш-памяти для новых данных, но и в тех моментах времени, когда шинный интерфейс не занят процессором или другими устройствами. В некоторых алгоритмах замещения предусматривается первоочередная выгрузка модифицированных, или, как еще говорят, «грязных» данных.

Механизм выполнения запросов в системах с двухуровневой кэш-памятью

Когда выполняется транзакция записи, кэш просматривается только с целью согласования его содержимого и содержимого основной памяти. Если происходит промах, то запросы на запись удовлетворяются только в оперативной памяти, не вызывая никаких изменений в содержимом кэш (рисунок 1.18). В некоторых же реализациях кэш-памяти при отсутствии данных в кэш они копируются туда из основной памяти независимо от того, выполняется запрос на чтение или на запись.

В соответствии с описанной логикой работы кэш-памяти следует, что при возникновении запроса сначала просматривается кэш, а затем, если произошел промах, выполняется обращение к основной памяти. Однако часто реализуется и другая схема работы кэш: поиск в кэш и в основной памяти начинается одновременно, а затем, в зависимости от результата просмотра кэш, операция в основной памяти либо продолжается, либо прерывается.

Согласование (когерентность) данных требует значительных временных затрат. Проблема заключается в том, что при кэш-промахе во время выполнения операции чтения процессор приостанавливается на время, пока данные не будут получены из оперативной памяти. Это время может составлять

несколько десятков или даже сотню или более тактов процессора. Чтобы избежать задержек во многих вычислительных системах используется двухуровневое кэширование (рис.1.20). Кэш первого уровня имеет меньший объем и более высокое быстродействие, чем кэш второго уровня. Кэш второго уровня играет роль основной памяти по отношению к кэш первого уровня.

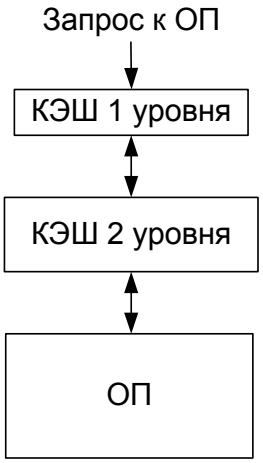


Рис.1.20 Структура памяти с двухуровневым кэш.

Алгоритм выполнения запроса на чтение в системе памяти с двухуровневым кэш следующий. Сначала делается попытка обнаружить данные в кэш первого уровня. Если произошел промах, поиск продолжается в кэш второго уровня. Если же нужные данные отсутствуют и здесь, тогда происходит считывание данных из основной памяти. Очевидно, что время доступа к данным оказывается минимальным, когда кэш-попадание происходит уже на первом уровне, несколько большим - при обнаружении данных на втором уровне. И самое большое время доступа составляет обращение к оперативной памяти, если нужных данных нет ни в том, ни в другом кэш. При считывании данных из оперативной памяти происходит их копирование в кэш второго уровня, а если данные считываются из кэш второго уровня, то они копируются в кэш первого уровня.

При работе такой иерархической организованной памяти необходимо обеспечить непротиворечивость данных на всех уровнях. КЭШ разных уровней могут согласовывать данные разными способами. Пусть, например, кэш первого уровня использует сквозную запись, а кэш второго уровня — обратную запись. (Именно такая комбинация алгоритмов согласования применена в процессоре Pentium при одном из возможных вариантов его работы.)

Если данные обнаружены в кэш первого уровня, то вступает в силу алгоритм сквозной записи: выполняется запись в кэш первого уровня и передается запрос на запись в кэш второго уровня, играющий в данном случае роль основной памяти. Т.е. в этом случае согласование данных между первым и вторым уровнями происходит автоматически. Согласование данных с оперативной памятью происходит на втором уровне кэш, при этом кэш первого уровня остается свободной для выполнения других запросов процессора. Запись в кэш второго уровня в соответствии с алгоритмом обратной записи, принятом на данном уровне, сопровождается установкой признака модификации. Хотя при этом никакой записи в оперативную память не производится, однако в подходящий момент, в соответствии с принятым алгоритмом замещения, измененные данные перемещаются в основную память. Такой способ кэширования существенно повышает производительность вычислительной системы.

Рассмотренные в данном разделе проблемы кэширования охватывают только такой класс систем организации памяти, в котором на каждом уровне имеется одно кэширующее устройство. Существует и другой класс систем памяти, главной отличительной особенностью которого является наличие нескольких кэш одного уровня. Этот вариант характерен для мультипроцессорных, многомашинных и распределенных систем обработки информации.

Одна из существенных проблем, возникающих при управлении кэш, связана с коллективным использованием основной памяти. Когда, как

показано на рис. 1.21, основная память совместно используется несколькими процессорами, каналами обмена с ВЗУ и т. д., возможно возникновение конфликтов между кэш-памятями, между кэш-памятью и основной памятью, поэтому необходимы меры, обеспечивающие когерентность кэш.

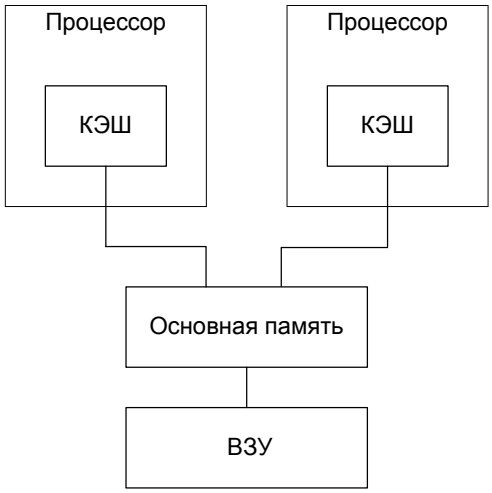


Рис.1.21. Совместное использование основной памяти

Если в мультипроцессорной системе применяется сквозная запись в память, то при записи информации в какую-либо строку её адрес сообщается всем другим процессорам (а не только тому, в кэш которого выполняются изменения данных). В этих других процессорах проверяется, нет ли данной строки в их кэш и, если есть, то в дальнейшем информация этой строки во всех кэш считается недостоверной. При использовании сквозной записи управление производится таким образом, что обновление строки происходит только в кэш одного процессора, а в остальных процессорах эта строка считается недостоверной. При обращении для считывания информации к строке, содержимое которой обновлялось в кэш другого процессора, это обновленное содержимое передается в основную память, затем строка переносится в кэш, к которому имело место обращение для считывания, после чего происходит считывание.

1.1.2. Виртуальная память

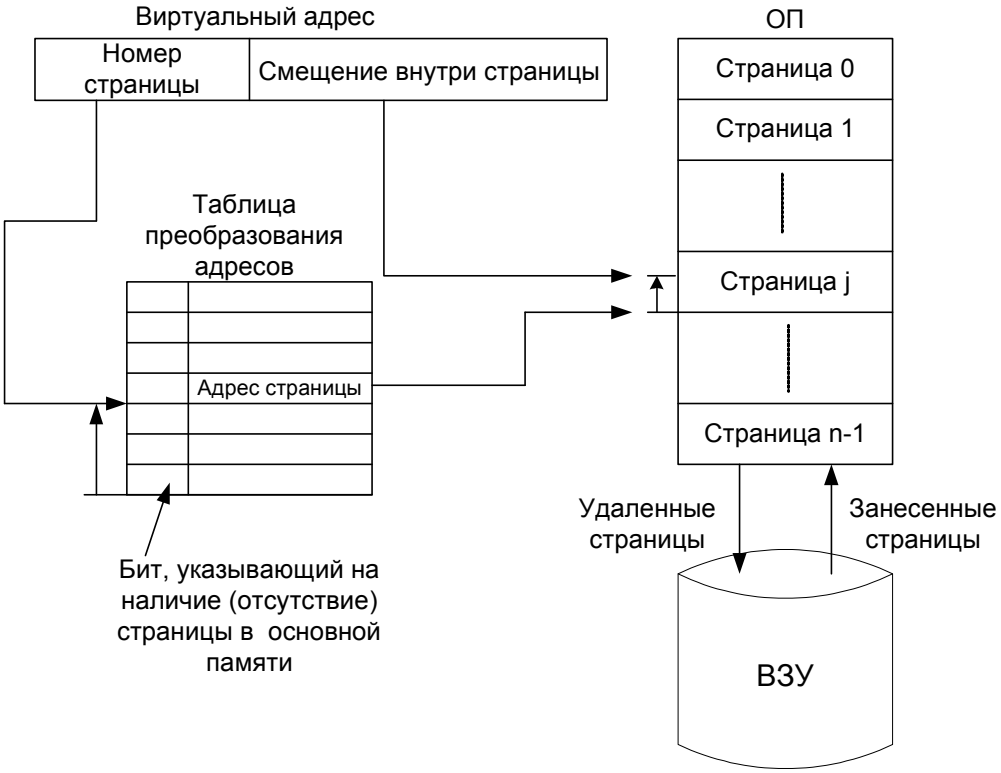
Система виртуальной адресации

Виртуальная память представляет собой единое адресное пространство, в котором физическая ограниченность емкости основной памяти скрыта от программиста. Таким образом, для программиста создается видимость произвольной адресации с отсутствием ограничений на емкость используемой памяти, что значительно облегчает программирование. Кроме того, использование виртуальной организации памяти способствует повышению взаимозаменяемости программ между вычислительными системами.

Реально существующую основную память называют физической, а ее адреса - физическими, логическую память - виртуальной, а ее адреса - виртуальными (логическими). Соответствие между физическими и виртуальными адресами устанавливается совместно аппаратными средствами ЭВМ и ее операционной системой. Обычно виртуальное адресное пространство размещается во внешней памяти, например на магнитных дисках. Часть этого пространства, необходимая для выполнения программ в данный момент, копируется в основной памяти.

Для реализации виртуальной памяти необходимо разделить все адресное пространство памяти на части и организовать соответствующий обмен между основной и внешней памятью. При этом память разбивается на страницы или сегменты.

При разбиении на страницы виртуальное и реальное адресные пространства делятся на части фиксированной длины, называемые



страницами.

Рис.1.3. Страничная организация памяти.

Адрес каждой страницы виртуального пространства ставится в соответствие адресу страницы физического адресного пространства. Взаимосвязь между адресами обоих типов устанавливается таблицей преобразования адресов (таблицей страниц), пример которой приведен на рис.1.3.

Перенос страницы виртуального пространства в основную память называется загрузкой страницы (подкачкой страницы в оперативную память), а обратное действие - удалением страницы (откачкой страницы из оперативной памяти).

Отметить достоинства и недостатки!

Сегментация

Сегментацией называется разделение адресного пространства памяти на части (сегменты) по логическим признакам, устанавливаемым программистом. Обычно величина сегмента соответствует объему

программы или подпрограммы и в отличие от страницы имеет переменную длину.

Виртуальный адрес состоит в этом случае из номера сегмента и относительного адреса в пределах сегмента; он преобразуется в физический адрес по таблице сегментов (рис.1.4). Поскольку в каждом сегменте адресное пространство является линейным, виртуальное адресное пространство в целом оказывается двумерным. С учетом того, что сегмент является логической единицей, можно организовывать защиту информации и управление для коллективного использования сегментированной информации.

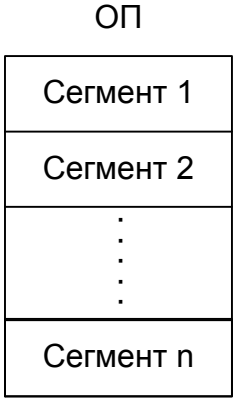


Рис.1.4.Сегментная организация памяти.

Сегментно- страничная организация памяти

Расширение виртуального пространства влечет за собой увеличение таблицы страниц. Одним из способов устранения этого неудобства служит многоуровневое разбиение на страницы. Суть этого разбиения состоит в том, что одномерное виртуальное пространство подразделяется на два уровня - сегментов и страниц, а преобразование виртуального адреса производится по двухуровневой таблице. Это дает возможность обойтись без ведения таблицы неиспользуемых страниц и, следовательно, экономит объем памяти, выделяемый для таблицы страниц. Сегмент в этом случае не является полностью двумерным пространством, но при необходимости в процессе использования его можно сделать двумерным.

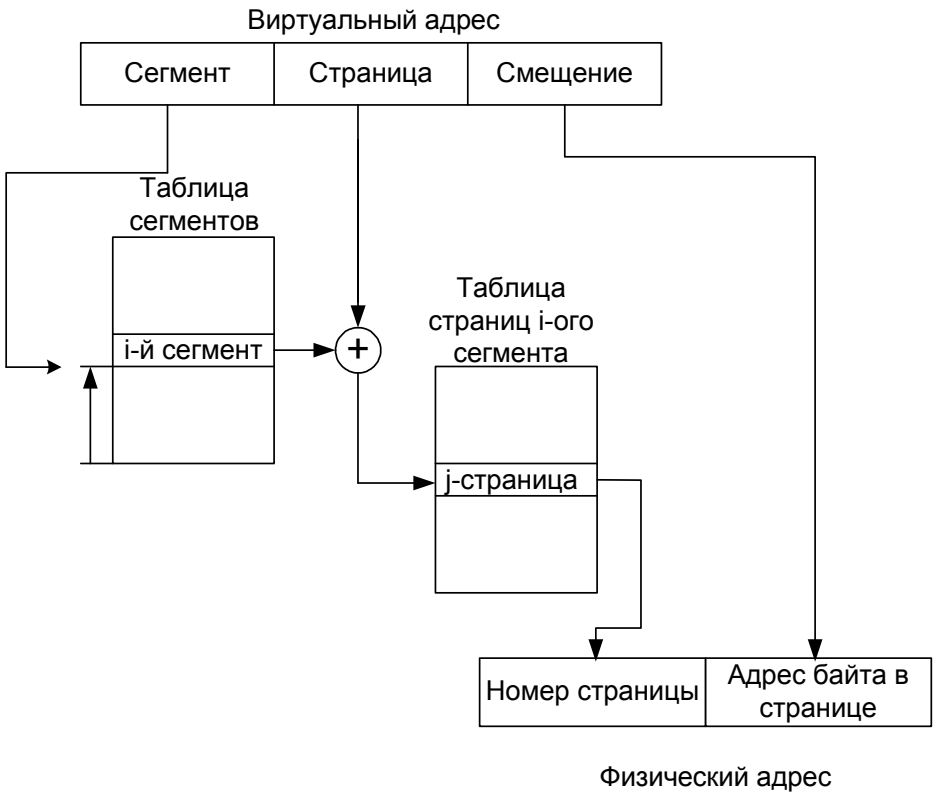


Рис.1.5.Сегментно-страничная организация памяти.

В большинстве вычислительных систем мультипрограммирование ориентировано на параллельную обработку нескольких задач и реализуется посредством системы виртуальной памяти следующими двумя методами. Первый метод основан на разделении виртуального пространства между несколькими задачами. По второму методу для каждой задачи создается отдельное виртуальное пространство адресов. Такая память называется мультиплексной виртуальной памятью.

Для управления мультиплексным виртуальным адресным пространством организуется несколько таблиц преобразования адресов, которые переключаются при переходе от задачи к задаче. Для этого в один из регистров устройства управления процессора вводится указатель, по которому выбирается соответствующая таблица преобразования адресов.

Преимущество этой системы заключается в том, что пространство памяти, используемое каждой задачей, полностью заполняет рамки виртуального пространства. Одновременно обеспечивается высокоэффективная защита памяти, так как никакая задача не может сформировать адрес, относящийся к другой задаче.

Однако время обращения к таблице при преобразовании виртуального адреса в реальный является относительно большим.

Для ускорения этой процедуры на основе использования аппаратных средств разработан так называемый механизм динамического преобразования адресов (Рис.1.6).

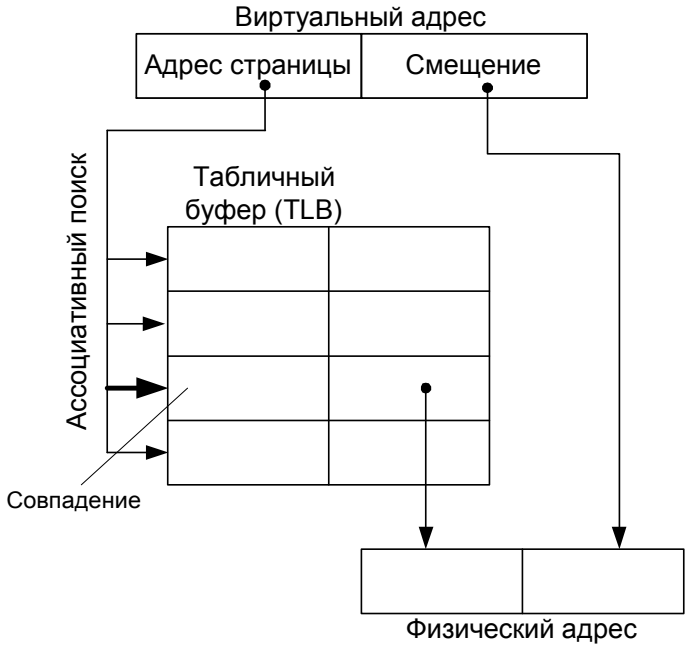


Рис.1.6. Механизм динамического преобразования адресов.

Смысл этого механизма состоит в том, что в ассоциативной памяти заранее записываются номера наиболее часто используемых в данное время страниц номера блоков, соответствующих этим страницам в основной памяти; в ходе преобразования адресов прежде всего проверяется эта ассоциативная память, и если в ней обнаруживаются сведения о

необходимых страницах, эти сведения могут быть сразу же использованы, т. е. сокращается длительность процедуры преобразования адресов.

Подобный буфер для высокоскоростного преобразования адресов называется буфером динамической трансляции адресов виртуальной памяти.

Процедура замены страниц

Когда требуемая страница в основной памяти отсутствует, она переписывается в нее из внешней памяти. Если же в основной памяти не оказывается свободного блока для загрузки страницы, то необходимо удалить какую-либо из страниц, находящихся в ней. Связанные с этим действия называются заменой страниц.

Известны следующие стратегии замены страниц:

1) стратегия FIFO, в соответствии с которой из основной памяти удаляются страницы, раньше других занесенные в нее.

2) стратегия LRU, при использовании которой удаляется та страница, обращение к которой имело место раньше, чем к другим.

3) стратегия WS (Working Set - рабочее множество), в соответствии с которой удаляются страницы, не содержащиеся в так называемом рабочем множестве, т. е. наборе страниц, к которым за определенный истекший интервал времени зафиксировано обращение.

Две из этих стратегий замены страниц - LRU и WS - основаны на предположении, что страницы, использовавшиеся в последний период, будут часто использоваться и впредь. По сравнению с ними реализация стратегии FIFO проще, но эффективность ее относительно ниже. На практике обычно используются стратегии LRU и WS, а также их сочетание и модификации.

Управление распределением основной памяти

Одна из проблем параллельной обработки состоит в том, каким образом распределить блоки основной памяти между всеми программами,

участвующими в этой обработке, В общем случае чем больше объем распределяемой основной памяти, тем реже приходится производить ее перераспределение. Для повышения эффективности выполнения каждой из параллельно обрабатываемых программ необходимо обеспечить их определенным объемом основной памяти. При чрезмерном увеличении уровня мультиплексирования программ уменьшается объем памяти, отводимой каждой из них, повышается частота обмена страниц, что приводит к резкому снижению эффективности всей системы мультипрограммной обработки. Возникает так называемое дробление памяти. Для устранения этого явления управление заметной страниц осуществляется с учетом обеспечения наибольшей эффективности использования центрального процессора. Хотя реализация этого управления занимает время, ситуация в целом улучшается благодаря своевременному сокращению степени мультиплексирования.

3. Процессоры с параллельным выполнением операций

Конвейер команд

Конвейерный принцип выполнения команд относится к MISD (МКОД) архитектурам. Машинные команды (К) выполняются (рисунок 1.23) с помощью ряда последовательных элементарных действий (микроопераций): выборки команды (ВК), дешифрации (декодирования) команды (ДК), формирования адресов операндов и выборки операндов (ВО), исполнения команды (ИК) и запоминания результата (ЗР). В машине с простой структурой аппаратной части выборка следующей машинной команды производится лишь после завершения выполнения предыдущей команды (рис. 1.22а). С другой стороны, в машинах с конвейерной организацией команд, как показано на рис. 1.22б, допустимо одновременное выполнение

нескольких команд путем совмещения во времени выполнения микроопераций этих команд.

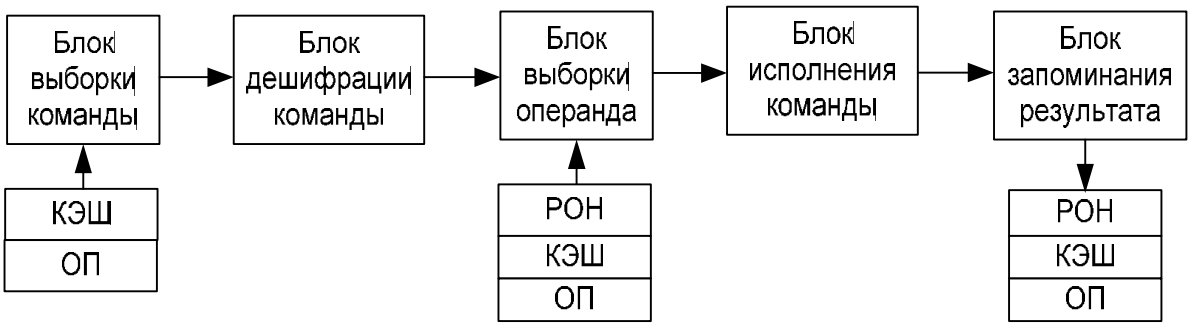


Рисунок 1.22 – Структура конвейерного процессора

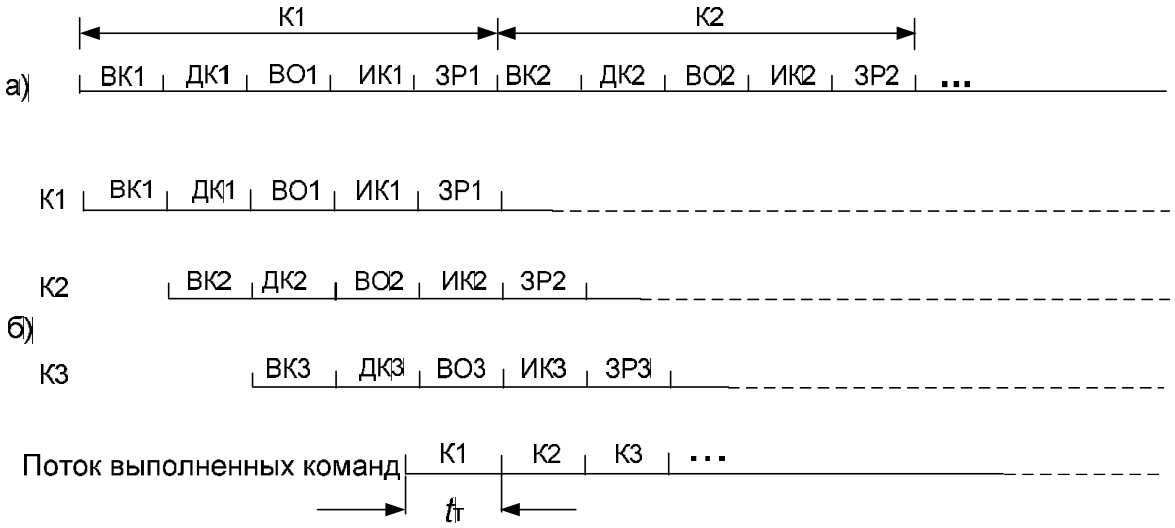


Рис. 1.23. Временная диаграмма работы конвейера микроопераций: а) процессор без конвейера; б) процессор с конвейером.

Если считать, что на выполнение каждого из этапов (шагов) команды затрачивается одинаковое время, например, равное процессорному такту (t_T), то в идеальном случае можно получать результаты операций в каждом процессорном такте.

Конвейерное выполнение команд основано на тех же принципах, что и поточные линии сборки на производстве. Оно имеет максимальную

эффективность, когда продолжительность выполнения всех этапов команд одинакова, бесперебойно подаются команды и данные и на каждом этапе отсутствуют «мертвые» временные зоны, нарушающие непрерывность конвейерной реализации команд. Ситуации, которые приводят к простоям конвейера, называют конфликтами. Существуют конфликты по данным, управлению и структурные конфликты.

К факторам, нарушающим непрерывность конвейера, можно отнести такие ситуации, когда:

1) для выполнения следующей команды требуется результат от предыдущей команды, или предыдущей командой определяется адрес операнда следующей команды (модификация адреса); в этом случае возникает задержка начала выполнения следующей команды, связанная с ожиданием выборки операнда или с преобразованием адресов;

2) операция, реализуемая машинной командой, имеет сложный характер, как, например, операция с плавающей точкой и для ее выполнения требуется много машинных циклов, поэтому последующая команда долго не может достичь стадии выполнения операции;

3) в кэш отсутствуют требуемые данные или команды, поэтому необходимо еще передать их в кэш из основной памяти. При конфликтном обращении к кэш (например, при наложении друг на друга этапов ВК, ВО, ЗР команд, следующих одна за другой) запросы с относительно низкими приоритетами будут находиться в стадии ожидания;

4) при ветвлении программы по результатам проверки условий командой условного перехода команды, находящиеся в процессе конвейерной обработки, остаются невыполненными, и требуется повторная загрузка конвейера, начиная с момента выборки команды условного перехода;

5) возникает прерывание и осуществляется переход к программе его обработки. При этом в командах, находящихся в это время на командном

конвейере остаются незавершенными стадии, на которых прерывается их выполнение, и приходится заново загружать конвейер командами, входящими в программу обработки прерывания.

Конфликты по данным. Для борьбы с конфликтами по данным (1 и 2-й факторы) применяются как программные, так и аппаратные методы. Программные методы ориентированы на устранение самой возможности конфликтов еще на стадии компиляции программы. Оптимизирующий компилятор должен создавать такой объектный код, чтобы между конфликтующими командами находилось достаточное количество нейтральных команд или «холостых» команд типа NOP (команд, не выполняющих никаких действий, но создающих временную задержку).

Фактическое разрешение конфликтов возлагается на аппаратные методы. Конвейерная реализация команд вынудила большинство фирм производителей процессоров перейти на упрощенную систему команд и технологию RISC с ориентацией её на конвейерную обработку. RISC – процессоры характеризуются сравнительной простым устройством управления, унификацией набора и размера команд, а также длительности их выполнения. Все эти факторы положительно сказываются на общем быстродействии процессора и приводят к устранению периодов ожидания в конвейере.

Другим очевидным решением является остановка команды *j* на несколько тактов с тем, чтобы команда *i* успела завершиться или, по крайней мере, миновать ступень конвейера, вызвавшую конфликт. Соответственно задерживаются и команды, следующие в конвейере за *j*-й командой. Данную ситуацию называют «пузырьком» в конвейере. Иногда приостанавливают только команду *j*, не задерживая следующие за ней команды. Это более эффективный прием, но его реализация усложняет конвейер.

Остановки конвейера снижают его эффективность, и разработчики процессоров любыми способами стремятся сократить общее число остановок

или хотя бы их длительность. Поскольку наиболее частые конфликты по данным возникают в результате действия фактора 1, основные усилия тратятся на противодействие именно этому типу конфликтов. Среди известных методов борьбы наибольшее распространение получил прием *ускоренного продвижения информации*. Обычно между двумя соседними ступенями конвейера располагается буферный регистр, через который предшествующая ступень передает результат своей работы на последующую ступень, то есть передача информации возможна лишь между соседними ступенями конвейера. При ускоренном продвижении, когда для выполнения команды требуется операнд, уже вычисленный предыдущей командой, этот операнд может быть получен непосредственно из соответствующего буферного регистра, минуя все промежуточные ступени конвейера. С данной целью в конвейере предусматриваются дополнительные тракты пересылки информации (тракты обхода), снабженные средствами мультиплексирования.

Описанный конфликт обусловлен тем, что одна из команд, например МК2, ожидает записи результата выполнения команды МК1 в РОН. Однако эти данные появляются на выходе АЛУ только по завершении шага ЗР1. Задержку выполнения команды МК2 можно сократить или устранить, предав результаты выполнения команды МК1 непосредственно команде МК2. На рис.1.24 показан тракт обхода, где выбор операнда для следующей операции производится не из РОН, а непосредственно с выхода блока исполнения.

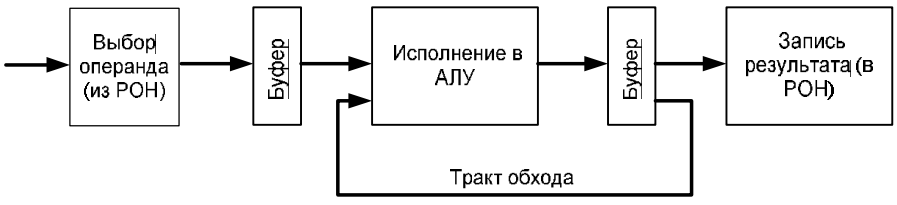


Рисунок 1.24- Ускоренное продвижение операндов в конвейере

Конфликты по управлению. Задача блока выборки команд состоит в обеспечении остальных блоков непрерывным потоком команд. Если этот поток

прерывается, конвейер останавливается. Так происходит в случае промаха при обращении к кэш или выполнении команд перехода (факторы 3,4 и 5). В первом случае блок выборки команды вынужден обратиться к оперативной памяти (если используется один уровень кэш), время доступа к которой значительно больше, чем время обращения к кэш и может занимать несколько десятков тактов. Первый путь решения этой проблемы заключается в применении кэш второго уровня, большего по размеру и с малым временем обращения. Однако из кэш второго уровня данные не могут быть получены за один такт процессора, поэтому в дополнение к этому применяют упреждающую выборку команд. Блок выборки команд извлекает команды ещё до того, как они понадобятся, и помещает их в специальную очередь, откуда они извлекаются диспетчером (дополнительное устройство в составе конвейера). Кроме того, если процессор содержит единственный кэш и для команд и для данных, то параллельно выполняющиеся микрооперации выборки команды и выборки операнда будут часто конфликтовать из-за доступа к кэш. Поэтому в большинстве процессоров используют два первичных кэш: один для хранения команд, другой для хранения операндов.

Существенные проблемы при разработке конвейера обусловлены командами, изменяющими естественный порядок вычислений (факторы 4 и 5). Простейший конвейер эффективно выполняет только линейные программы. Он на этапе (ступени) выборки извлекает команды из последовательных ячеек памяти, используя для этого счетчик команд. Адрес очередной команды в линейной программе формируется автоматически, за счет прибавления к содержимому счетчику команд шага адресации - числа, равного длине текущей команды в байтах. В реальных программах обязательно присутствуют команды передачи управления, изменяющие последовательность вычислений: безусловный и условный переходы, вызов подпрограммы, возврат из подпрограммы и т. п. Выполнение команд перехода приводит к приостановке конвейера на несколько тактов, из-за чего производительность процессора снижается.

Приостановка конвейера связана с выборкой команды из точки перехода (по адресу, указанному в команде перехода). То, что текущая команда относится к командам перехода, становится ясным только после прохождения ступени декодирования (для команд безусловного перехода), или после исполнения команды (для команд условного перехода) то есть спустя 2 такта от момента поступления команды на конвейер или даже выше для длинных конвейеров (с большим числом ступеней). За это время на первые ступени конвейера уже поступят новые команды, извлеченные в предположении, что естественный порядок вычислений не будет нарушен. В случае перехода эти ступени нужно очистить и загрузить в конвейер команду, расположенную по адресу перехода, для чего нужен исполнительный адрес последней. Поскольку в командах перехода обычно указаны лишь способ адресации и адресный код, исполнительный адрес предварительно должен быть вычислен, что и делается па третьей ступени конвейера. Таким образом, реализация перехода в конвейере требует определенных дополнительных операций, выполнение которых равносильно остановке конвейера как минимум на два такта.

Вторая причина нарушения непрерывности работы конвейера имеет отношение только к командам условного перехода. Команды условного перехода в типичной программе могут появляться через каждые 5-10 команд линейного участка и занимают до 20% программного кода. Накладные расходы, связанные с командами перехода, при таком количестве приводят к существенному снижению производительности процессора. Отрицательное влияние команд перехода на производительность можно существенно уменьшить за счет применения специальных методов их обработки.

Буфер предвыборки команд

Конфликты по данным и управлению приводят к приостановке конвейера на один и более тактов. Для устранения этого явления в процессор включают сложные блоки выборки, которые извлекают команды раньше, чем они понадобятся, и помещают их в аппаратно организованную очередь,

называемую буфером предвыборки команд. Чтобы извлекать команды из буфера, необходимо блок дешифрации снабдить устройством, называемым *блоком диспетчеризации,* который извлекает команды, расположенные в начале очереди, и обеспечивает их дешифрацию. Схема конвейера для такой обработки команд приведена на рис. 1.25.

Блок выборки в этом случае должен постоянно формировать основной буфер команд, чтобы уменьшить влияние на работу конвейера различного рода задержек при выборке команд. Кроме того, в блок выборки должен включаются средства распознавания команд перехода. Если останов конвейера вызван конфликтом по данным, блок диспетчеризации должен приостановить выборку команд и их передачу из буфера далее по конвейеру. Блок же выборки может продолжать извлекать команды из памяти и помещать их в буфер. Когда возникает задержка из-за промаха к кэш или выполнения перехода, блок диспетчеризации может продолжать извлекать команды из буфера и передавать их следующим блокам для выполнения.

Когда происходит кэш-промах, блок диспетчеризации продолжает выборку команд из буфера и направляет их на выполнение, пока очередь не опустеет. Тем временем данные считываются из основной памяти или кэш-памяти второго уровня. Если очередь не опустеет до получения новых данных, промах, возникший при обращении к кэш, не отразится на скорости выполнения команд. Очевидно, что эффективность такой технологии конвейерной обработки зависит от величины буфера. Чем больше команд будет находиться в буфере (т.е. длиннее очередь), тем меньше будет величина задержек конвейера.

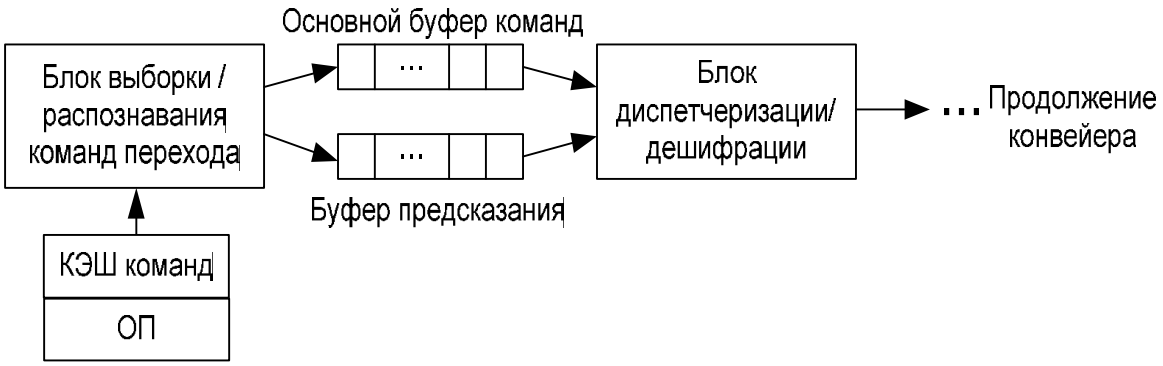


Рисунок 1.25- Включение буферов команд в конвейер

Для обеспечения равномерной работы конвейера при выполнении команд условных переходов в него включают дополнительный буфер. Команды, извлеченные из памяти, анализируется блоком распознавания перехода. При обнаружении команды условного перехода блок распознавания формирует адрес перехода и организует выборку команд в дополнительный буфер предположительной последовательности команд с указанного адреса. Одновременно с этим продолжается выборка и в основной буфер. Далее блок распознавания перехода определяет условие перехода, в зависимости от которого подключает к продолжению конвейера основной или буфер предсказания. Если условие перехода соблюдается, то содержимое основного буфера затем сбрасывается и наоборот.

Предсказание переходов

Предсказание переходов рассматривается как один из наиболее эффективных способов борьбы с конфликтами по управлению. Идея заключается в том, что еще до момента выполнения команды условного перехода делается предположение о том, что с большой вероятностью переход произойдет. Это предположение основано на том, что программы носят циклический характер. После выполнения *i*–й итерации цикла наиболее вероятно будет выполнена (*i*+1) итерация. Если цикл содержит *n* повторений, то вероятность того, что переход к началу цикла состоится, равна (1-1/*n*) и последующие команды будут подаваться

на конвейер в соответствии с предсказанием. При ошибочном предсказании конвейер необходимо очистить начальные ступени конвейера, и приступить к загрузке, начиная с точки перехода, что по эффекту эквивалентно приостановке конвейера. При высокой точности предсказания конвейер функционирует ритмично без остановок и задержек. Исследования показывают, что точность предсказания переходов может составлять более 90%.

Существуют статические и динамические методы предсказания переходов. В современных процессорах применяют те и другие методы или их сочетания. Статические методы предсказания ветвлений являются наиболее простыми. Суть этих методов состоит в том, что различные типы переходов либо выполняются всегда, либо не выполняются никогда. В современных процессорах статические методы используются лишь в том случае, когда невозможно использование динамического предсказания. Статическое предсказание, как правило, не обеспечивает непрерывность работы конвейера при выполнении любой команды перехода. Статическое предсказание часто основывают на результатах компиляции программы.

Рассмотрим выполнение команд безусловного перехода. Из рисунка 1.23 видно, что декодирование команды заканчивается на второй стадии конвейера. Следовательно, блоку выборки команд приходится решать, откуда вызывать следующую команду еще до того, как он узнает, какого типа он только что вызвал. Только в следующем такте он сможет узнать, что получил команду безусловного перехода, а до этого момента времени он уже начал вызывать команды, следующие за безусловным переходом. Эти команду либо надо выполнить и сохранить результат, либо выгрузить, что осуществить достаточно сложно, тем более что 1-2 такта работы конвейера все равно теряются. Кроме того, если загруженным в конвейер командам дается возможность выполниться, то они могут изменить состояние регистров процессора, что приведет к неправильному результату выполнения программы. В процессорах Pentium пошли другим путем. На этапе компиляции производится анализ команд перехода. При их наличии,

компилятор добавляет в программу после команды перехода дополнительное необходимое число команд типа NOP («холостых» команд, не выполняющих никаких действий). После обнаружения команды перехода (на стадии дешифрации или стадии исполнения в зависимости от типа команды: безусловного или условного перехода), в конвейере будут находиться команды NOP, не выполняющие никаких действий. Состояния регистров процессора при их выполнении не претерпят изменений, поэтому выгружать конвейер не потребуется. Такая реализация несколько удлиняет программу, но она оказывается простой.

В других процессорах, например, в SPARC фирмы SUN команда перехода содержит специальный бит предсказания перехода, устанавливаемый компилятором в состояние 0 или 1. Блок выборки команды проверяет этот разряд и выполняет действия в соответствии с указанным в нем предсказанием.

В статических методах особенно плохо дело обстоит с условными переходами. Здесь блок выборки команд узнает, откуда нужно считывать команду, не на этапе декодирования, гораздо позже. Поэтому большинство процессоров прогнозируют, состоится ли условный переход, или нет. Для этого, например, можно предполагать, что все условные переходы назад будут осуществляться, а все условные переходы вперед не будут. Что касается первого предположения, то причина такого выбора кроется в том, что переходы назад часто помещаются в конце цикла. Большинство циклов выполняется много раз, поэтому переход к началу цикла встречается очень часто.

Динамические методы, широко используемые в современных процессорах, подразумевают анализ истории ветвлений. Механизм предсказания переходов выполняет две основные функции — предсказание адреса команды, на которую производится переход (для всех типов команд перехода), и предсказание направления ветвления (для команд условного перехода). Оба предсказания должны быть выполнены заблаговременно —

раньше, чем начнётся декодирование и обработка команды перехода — для того, чтобы выборка новой команды (строки кэш, содержащей новую команду) была произведена без потерь лишних тактов либо с минимальными потерями.

Необходимость предсказания адреса «целевой» команды вызвана тем, что этот адрес может быть извлечён из команды перехода только на финальной стадии декодирования, т.е. с довольно большой задержкой. В современных процессорах для предсказания адреса перехода обычно используют специальную таблицу адресов переходов BTB (Branch Target Buffer). Эта таблица устроена подобно кэш и содержит адреса команд, на которые ранее производились переходы. Например, в процессоре P-III таблица BTB имеет размер 512 элементов и организована в виде 128 наборов с ассоциативностью 4. Если в адресуемой строке кэш есть команды перехода, и если эти команды отрабатывали ранее, то алгоритм предсказания может очень быстро найти адрес целевой команды в таблице BTB и начать считывание строки кэш, содержащей эту команду. Адреса целевых команд помещаются в BTB после выполнения соответствующих команд перехода. В современных процессорах размер таблицы BTB достигает 4096 элементов.

Для предсказания направления условного перехода используется другой механизм, основанный на изучении поведения переходов в программе в процессе её выполнения (сбор статистики). Этот механизм учитывает поведение команды перехода, например, «скорее всего, будет выполнен», «возможно, будет выполнен», «возможно, не будет выполнен», «скорее всего, не будет выполнен». История поведения команды условного перехода записывается в специальных управляющих битах таблицы BTB, обычно называемых битами прогнозирования перехода или «таблицами истории переходов» (Branch History Table, BHT), которые обновляется после выполнения каждого перехода.

Что касается фактора 5, вызываемого прерываниями, то к счастью они случаются не так часто как команды переходов и на производительность конвейера большого влияния не оказывают.

Суперскалярные процессоры.

В конвейерных процессорах наиболее трудоемкой является стадия исполнения операции. Объясняется это тем, что даже в RISC процессорах команды могут иметь разную длину. Например, команды с плавающей точкой содержат большее число микроопераций, чем целочисленные и, следовательно, время выполнения их значительно больше. Поскольку цикл выполнения коротких команд, как указывалось ранее, должен равняться циклу выполнения длинных команд, то это приводит к снижению производительности процессора. С целью повышения производительности конвейер может быть оборудован несколькими исполнительными блоками, чтобы на каждом из них обрабатывалось параллельно несколько команд (рисунок 1.26). Процессоры такого типа называются *суперсколярными.* Суперскалярный процессор является многофункциональным устройством, содержащим множество операционных устройств. Такую архитектуру имеют многие современные высокопроизводительные процессоры.

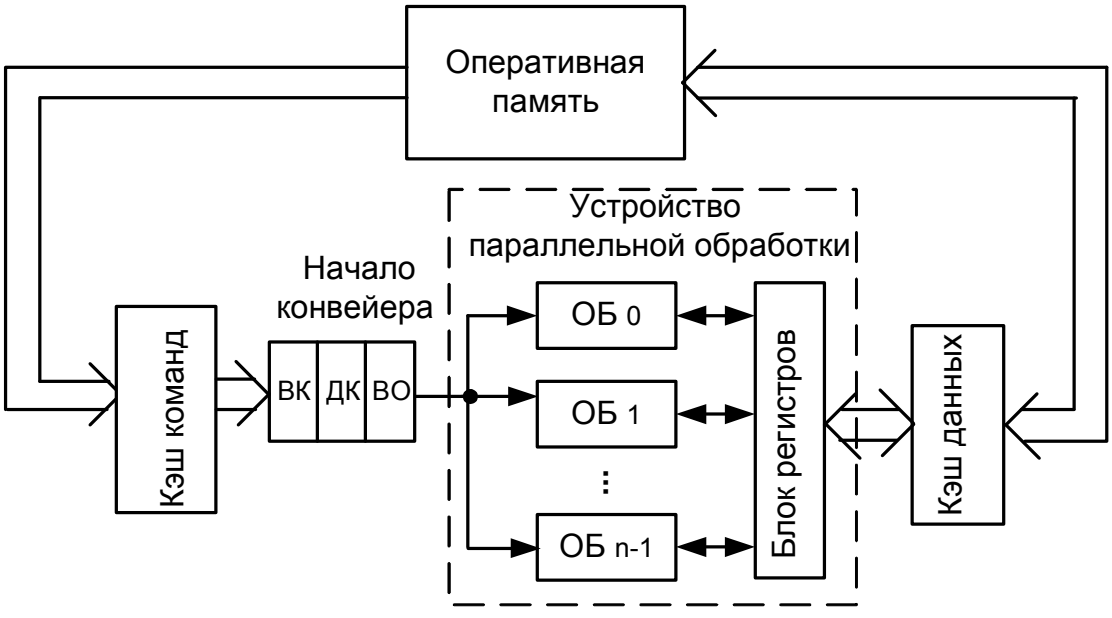


Рис.1.26. Структура суперскалярного процессора

В соответствии с используемым принципом, команды, последовательно формируемые конвейером команд, выполняются свободными операционными блоками (ОБ), причем результаты операций могут выдаваться с очередностью, отличной от очередности команд в командном конвейере. Например, в машине Эльбрус-2 десять операционных блоков (сумматор с плавающей точкой, умножитель, сумматор с фиксированной точкой и др.) параллельно выполняют соответствующие операции, совместно используя одни и те же регистры. Причём в процессе выполнения команд производится проверка занятости необходимых операционных блоков, управление поступлением требуемых данных (операндов) из других операционных блоков и памяти и передача управления операционным блокам, когда для выполнения команд готовы операнды. При невозможности выполнения команды из-за того, что данные не готовы, происходит процесс ожидания до тех пор, пока выполнение команды будет возможно. Таким образом, если длительность операции, задаваемой последующей командой, меньше и связь между ее данными и результатами выполнения предыдущей команды отсутствует, то результат этой операции окажется в регистре раньше, чем результат операции предыдущей команды. В машине Эльбрус-2 новая команда поступает в каждом машинном цикле, тогда как быстродействие операционных блоков позволяет произвести суммирование с фиксированной точкой за три машинных цикла, умножение с плавающей точкой — за 10 машинных циклов, а деление с плавающей точкой — за 29 машинных циклов. Другими словами, выполнение обработки операционным блоком отстает от темпа поступления команд, поэтому введением дополнительных операционных блоков устанавливается баланс между темпом поступления команд и скоростью выполнения операций.

Подобный же принцип управления вычислительным процессом применён в процессорах Pentium, причём в зависимости от семейства число операционных блоков может достигать пяти. Так в процессорах семейства P6 в операционной части конвейера используются два целочисленных блока,

один блок для операций с плавающей точкой, один блок MMX для обработки потока целочисленных данных (Pentium, Pentium II) и один блок SSE для обработки потока чисел с плавающей точкой (Pentium III). В Pentium IV

В суперскалярных процессорах возникает новая проблема, связанная с недостаточностью загрузки операционных блоков. Она вызвана нерегулярностью поступления различных типов команд, необходимых для одновременной загрузки всех исполнительных блоков. Чтобы обеспечить загрузку необходимо иметь возможность выбирать из памяти несколько команд за один такт, причем тип команд должен соответствовать типу операционного блока. Для реализации такого режима выборки применяют широкую шину для связи с кэш-памятью, обеспечивающую выборку двойными или четверными словами за один такт.

На рисунке 1.27 приведен пример процессора с двумя операционными блоками: для операций с фиксированной точкой (целочисленных) и операций с плавающей точкой. Блок выборки команд считывает из кэш по две команды за раз и сохраняет их в буфере. На каждом такте блок диспетчеризации извлекает из очереди и декодирует одну или две команды. Если одна из команд обрабатывает числа с фиксированной точкой, а другая — числа с плавающей точкой, то при отсутствии конфликтов обе команды запускаются на выполнение на одном такте.

Для реализации такого режима применяют оптимизирующие компиляторы, предотвращающие многие конфликты путем переупорядочивания команд до их подачи в конвейер. Например, в компьютерах на основе процессоров Pentium, компилятор по возможности обеспечивает чередование операций с плавающей точкой и операций с фиксированной точкой. Это дает возможность блоку диспетчеризации поддерживать непрерывную работу арифметических устройств с и плавающей фиксированной точкой, что, в конечном счете, обеспечивает максимальную загрузку операционных блоков и повышение производительности. Будем называть согласованным выполнение команд,

если результаты на завершающем этапе работы конвейера появляются в порядке их следования в программе.

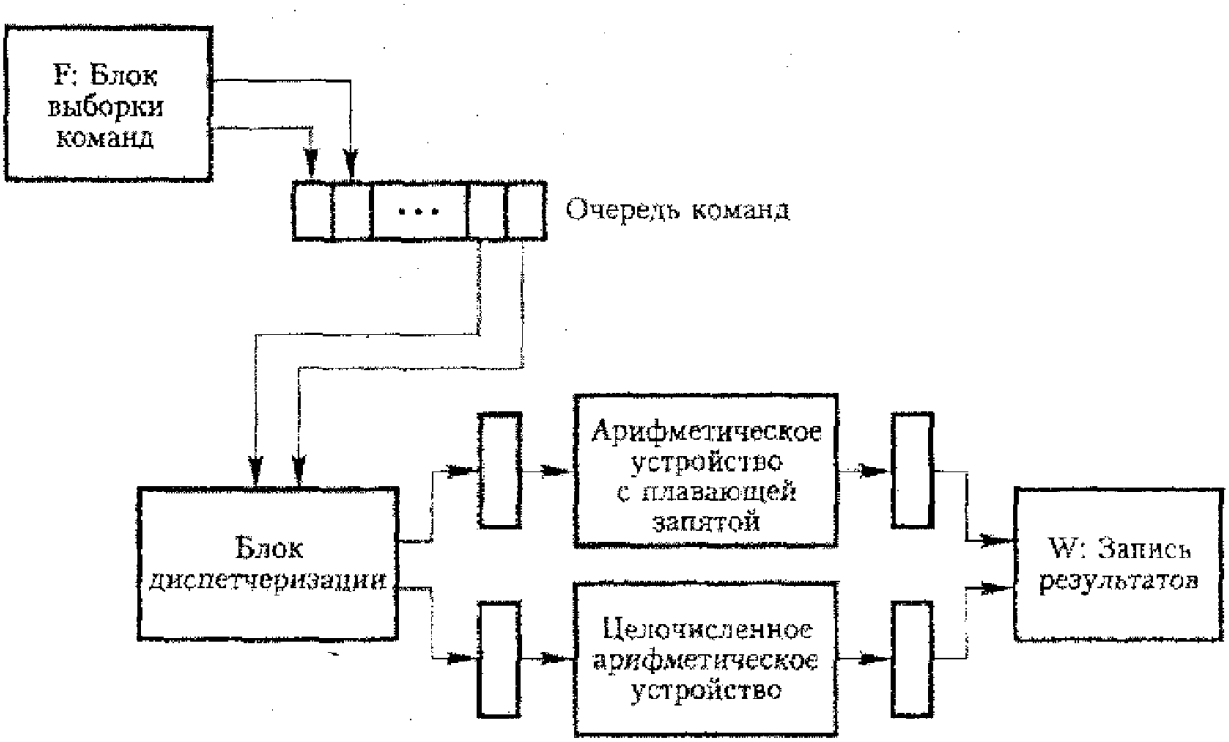


Рис. 1.27. Процессор с двумя операционными блоками.

На рис. 1.28 представлена временная диаграмма конвейера суперскалярного процессора. Пусть К1 и К3 являются командами с плавающей точкой, К2 и К4 – с фиксированной точкой. Для выполнения операции командам К1 и К3 в арифметическом устройстве с плавающей точкой требуются три такта. Устройство с фиксированной точкой выполняет команды К2 и К4 за один такт. Предполагается, что устройство с плавающей запятой организовано как трехступенчатый конвейер. Поэтому на каждом такте оно может принимать новую команду.

Если в ходе выполнения вычислительного процесса не произойдет никаких конфликтов, выполнение команд завершится так, как показано на рис. 1.28.

Рис. 1.28. Временная диаграмма не согласованной работы конвейера команд с двумя операционными блоками

Внеочередное завершение команд

В традиционных процессорах команды завершаются в том порядке, в каком они следуют в программе. В суперскалярных процессорах этот порядок может нарушаться. Короткие команды, следующие в программе после длинных, могут выполниться раньше (см. рис. 1.28). Их выполнение необходимо задержать на некоторое число тактов, чтобы соблюсти порядок выдачи результата, что приведет к необоснованным простоям оборудования и потерям производительности. Как следует из примера на рис. 1.28 выполнение команд К2 и К4 следует задержать на два такта, чтобы обеспечить заданный в программе порядок выполнения.

Однако, чтобы ускорить освобождение операционных блоков и загрузку в них новых команд, необходимо разрешить внеочередное завершение команд. Для этого команды должны выполняться так, как показано на рис. 1.28, но результаты при этом следует сохранять в теневых (временных) регистрах, чтобы позднее пересылать их в основные (постоянные) регистры процессора. Пример такого варианта выполнения программы приведен на рис. 1.29. На такте ЗВР (запись во временный регистр) осуществляется запись в теневой регистр, а на шаге ЗР, завершающем для команды, содержимое теневого регистра переписывается в соответствующий основной регистр. Этот шаг

часто называют *шагом сохранения,* поскольку после него результат выполнения команды уже не может быть отменен.

Теневой регистр некоторое время исполняет роль основного регистра, причем его имя то же, что и у основного регистра. Например, если результат выполнения команды К2 сохраняется в регистре *R*5, то временный регистр, используемый на шаге ЗВР2 интерпретируется как *R*5. Его содержимое передается любым следующим командам, обращающимся к регистру *R*5. Описанная технология называется *переименованием регистров.* Следует обратить внимание на то, что временный регистр применяется только теми командами, которые при естественном порядке выполнения программы следуют за К2. Если регистр *R*5 потребуется команде, предшествующей К2, она получит доступ к реальному регистру *R*5.

Внеочередное завершение команд требует наличия в процессоре специального управляющего блока - *блока сохранения*. Этот блок выбирает следующую команду для сохранения, используя очередь, называемую *буфером реорганизации.* Команды записываются в эту очередь в том порядке, в каком они следуют в программе, по мере диспетчеризации для выполнения. Когда команда достигает начала очереди и завершается, соответствующие результаты пересылаются из теневых регистров в основные. После этого команда удаляется из очереди, а все выделенные ей ресурсы, включая теневые регистры, очищаются. Итак, команда теперь считается покинувшей конвейер.

Описанный алгоритм позволяет выполнять команды в любой последовательности и обеспечивает их выход с конвейера в строгом соответствии с порядком расположения в программе.

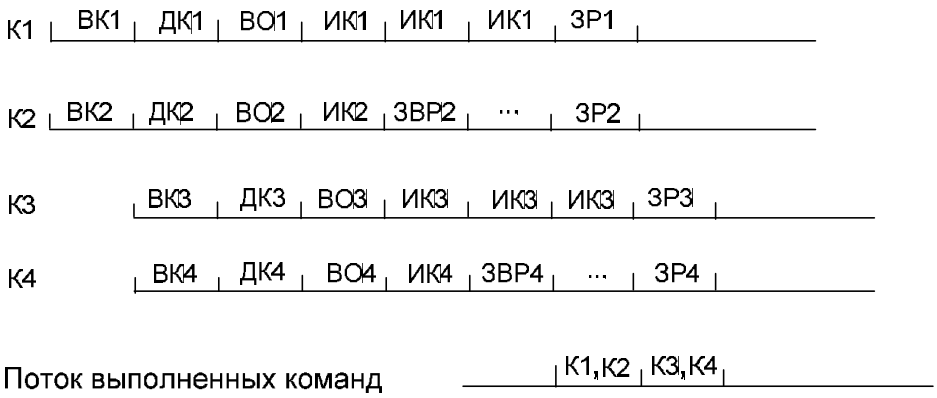


Рис. 1.29. Согласованное выполнение команд в программе использованием временных регистров

Преимущества:

- Аппаратные решения обеспечивают:

- автоматическое обнаружение потенциального параллелизма между командами;

- выдачу максимально возможного числа команд, выполняемых параллельно;

- переименование регистров.

- Совместимость на уровне программ:

- если улучшения были внесены в архитектуру без изменения системы команд, то старые программы будут выполняться быстрее из-за наличия параллелизма.

Недостатки:

- Высокая сложность процессора из-за большого объема оборудования, необходимого для реализации дополнительных функций;

- Традиционные ограничения на степень параллелизма на уровне команд, характерные для конвейеров операций;
- Размер окна исполнения (число активных команд, могущих исполняться параллельно) ограничивает возможный присущий программе параллелизм, так как не рассматривается параллельное исполнение команд, находящихся на расстоянии, превышающем размер окна.

Векторные процессоры

Векторные и векторно-конвейерные вычислительные системы

Хотя производительность ВС общего назначения неуклонно возрастает, по-прежнему остаются задачи, требующие существенно большей вычислительной мощности. К таким, прежде всего, следует отнести задачи моделирования реальных процессов и объектов, для которых характерна обработка больших регулярных массивов чисел в форме с плавающей запятой. Такие массивы представляются матрицами и векторами, а алгоритмы их обработки описываются в терминах матричных операций. Как известно, основные матричные операции сводятся к однотипным действиям над парами элементов исходных матриц, которые, чаще всего, можно производить параллельно. В универсальных ВС, ориентированных на скалярные операции, обработка матриц выполняется поэлементно и последовательно. При большой размерности массивов последовательная обработка элементов матриц занимает слишком много времени, что и приводит к неэффективности универсальных ВС для рассматриваемого класса задач. Для обработки массивов требуются вычислительные средства, позволяющие с помощью единой команды производить действие сразу над всеми элементами массивов — средства *векторной обработки.*

Понятие вектора и размещение данных в памяти

В средствах векторной обработки под *вектором* понимается одномерный массив однотипных данных (обычно в форме с плавающей запятой), регулярным образом размещенных в памяти ВС. Если обработке подвергаются многомерные массивы, их также рассматривают как векторы. Такой подход допустим, если учесть, каким образом многомерные массивы хранятся в памяти ВМ. Пусть имеется массив данных *A,* представляющий собой прямоугольную матрицу размерности 4x5 (рис. 13.2).

a_{11}	a_{12}	a_{13}	a_{14}	a_{15}
a_{21}	a_{22}	a_{23}	a_{24}	a_{25}
a_{31}	a_{32}	a_{33}	a_{34}	a_{35}
a_{41}	a_{42}	a_{43}	a_{44}	a_{45}

Рис. 13.2. Прямоугольная матрица данных

При размещении матрицы в памяти все ее элементы заносятся в ячейки с последовательными адресами, причем данные могут быть записаны строка за строкой или столбец за столбцом (рис. 13.3). С учетом такого размещения многомерных массивов в памяти вполне допустимо рассматривать их как векторы и ориентировать соответствующие вычислительные средства на обработку одномерных массивов данных (векторов).

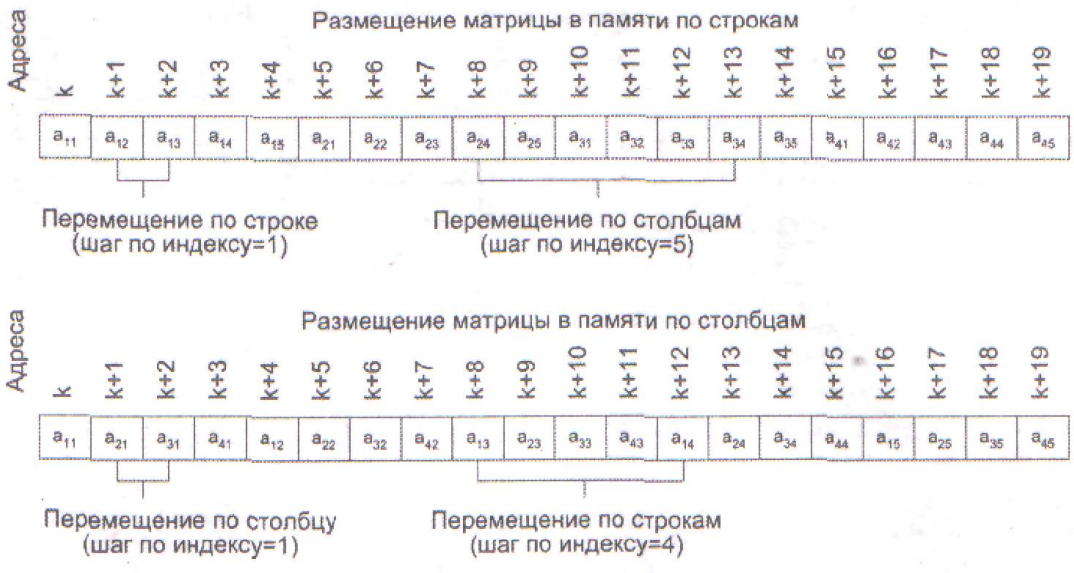


Рис. 13.3. Способы размещения в памяти матрицы 4 × 5

Действия над многомерными массивами имеют свою специфику. Например, в двумерном массиве обработка может вестись как по строкам, так и по столбцам. Это выражается в том, с каким шагом должен меняться адрес очередного выбираемого из памяти элемента. Так, если рассмотренная в примере матрица расположена в памяти построчно, адреса последовательных элементов строки различаются на единицу, а для элементов столбца шаг равен пяти. При размещении матрицы по столбцам единице будет равен шаг по столбцу, а шаг по строке - четырем. В векторной концепции для обозначения шага, с которым элементы вектора извлекаются из памяти, применяют термин *шаг по индексу* (stride).

Еще одной характеристикой вектора является число составляющих его элементов - *длина вектора.*

Понятие векторного процессора

Векторный процессор - это процессор, в котором операндами некоторых команд могут выступать упорядоченные массивы данных - векторы. Векторный процессор может быть реализован в двух вариантах. В первом он представляет собой дополнительный блок к универсальной вычислительной машине (системе). Во втором - векторный процессор - это основа самостоятельной ВС.

Рассмотрим возможные подходы к архитектуре средств векторной обработки. Наиболее распространенные из них сводятся к трем группам:

- конвейерное АЛУ;
- массив АЛУ;
- массив процессорных элементов.

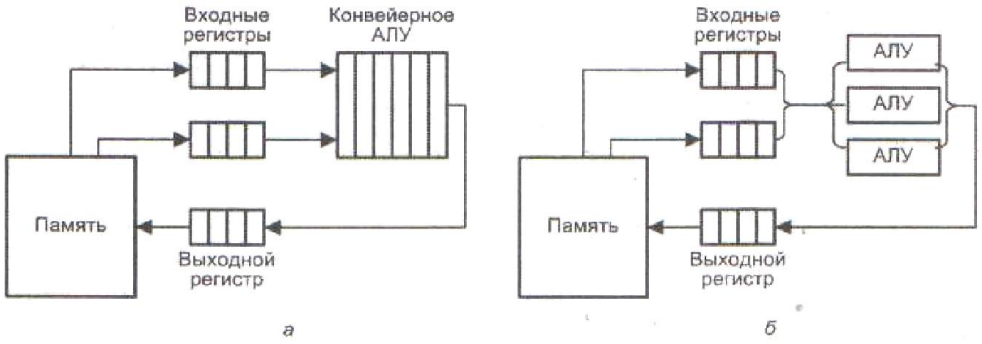


Рис. 13.4. Варианты векторных вычислений: *а* — с конвейерным АЛУ; *б* — с несколькими АЛУ

В варианте с конвейерным АЛУ (рис. 13.4, *а*) обработка элементов векторов производится конвейерным АЛУ для чисел с плавающей запятой (ПЗ). Операции с числами в форме с ПЗ достаточно сложны, но поддаются разбиению на отдельные шаги. Так, сложение двух чисел может быть сведено к четырем этапам [200]:

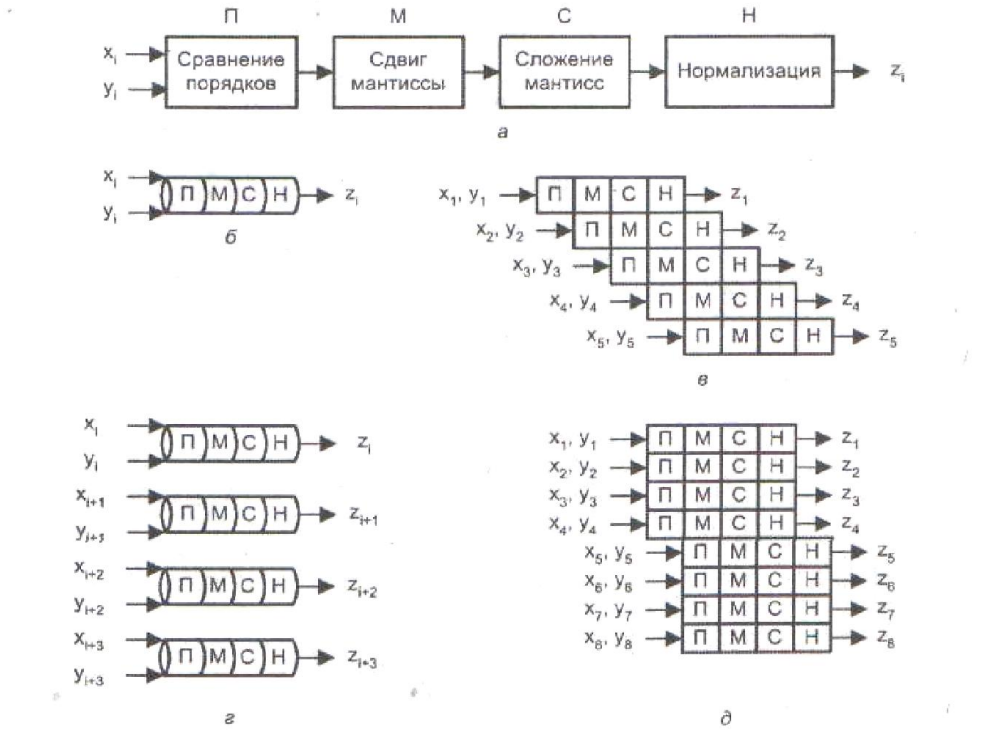


Рис. 13.5. Обработка векторов: *а* — структура арифметического конвейера для чисел с плавающей запятой; *б* — обозначение конвейера; *в* — обработка векторов в конвейерном АЛУ; *г* — параллельная обработка векторов несколькими конвейерными АЛУ; *д* — конвейерная обработка векторов четырьмя АЛУ

Последний вариант - один из случаев многопроцессорной системы, известной как *матричная ВС.* Понятие векторного процессора имеет отношение к двум первым группам, причем, как правило, к первой. Эти две категории иллюстрирует рис. 13.4 [200] сравнению порядков, сдвигу мантиссы меньшего из чисел, сложению мантисс и нормализации

результата (рис. 13.5,*а)*. Каждый этап может быть реализован с помощью отдельной ступени конвейерного АЛУ (рис. 13.5,*б)*. Очередной элемент вектора подается на вход конвейера, как только освобождается первая ступень (рис. 13.5,*в)*. Ясно, что такой вариант вполне годится для обработки векторов.

Одновременные операции над элементами векторов можно проводить и с помощью нескольких параллельно используемых АЛУ, каждое из которых отвечает за одну пару элементов (см. рис. 13.4,*б)*. Такого рода обработка, когда каждое из АЛУ является конвейерным, показана на рис. 13.5,*г*.

Если параллельно используются конвейерные АЛУ, то возможен еще один уровень конвейеризации, что иллюстрирует рис. 13.5, *д*. Вычислительные системы, где реализована эта идея, называют *векторно-конвейерными.* Коммерческие векторно-конвейерные ВС, и состав которых для обеспечения универсальности включен также скалярный процессор, известны как *суперЭВМ.*

Структура векторного процессора

Обобщенная структура векторного процессора приведена на рис. 13.6. На схеме показаны основные узлы процессора, без детализации некоторых связей между ними.

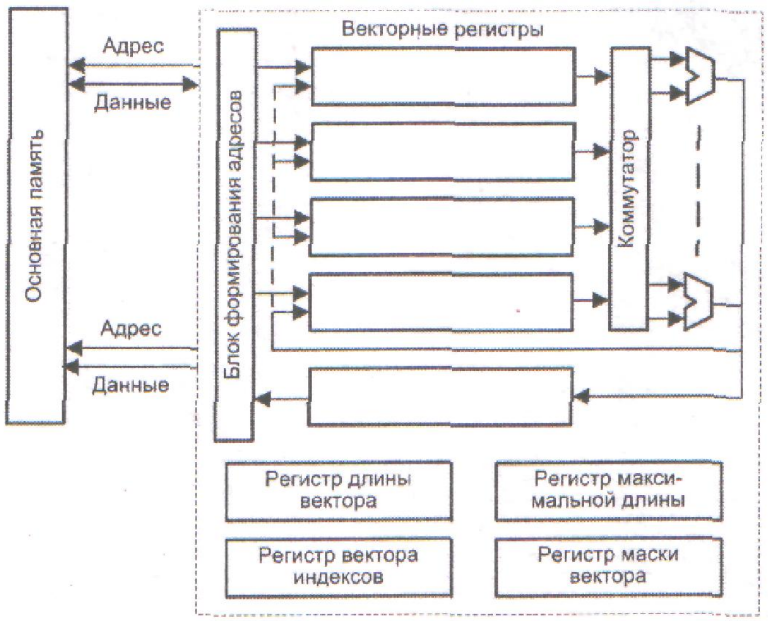


Рис. 13.6. Упрощенная структура векторного процессора

Обработка всех *n* компонентов векторов-операндов задается одной *векторной командой.* Общепринято, что элементы векторов представляются числами в форме с плавающей запятой (ПЗ). АЛУ векторного процессора может быть реализовано в виде единого конвейерного устройства, способного выполнять все предусмотренные операции над числами с ПЗ. Более распространена, однако, иная структура, - в ней АЛУ состоит из отдельных блоков сложения и умножения, а иногда и блока для вычисления обратной величины, когда операция деления *X/Y* реализуется в виде *X(1/Y).* Каждый ил таких блоков также конвейеризирован.

Кроме того, в состав *векторной вычислительной системы* обычно включают и скалярный процессор, что позволяет параллельно выполнять векторные и скалярные команды.

Для хранения векторов-операндов вместо множества скалярных регистров используют так называемые *векторные регистры,* которые представляют собой совокупность скалярных регистров, объединенных в очередь типа FIFO, способную хранить 50-100 чисел с плавающей запятой. Набор векторных регистров (V_a, V_b, V_c,...) имеется в любом векторным процессоре. Система команд векторного процессора поддерживает работу с векторными регистрами и обязательно включает в себя команды:

- загрузки векторного регистра содержимым последовательных ячеек памяти, указанных адресом первой ячейки этой последовательности;
- выполнения операций над всеми элементами векторов, находящихся в векторных регистрах;
- сохранения содержимого векторного регистра в последовательности ячеек памяти, указанных адресом первой ячейки этой последовательности.

Примером одной из наиболее распространенных операций, возлагаемых на векторный процессор, может служить операция перемножения матриц [161]. Рассмотрим перемножение двух матриц A и B размерности 3x3.

$$\begin{bmatrix} c_{11} & c_{12} & c_{13} \\ c_{21} & c_{22} & c_{23} \\ c_{31} & c_{32} & c_{33} \end{bmatrix} = \begin{bmatrix} a_{11} & a_{12} & a_{13} \\ a_{21} & a_{22} & a_{23} \\ a_{31} & a_{32} & a_{33} \end{bmatrix} \times \begin{bmatrix} b_{11} & b_{12} & b_{13} \\ b_{21} & b_{22} & b_{23} \\ b_{31} & b_{32} & b_{33} \end{bmatrix}.$$

Элементы матрицы результата **C** *связаны с соответствующими элементами исходных матриц* **A** и **B** операцией скалярного произведения:

$$c_{ij} = \sum_{k=1}^{3} a_{ik} \times b_{kj}.$$

Так, элемент c_{11} вычисляется как

$$c_{11} = a_{11} \times b_{11} + a_{12} \times b_{21} + a_{13} \times b_{31}.$$

Это требует трех операций умножения и (после инициализации $c_{,}$, нулем) трех операций сложения. Общее число умножений и сложений для рассматриваемого примера составляет 9 x 3 = 27. Если рассматривать связанные операции умножения и сложения как одну кумулятивную операцию $c + a$ x b, то для умножения двух матриц nxn необходимо n^3 операций типа «умножение-сложение». Вся процедура сводится к получению n^2 скалярных произведений, каждое из которых является итогом n операций «умножение-сложение», учитывая, что перед вычислением каждого элемента c_{ij} его необходимо обнулить. Таким образом, скалярное произведение состоит изk членов:

$$\mathbf{C} = \mathbf{A}_1\mathbf{B}_1 + \mathbf{A}_2\mathbf{B}_2 + \mathbf{A}_3\mathbf{B}_3 + \mathbf{A}_4\mathbf{B}_4 + \ldots + \mathbf{A}_k\mathbf{B}_k.$$

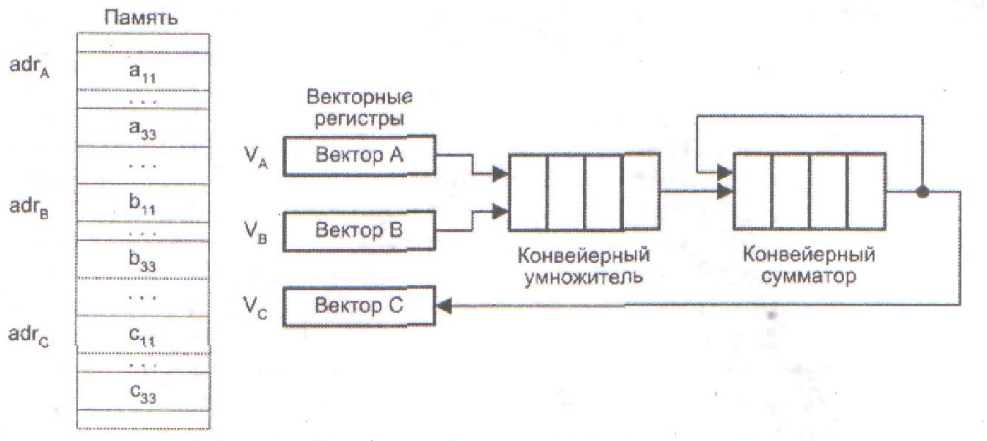


Рис. 13.7. Векторный процессор для вычисления скалярного произведения

Векторный процессор с конвейеризированными блоками обработки для вычисления скалярного произведения показан на рис. 13.7.

Векторы A и B, хранящиеся в памяти начиная с адресов adr_A и adr_B, загружаются в векторные регистры V_A и V_B соответственно. Предполагается, что конвейерные умножитель и сумматор состоят из четырех сегментов, которые вначале инициализируются нулем, поэтому в течение первых восьми циклов, пока оба конвейера не заполнятся, на выходе сумматора будет 0. Пары (A_i, B_i) подаются на вход умножителя и перемножаются в темпе одна пара за цикл. После первых четырех циклов произведения начинают суммироваться с данными, поступающими с выхода сумматора. В течение следующих четырех циклов на вход сумматора поступают суммы произведений из умножителя с нулем. К копну восьмого цикла в сегментах сумматора находятся четыре первых произведения $\mathbf{A_1B_1}$, ..., $\mathbf{A_4B_4}$, а в сегментах умножителя - следующие четыре произведения: A_5B_5,..., A_8B_8. К началу девятого цикла на выходе сумматора будет $\mathbf{A_1B_1}$, а на выходе умножителя – $\mathbf{A_5B_5}$. Таким образом, девятый цикл начнется со сложения в сумматоре $\mathbf{A_1B_1}$ и $\mathbf{A_5B_5}$. Десятый цикл начнется со сложения $\mathbf{A_2B_2}$ + A_6B_6 и т. д. Процесс суммирования в четырех секциях выглядит так:

$$\begin{aligned}\mathbf{C} = {} & \mathbf{A_1B_1} + \mathbf{A_5B_5} + \mathbf{A_9B_9} + \mathbf{A_{13}B_{13}} + \ldots \\ & + \mathbf{A_2B_2} + \mathbf{A_6B_6} + \mathbf{A_{10}B_{10}} + \mathbf{A_{14}B_{14}} + \ldots \\ & + \mathbf{A_3B_3} + \mathbf{A_7B_7} + \mathbf{A_{11}B_{11}} + \mathbf{A_{15}B_{15}} + \ldots \\ & + \mathbf{A_4B_4} + \mathbf{A_8B_8} + \mathbf{A_{12}B_{12}} + \mathbf{A_{16}B_{16}} + \ldots\end{aligned}$$

Когда больше не остается членов для сложения, система заносит в умножитель четыре нуля. Конвейер сумматора в своих четырех сегментах при этом будет содержать четыре скалярных произведения, соответствующие четырем суммам, приведенным в четырех строках показанного выше уравнения. Далее четыре частичных суммы складываются для получения окончательного результата.

Программа для вычисления скалярного произведения векторов **A** и **B**, хранящихся в двух областях памяти с начальными адресами adr_A и adr_B, соответственно может выглядеть так:

```
V_load V_A, adr_A
V_load V_B, adr_B
V_multiply V_C, V_A, V_B.
```

Первые две векторные команды V_Load загружают векторы из памяти в векторные регистры $V_Л$ и V_B. Векторная команда умножения V_multiply вычисляет произведение для всех пар одноименных элементов векторов и записывает полученный вектор в векторный регистр V_c.

Важным элементом любого векторного процессора (ВП) является *регистр длины вектора.* Этот регистр определяет, сколько элементов фактически содержит обрабатываемый в данный момент вектор, то есть сколько индивидуальных операций с элементами нужно сделать. В некоторых ВП присутствует также *регистр максимальной длимы вектора,* определяющий максимальное число элементов вектора, которое может быть одновременно обработано аппаратурой процессора. Этот регистр используется при разделении очень длинных векторов на сегменты, длина которых соответствует максимальному числу элементов, обрабатываемых аппаратурой за один прием.

Достаточно часто приходится выполнять такие операции, в которых должны участвовать не все элементы векторов. Векторный процессор обеспечивает данный режим с помощью *регистра маски вектора.* В этом регистре каждому элементу вектора соответствует один бит. Установка бита в единицу разрешает запись соответствующего элемента вектора результата в выходной векторный регистр, а сброс в ноль - запрещает.

Как уже упоминалось, элементы векторов в памяти расположены регулярно, и при выполнении векторных операций достаточно указать значение шага по индексу. Существуют, однако, случаи, когда необходимо обрабатывать только ненулевые элементы векторов. Для поддержки подобных операций в системе команд ВП предусмотрены операции *упаковки /распаковки* (gather/scatter). Операция упаковки формирует вектор, содержащий только ненулевые элементы исходного вектора, а операция распаковки производит обратное преобразование. Обе этих задачи векторный процессор решает с помощью вектора индексов, для хранения которого используется *регистр вектора индексов,* по структуре аналогичный регистру маски. В векторе индексов каждому элементу исходного вектора

соответствует один бит. Нулевое значение бита свидетельствует, что соответствующий элемент исходного вектора равен нулю.

Использование векторных команд окупается благодаря двум качествам. Во-первых, вместо многократной выборки одних и тех же команд достаточно произвести выборку только одной векторной команды, что позволяет сократить издержки за счет устройства управления и уменьшить требования к пропускной способности памяти. Во-вторых, векторная команда обеспечивает процессор упорядоченными данными. Когда инициируется векторная команда, ВС знает, что ей нужно извлечь *n* пар операндов, расположенных в памяти регулярным образом. Таким образом, процессор может указать памяти на необходимость начать извлечение таких пар. Если используется память с чередованием адресов, эти пары могут быть получены со скоростью одной пары за цикл процессора и направлены для обработки в конвейеризированный функциональный блок. При отсутствии чередования адресов или других средств извлечения операндов с высокой скоростью преимущества обработки векторов существенно снижаются.

Структуры типа «память-память» и «регистр-регистр»

Принципиальное различие архитектур векторных процессоров проявляется в том, каким образом осуществляется доступ к операндам. При организации *«память-память»* элементы векторов поочередно извлекаются из памяти и сразу же направляются в функциональный блок. По мере обработки получающиеся элементы вектора результата сразу же заносятся в память. В архитектуре типа *«регистр-регистр»* операнды скачала загружаются в *векторные регистры,* каждый из которых может хранить сегмент вектора, например 64 элемента. Векторная операция реализуется путем извлечения операндов из векторных регистров и занесения результата в векторные регистры.

Преимущество ВП с режимом «память-память» состоит в возможности обработки длинных векторов, в то время как в процессорах типа «регистр-регистр» приходится разбивать длинные векторы на сегменты фиксированной

длины. К сожалению, за Гибкость режима «память-память» приходится расплачиваться относительно большими издержками, известными как *время запуска,* представляющее собой временной интервал между инициализацией команды и моментом, когда первый результат появится на выходе конвейера. Большое время запуска в процессорах типа «память-память» обусловлено скоростью доступа к памяти, которая намного больше скорости доступа к внутреннему регистру. Однако когда конвейер заполнен, результат формируется в каждом цикле. Модель времени работы векторного процессора имеет вид:

$$T = s + \alpha \times N,$$

где s - время запуска, α - константа, зависящая от команды (обычно 1/2,1 или 2) и *N*- длина вектора.

Архитектура типа «память-память» реализована в векторно-конвейерных вычислительных системах Advanced Scientific Computer фирмы Texas Instruments Inc., семействе вычислительных систем фирмы Control Data Corporation, прежде всего, Star 100, серии Cyber 200 и ВС типа ЕТЛ-10. Все эти вычислительные системы, появились в середине 70-х прошлого века после длительного цикла разработки, но к середине 80-х годов от них отказались. Причиной послужило слишком большое время запуска - порядка 100 циклов процессора. Это означает, что операции с короткими векторами выполняются очень неэффективно, и даже при длине векторов в 100 элементов процессор достигал только половины потенциальной производительности.

В вычислительных системах типа «регистр-регистр» векторы имеют сравнительно небольшую длину (в ВС семейства Cray - 64), но время запуска значительно меньше чем в случае «память-память». Этот тип векторных систем гораздо более эффективен при обработке коротких векторов, но при операциях над длинными векторами векторные регистры должны загружаться сегментами несколько раз. В настоящее время ВП типа «регистр-регистр» доминируют на компьютерном рынке. Это

вычислительные системы фирмы Cray Research Inc., в частности модели Y-MP и C-90. Аналогичный подход заложен в системы фирм Fujitsu, Hitachi и NEC. Время цикла в современных ВП варьируется от 2,5 не (NEC SX-3) до 4,2 не (Cray C90), а производительность, измеренная по тесту LINPACK, лежит в диапазоне от 1000 до 2000 MFLOPS (от 1 до 2 GFLOPS).

Обработка длинных векторов и матриц

Аппаратура векторных процессоров типа «регистр-регистр» ориентирована на об работку векторов, длина которых совпадает с длиной векторных регистров (ВР), поэтому обработка коротких векторов не вызывает проблем - достаточно записать фактическую длину вектора в регистр длины вектора.

Если размер векторов превышает емкость ВР, используется техника разбиения исходного вектора на сегменты одинаковой длины, совпадающей с емкостью векторных регистров (последний сегмент может быть короче), и последовательной обработки полученных сегментов. В английском языке этот прием называют *strip-mining.* Процесс разбиения обычно происходит на стадии компиляции, но в ряде ВП данная процедура производится по ходу вычислений с помощью аппаратных средств на основе информации, хранящейся в регистре максимальной длины вектора.

Ускорение вычислений

Для повышения скорости обработки векторов все функциональные блоки векторных процессоров строятся по конвейерной схеме, причем так, чтобы каждая ступень любого из конвейеров справлялась со своей операцией за один такт (число ступеней в разных функциональных блоках может быть различным), В некоторых векторных ВС, например Cray C90, этот подход несколько усовершенствован - конвейеры во всех функциональных блоках продублированы (рис. 13.8).

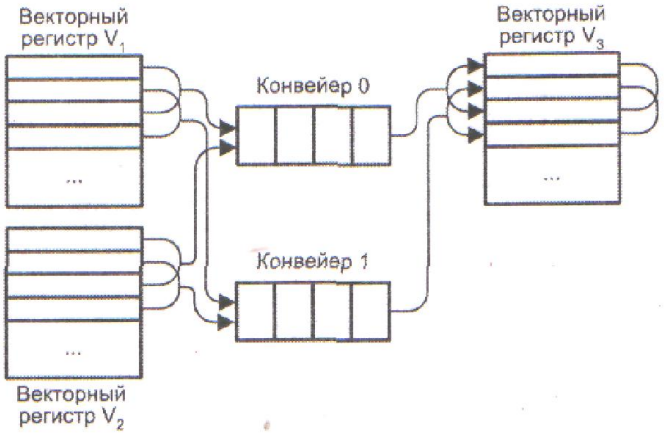


Рис. 13.8. Выполнение векторных операций при двух конвейерах

На конвейер 0 всегда подаются элементы векторов с четными номерами, а на конвейер 1 - с нечетными. В начальный момент на первую ступень конвейера 0 из ВРV$_1$ и V$_2$ поступают нулевые элементы векторов. Одновременно первые элементы векторов из этих регистров подаются на первую ступень конвейера 1. На следующем такте на конвейер 0 подаются вторые элементы из V$_1$, и V$_2$, а на конвейер 2 - третьи элементы и т. д. Аналогично происходит распределение результатов в выходном векторном регистре V$_3$. В итоге функциональный блок при максимальной загрузке в каждом такте выдает не один результат, а два. Добавим, что в скалярных операциях работает только конвейер 0.

Интересной особенностью некоторых ВП типа «регистр-регистр», например ВС фирмы Cray Research Inc., является так называемое *сцепление векторов* (vector chaining или vector linking), когда ВР результата одной векторной операции используется в качестве входного регистра для последующей векторной операции. Для примера рассмотрим последовательность из двух векторных команд, предполагая, что длина векторов равна 64: V$_2$ =V$_o$ x V$_1$, V$_4$ =V$_2$ + V$_3$.

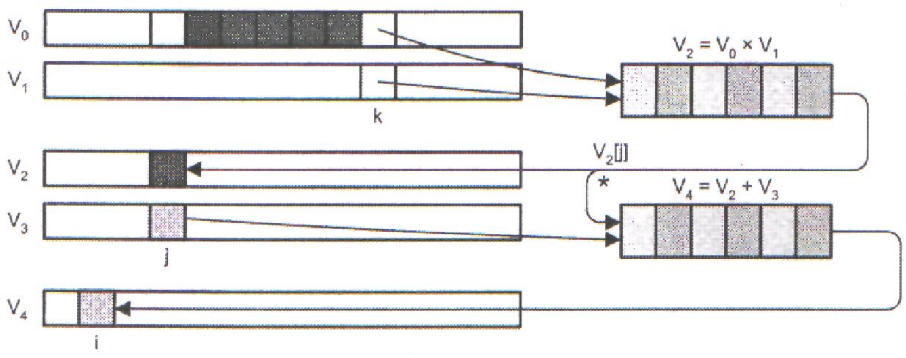


Рис. 1. 3. 9. Сцепление векторов

Результат первой команды служит операндом для второй. Напомним, что поскольку команды являются векторными, первая из них должна послать в конвейерный умножитель до 64 пар чисел. Примерно в середине выполнения команды складывается ситуация, когда несколько начальных элементов вектора $\mathbf{V_2}$ будут уже содержать недавно вычисленные произведения; часть элементов $\mathbf{V_2}$ все еще будет находиться в конвейере, а оставшиеся элементы операндов $\mathbf{V_o}$ и $\mathbf{V_1}$ еще остаются во входных векторных регистрах, ожидая загрузки в конвейер. Такая ситуация показана на рис. 13.9, где элементы векторов $\mathbf{V_o}$ и $\mathbf{V_1}$ находящиеся в конвейерном умножителе**,** имеют темную закраску. В этот момент система извлекает элементы $\mathbf{V_o[}k\mathbf{]}$ и $\mathbf{V_1[}k\mathbf{]}$ с тем, чтобы направить их на первую ступень конвейера, в то время как $\mathbf{V_2[}j\mathbf{]}$ покидает конвейер. Сцепление векторов иллюстрирует линия, обозначенная звездочкой. Одновременно с занесением $\mathbf{V_2[}j\mathbf{]}$ в ВР этот элемент направляется и в конвейерный сумматор, куда также подается и элемент $\mathbf{V_3[}j\mathbf{]}$**.** Как видно из рисунка, выполнение второй команды может начаться до завершения первой, и поскольку одновременно выполняются две команды, процессор формирует два результата за цикл $\mathbf{V_4[}i\mathbf{]}$ и $\mathbf{V_2[}j\mathbf{]}$ вместо одного. Без сцепления векторов пиковая производительность Сгау-1 была бы 80 MFLOPS (один полный конвейер производит результат каждые 12,5 нс). При сцеплении трех конвейеров теоретический пик производительности - 240 MFLOPS. В принципе сцепление векторов можно реализовать и *в* векторных процессорах типа «память-память», но для этого необходимо

повысить пропускную способность памяти. Без сцепления необходимы три «канала»: два для входных потоков операндов и один - для потока результата. При использовании сцепления требуется обеспечить пять каналов: три входных и два ВЫХОДНЫХ.

Завершая обсуждение векторных и векторно-конвейерных ВС, следует отметить, что с середины 90-х годов прошлого века этот вид ВС стал уступать свои позиции другим более технологичным видам систем. Тем не менее одна из последних разработок корпорации NEC (2002 год) - вычислительная система «Модель Земли» (The Earth Simulator), - являющаяся на сегодняшний момент самой производительной вычислительной системой в классе, по сути представляет собой векторно- конвейерную ВС. Система включает в себя 640 вычислительных узлов по 8 векторных процессоров в каждом. Пиковая производительность суперкомпьютера превышает 40 TFLOPS.

Принцип SIMD используется в современных процессорах путем встраивания в их кристаллы относительно самостоятельных блоков, обеспечивающих технологии MMX (Pentium, Pentium-II) и SSE (Pentium-III и выше), которые значительно ускоряют операции обработки изображений.

Процессоры с длинным командным словом.

Процессоры с длинным командным словом (VLIW) используют параллелизм на уровне команд. В соответствии с этой концепцией, как показано на рис.1.20, сравнительно длинная команда делится на множество полей и каждый операционный блок управляется отдельным полем.

В отличие от суперскалярных процессоров, где возможность распараллеливания операций выясняется в процессе их выполнения, в процессорах данного типа это выяснение происходит в ходе компиляции программы. Компилятор извлекает из программы команды, которые могут быть выполнены параллельно, и из них формируется одна команда.

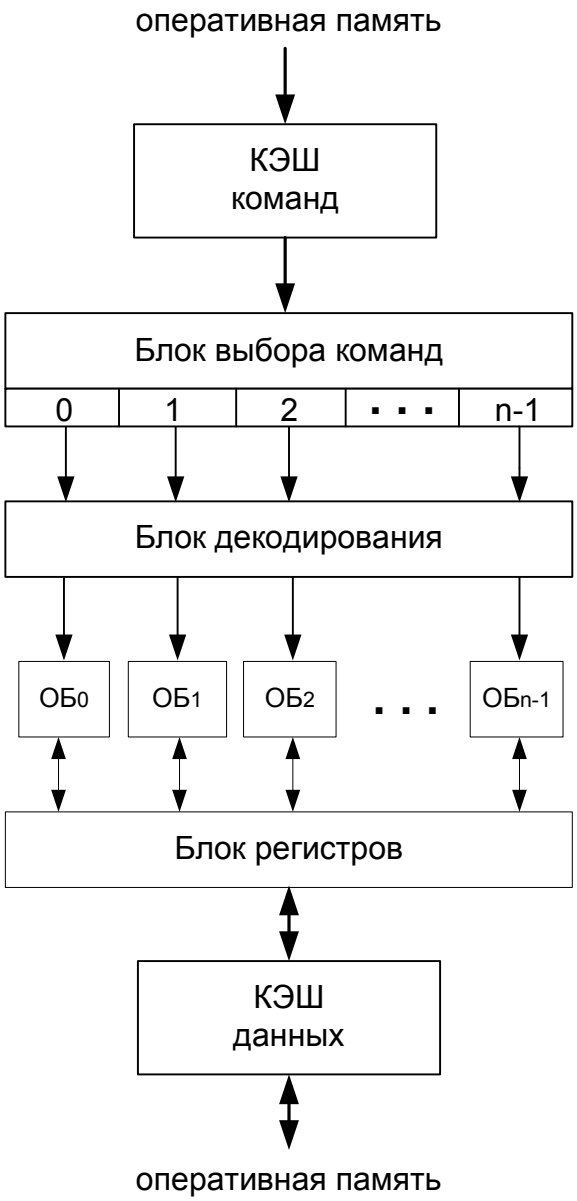


Рис.1.21. Структура VLIW- процессора.

Достоинства VLIW заключаются в следующем. Во-первых, компилятор может более эффектно выявлять зависимости между командами и выбирать параллельно исполняемые команды, чем это делает аппаратура суперскалярного процессора, ограниченная размером окна исполнения. При этом за счет обеспечения степени параллельности, близкой к числу операционных блоков, можно достичь высокой скорости обработки. Во-вторых, VLIW- процессор имеет более простое устройство управления и потенциально может иметь более высокую тактовую частоту.

Однако у VLIW процессоров есть серьезный фактор, снижающий их производительность. Это команды ветвления, зависящие от данных, значения которых становятся известны только в динамике вычислений. Число одновременно выполняющихся команд VLIW-процессора, не может быть очень большим в виду отсутствия у компилятора информации о зависимостях, формируемых динамически, в процессе выполнения.

• Отсутствует оборудование необходимое для выполнения время обнаружения параллелизма.

• Решена проблема окна исполнения, т.к. компилятор может потенциально анализировать всю программу в целях выявления параллельных операций.

Преимущества:

• Простота оборудования:

- Число операционных блоков может быть увеличено без дополнительных сложных аппаратных решений выявления параллелизма, как делается в суперскалярных процессорах.

• Хорошие компиляторы могут обнаружить параллелизм на основе глобального анализа всей программы (без окна выполнения задачи).

Проблемы:

• Чтобы поддерживать все операционные блоки в активном состоянии, требуется большое количество регистров для хранения операндов и результатов.

• Для передачи данных между КЭШем команд и блоком выборки команд, а также между операционными блоками и блоком регистров необходима большая ширина шины.

• Высокая пропускная способность между кэш инструкций и

извлечение устройства. Например, если команда содержит 7 операций, каждая из которых составляет 24 бита, то потребуется 168 бит /команду.

• Большой размер кода, отчасти потому, что неиспользованные операций ⇒ неиспользуемых битов в инструкции слова.

• Incomputability бинарного кода

Например:

Если для новой версии процессора дополнительные Фу вводятся ⇒ число операций можно выполнять параллельно увеличивается ⇒

Слово изменения инструкции ⇒ старый бинарный код не может будет работать на этом процессоре.

Матричные процессоры

Как показано на рис. 1.21, в архитектуре SIMD (Single Instruction Multiple Data Stream — один поток команд и много потоков данных) команда, выделяемая управляющим устройством, одновременно передается множеству операционных элементов с одинаковой структурой (ПЭ), и все операционные блоки параллельно выполняют одну и ту же операцию. Управляющее устройство разрешает или запрещает выполнение операций на основе информации о состоянии каждого операционного элемента. Информация о состоянии хранится в специальном внутреннем регистре ПЭ. Выполнение операций разрешается только тем процессорным элементам, в которых выполняются определенные условия.

Поскольку поток команд является одиночным, то в случае необходимости условного перехода по результатам проверки выполнения условий заданные операции выполняются только теми элементами, для которых результаты проверки подтверждают выполнение условий, а затем только теми элементами, для которых результаты проверки говорят о том, что условия не выполняются. Следовательно, сначала запрещается выполнение операций процессорным элементам с отрицательными

результатами проверки условий, а затем — элементам, в которых проверяемые условия выполняются.

Рис. 1.22. Архитектура матричного процессора.

Управляющее устройство контролирует взаимосвязь между всеми операционными блоками и управляет обращением к общей памяти. (Пример ILLIAC-IV)

Архитектура IA64

Эта архитектура, известная под названием Intel Architecture-64 (IA-64), полностью "порывает с прошлым". IA-64 не является как 64-разрядным расширением 32-разрядной архитектуры x86 компании Intel, так и переработкой 64-разрядной архитектуры PA-RISC компании HP. IA-64 представляет собой нечто абсолютно новое - передовую архитектуру,

использующую длинные слова команд (long instruction words -- LIW), предикаты команд (instruction predication), устранение ветвлений (branch elimination), предварительную загрузку данных (speculative loading) и другие ухищрения для того, чтобы "извлечь больше параллелизма" из кода программ.

Несмотря на то, что Intel и HP обещали добиться обратной совместимости с существующим программным обеспечением, работающим на процессорах архитектур x86 и PA-RISC, они до сих пор не разглашают, каким образом это будет сделано. На самом деле обеспечить такую совместимость совсем не просто; достаточно вспомнить гораздо менее кардинальный переход с 16-разрядной на 32-разрядную архитектуру x86, продолжавшийся 12 лет и до сих пор не завершённый.

По поводу совместимости, стоит заметить, что но в Merced на самом деле существует два режима декодирования команд VLIW и старый CISC. Т.е. программы переключаются в необходимый режим исполнения. В архитектуре x86 были добавлен ряд команд для перехода в новый режим, а также для передачи данных. В IA-64 такие команды есть изначально. Так что теперь ОС будут содержать и 64-х разрядную часть на IA-64 и старую 32-х разрядную.

Правда, переход к архитектуре IA-64 в ближайшее время вряд ли затронет большинство пользователей, поскольку Intel заявила, что Merced разрабатывается для серверов и рабочих станций класса high-end, а не для компьютеров среднего уровня. Фактически, компания заявила, что IA-64 не заменит x86 в ближайшем будущем. Похоже на то, что Intel и другие поставщики продолжат разрабатывать чипы x86.

Перед тем, как углубиться в технические детали, попробуем понять, почему Intel и HP рискнули пойти на столь кардинальные перемены. Причина сводится к следующему: они считают, что как CISC, так и RISC-архитектуры исчерпали себя.

Разработчики процессоров стремятся создавать чипы, содержащие как можно больше функциональных узлов - что позволяет обрабатывать больше команд параллельно - но одновременно приходится существенно усложнять управляющие цепи для распределения потока команд по обрабатывающим узлам. На данный момент лучшие процессоры не могут выполнять более четырёх команд одновременно, при этом управляющая логика занимает слишком много места на кристалле.

В то же время, последовательная структура кода программ и большая частота ветвлений делают задачу распределения потока команд крайне сложной. Современные процессоры содержат огромное количество управляющих элементов для того, чтобы минимизировать потери производительности, связанные с ветвлениями, и извлечь как можно больше "скрытого параллелизма" из кода программ. Они изменяют порядок команд во время исполнения программы, пытаются предсказать, куда необходимо будет перейти в результате очередного ветвления, и выполняют команды до вычисления условий ветвления. Если путь ветвления предсказан неверно, процессор должен сбросить полученные результаты, очистить конвейеры и загрузить нужные команды, что требует достаточно большого числа тактов. Таким образом, процессор, теоретически выполняющий четыре команды за такт, на деле выполняет менее двух.

Проблему ещё осложняет тот факт, что микросхемы памяти не успевают за тактовой частотой процессоров. Когда Intel разработала архитектуру x86, процессор мог извлекать данные из памяти с такой же скоростью, с какой он их обрабатывал. Сегодня процессор тратит сотни тактов на ожидание загрузки данных из памяти, даже несмотря на наличие большой и быстрой кэш-памяти.

Называют новую LIW-технологию Explicitly Parallel Instruction Computing или EPIC (Вычисления с Явной Параллельностью Инструкций, где "явной" означает явно указанной при трансляции).

Команды в формате IA-64 упакованы по три в 128-битный пакет для быстрейшей обработки. Каждый содержит шаблон (template) длиной в несколько бит, помещаемый в него компилятором, который указывает процессору, какие из команд могут выполняться параллельно. Теперь процессору не нужно будет анализировать поток команд в процессе выполнения для выявления "скрытого параллелизма". Вместо этого наличие параллелизма определяет компилятор и помещает информацию в код программы. Каждая команда (как для целочисленных вычислений, так и для вычислений с плавающей точкой) содержит три 7-битных поля регистра общего назначения (РОН). Из этого следует, что процессоры архитектуры IA-64 содержат 128 целочисленных РОН и 128 регистров для вычислений с плавающей точкой. Все они доступны программисту и являются регистрами с произвольным доступом (programmer-visible random-access registers). По сравнению с процессорами x86, у которых всего восемь целочисленных РОН и стек глубины 8 для вычислений с плавающей точкой, IA-64 намного "шире" и, соответственно, будет намного реже простаивать из-за "нехватки регистров".

Компиляторы для IA-64 будут использовать технологию "отмеченных команд" (predication) для устранения потерь производительности из-за неправильно предсказанных переходов и необходимости пропуска участков кода после ветвлений. Когда процессор встречает "отмеченное" ветвление в процессе выполнения программы, он начинает одновременно выполнять все ветви. После того, как будет определена "истинная" ветвь, процессор сохраняет необходимые результаты и сбрасывает остальные.

Компиляторы для IA-64 будут также просматривать исходный код с целью поиска команд, использующих данные из памяти. Найдя такую команду, они будут добавлять пару команд - команду предварительной загрузки (speculative loading) и проверки загрузки (speculative check). Во время выполнения программы первая из команд загружает данные в память до того, как они понадобятся программе. Вторая команда проверяет, успешно

ли произошла загрузка, перед тем, как разрешить программе использовать эти данные. Предварительная загрузка позволяет уменьшить потери производительности из-за задержек при доступе к памяти, а также повысить параллелизм.

Из всего вышесказанного следует, что компиляторы для процессоров архитектуры IA-64 должны быть намного "умнее" и лучше знать микроархитектуру процессора, код для которого они вырабатывают. Существующие чипы, в том числе и RISC-процессоры, производят гораздо больше оптимизации на этапе выполнения программ, даже при использовании оптимизирующих компиляторов. IA-64 перекладывает практически всю работу по оптимизации потока команд на компилятор. Таким образом, программы, скомпилированные для одного поколения процессоров архитектуры IA-64, на процессорах следующего поколения без перекомпиляции могут выполняться неэффективно. Это ставит перед поставщиками нелёгкую задачу по выпуску нескольких версий исполняемых файлов для достижения максимальной производительности.

Другим не очень приятным следствием будет увеличение размеров кода, так как команды IA-64 длиннее, чем 32-битные RISC-команды (порядка 40 бит). Компиляция при этом будет занимать больше времени, поскольку IA-64, как уже было сказано, требует от компилятора гораздо больше действий. Intel и HP заявили, что уже работают совместно с поставщиками средств разработки над переработкой этих программных продуктов.

Технология "отмеченных команд" является наиболее характерным примером "дополнительной ноши", перекладываемой на компиляторы. Эта технология является центральной для устранения ветвлений и управления параллельным выполнением команд.

Обычно компилятор транслирует оператор ветвления (например, IF-THEN-ELSE) в блоки машинного кода, расположенные последовательно в потоке. В зависимости от условий ветвления процессор выполняет один из

этих блоков и перескакивает через остальные. Современные процессоры стараются предсказать результат вычисления условий ветвления и предварительно выполняют предсказанный блок. При этом в случае ошибки много тактов тратится впустую. Сами блоки зачастую весьма малы - две или три команды, - а ветвления встречаются в коде в среднем каждые шесть команд. Такая структура кода делает крайне сложным его параллельное выполнение.

Когда компилятор для IA-64 находит оператор ветвления в исходном коде, он исследует ветвление, определяя, стоит ли его "отмечать". Если такое решение принято, компилятор помечает все команды, относящиеся к одному пути ветвления, уникальным идентификатором, называемым предикатом (predicate). Например, путь, соответствующий значению условия ветвления TRUE, помечается предикатом P1, а каждая команда пути, соответствующего значению условия ветвления FALSE - предикатом P2. Система команд IA-64 определяет для каждой команды 6-битное поле для хранения этого предиката. Таким образом, одновременно могут быть использованы 64 различных предиката. После того, как команды "отмечены", компилятор определяет, какие из них могут выполняться параллельно. Это опять требует от компилятора знания архитектуры конкретного процессора, поскольку различные чипы архитектуры IA-64 могут иметь различное число и тип функциональных узлов. Кроме того, компилятор, естественно, должен учитывать зависимости в данных (две команды, одна из которых использует результат другой, не могут выполняться параллельно). Поскольку каждый путь ветвления заведомо не зависит от других, какое-то "количество параллелизма" почти всегда будет найдено.

Заметим, что не все ветвления могут быть отмечены: так, использование динамических методов вызова приводит к тому, что до этапа выполнения невозможно определить, возникнет ли исключение. В других случаях применение этой технологии может привести к тому, что будет затрачено больше тактов, чем сэкономлено.

После этого компилятор транслирует исходный код в машинный и упаковывает команды в 128-битные пакеты. Шаблон пакета (bundle's template field) указывает не только на то, какие команды в пакете могут выполняться независимо, но и какие команды из следующего пакета могут выполняться параллельно. Команды в пакетах не обязательно должны быть расположены в том же порядке, что и в машинном коде, и могут принадлежать к различным путям ветвления. Компилятор может также помещать в один пакет зависимые и независимые команды, поскольку возможность параллельного выполнения определяется шаблоном пакета. В отличие от некоторых ранее существовавших архитектур со сверхдлинными словами команд (VLIW), IA-64 не добавляет команд "нет операции" (NOPS) для дополнения пакетов.

Во время выполнения программы IA-64 просматривает шаблоны, выбирает взаимно независимые команды и распределяет их по функциональным узлам. После этого производится распределение зависимых команд. Когда процессор обнаруживает "отмеченное" ветвление, вместо попытки предсказать значение условия ветвления и перехода к блоку, соответствующему предсказанному пути, процессор начинает параллельно выполнять блоки, соответствующие всем возможным путям ветвления. Таким образом, на машинном уровне ветвления нет.

Разумеется, в какой-то момент процессор наконец вычислит значение условия ветвления в нашем операторе IF-THEN-ELSE. Предположим, оно равно TRUE, следовательно, правильный путь отмечен предикатом P1. 6-битному полю предиката соответствует набор из 64 предикатных регистров (predicate registers) P0-P63 длиной 1 бит. Процессор записывает 1 в регистр P1 и 0 во все остальные.

К этому времени процессор, возможно, уже выполнил некоторое количество команд, соответствующих обоим возможным путям, но до сих пор не сохранил результат. Перед тем, как сделать это, процессор проверяет соответствующий предикатный регистр. Если в нём 1 - команда верна и

процессор завершает её выполнение и сохраняет результат. Если 0 - результат сбрасывается.

Технология "отмеченных команд" существенно снижает негативное влияние ветвлений на машинном уровне. В то же время, если компилятор не "отметил" ветвление, IA-64 действует практически так же, как и современные процессоры: пытается предсказать путь ветвления и т.д. Испытания показали, что описанная технология позволяет устранить более половины ветвлений в типичной программе, и, следовательно, уменьшить более чем в два раза число возможных ошибок в предсказаниях.

Другой ключевой особенностью IA-64 является предварительная загрузка данных. Она позволяет не только загружать данные из памяти до того, как они понадобятся программе, но и генерировать исключение только в случае, если загрузка прошла неудачно. Цель предварительной загрузки - разделить собственно загрузку и использование данных, что позволяет избежать простоя процессора. Как и в технологии "отмеченных команд" здесь также сочетается оптимизация на этапе компиляции и на этапе выполнения.

Сначала компилятор просматривает код программы, определяя команды, использующие данные из памяти. Везде, где это возможно, добавляется команда предварительной загрузки на достаточно большом расстоянии перед командой, использующей данные и команда проверки загрузки непосредственно перед командой, использующей данные.

На этапе выполнения процессор сначала обнаруживает команду предварительной загрузки и, соответственно, пытается загрузить данные из памяти. Иногда попытка оказывается неудачной - например, команда, требующая данные, находится после ветвления, условия которого ещё не вычислены. "Обычный" процессор тут же генерирует исключение. IA-64 откладывает генерацию исключения до того момента, когда встретит соответствующую команду проверки загрузки. Но к этому времени условия

ветвления, вызывавшего исключение, уже будут вычислены. Если команда, инициировавшая предварительную загрузку, относится к неверному пути, загрузка признается неудачной и генерируется исключение. Если же путь верен, то исключение вообще не генерируется. Таким образом, предварительная загрузка в архитектуре IA-64 работает аналогично структуре типа TRY-CATCH.

Возможность располагать команду предварительной загрузки до ветвления очень существенна, так как позволяет загружать данные задолго до момента использования (напомню, что в среднем каждая шестая команда является командой ветвления).

Потоковые процессоры

Потоковыми называют процессоры, в основе работы которых лежит принцип обработки многих данных с помощью одной команды. Согласно классификации Флинна, они принадлежат к SIMD (single instruction stream / multiple data stream) архитектуре. Технология SIMD позволяет выполнять одно и то же действие, например, вычитание и сложение, над несколькими наборами чисел одновременно. SIMD-операции для чисел двойной точности с плавающей запятой ускоряют работу ресурсоемких приложений для создания контента, трехмерного рендеринга, финансовых расчетов и научных задач. Кроме того, усовершенствованы возможности 64-разрядной технологии MMX (целочисленных SIMD-команд); эта технология распространена на 128-разрядные числа, что позволяет ускорить обработку видео, речи, шифрование, обработку изображений и фотографий. Потоковый процессор повышает общую производительность, что особенно важно при работе с 3D-графическими объектами.

Может быть отдельный потоковый процессор (Single-streaming processor — SSP) и многопотоковый процессор (Multi-Streaming Processor – MSP).

Ярким представителем потоковых процессоров является семейство процессоров Intel, начиная с Pentium III, в основе работы которых лежит технология Streaming SIMD Extensions (SSE, потоковая обработка по принципу «одна команда – много данных»). Эта технология позволяет выполнять такие сложные и необходимые в век Internet задачи как

обработка речи, кодирование и декодирование видео- и аудиоданных, разработка трехмерной графики и обработка изображений.

Представителями класса SIMD считаются матрицы процессоров: ILLIAC IV, ICL DAP, Goodyear Aerospace MPP, Connection Machine 1 и т.п. В таких системах единое управляющее устройство контролирует множество процессорных элементов. Каждый процессорный элемент получает от устройства управления в каждый фиксированный момент времени одинаковую команду и выполняет ее над своими локальными данными.

Другими представителями SIMD-класса являются векторные процессоры, в основе которых лежит векторная обработка данных. Векторная обработка увеличивает производительность процессора за счет того, что обработка целого набора данных (вектора) производится одной командой. Векторные компьютеры манипулируют массивами сходных данных подобно тому, как скалярные машины обрабатывают отдельные элементы таких массивов. В этом случае каждый элемент вектора надо рассматривать как отдельный элемент потока данных. При работе в векторном режиме векторные процессоры обрабатывают данные практически параллельно, что делает их в несколько раз более быстрыми, чем при работе в скалярном режиме. Максимальная скорость передачи данных в векторном формате может составлять 64 Гбайт/с, что на 2 порядка быстрее, чем в скалярных машинах. Примерами систем подобного типа являются, например, процессоры фирм NEC и Hitachi.

Процессор Intel Pentium 4

Основные особенности процессора Pentium 4 связаны с его микроархитектурой. *Микроархитектура процессора* определяет реализацию его внутренней структуры, принципы выполнения поступающих команд, способы размещения и обработки данных. Новая микроархитектура процессора Pentium 4, получившая название NetBurst (пакетно-сетевая), ориентирована на эффективную работу с Интернет-приложениями. Необходимо отметить, что в микроархитектуре NetBurst реализованы многие принципы, использованные в предыдущей модели Pentium III (микроархитектура P6). Характерными чертами этой микроархитектуры являются:

- гарвардская структура с разделением потоков команд и данных;
- суперскалярная архитектура, обеспечивающая одновременное выполнение нескольких команд в параллельно работающих исполнительных устройствах;
- динамическое изменение последовательности команд (выполнение команд с опережением — спекулятивное выполнение);
- конвейерное исполнение команд;
- предсказание направления ветвлений.

Практическая реализация данных принципов в структуре процессора Pentium 4 имеет ряд существенных особенностей (рис. 4).

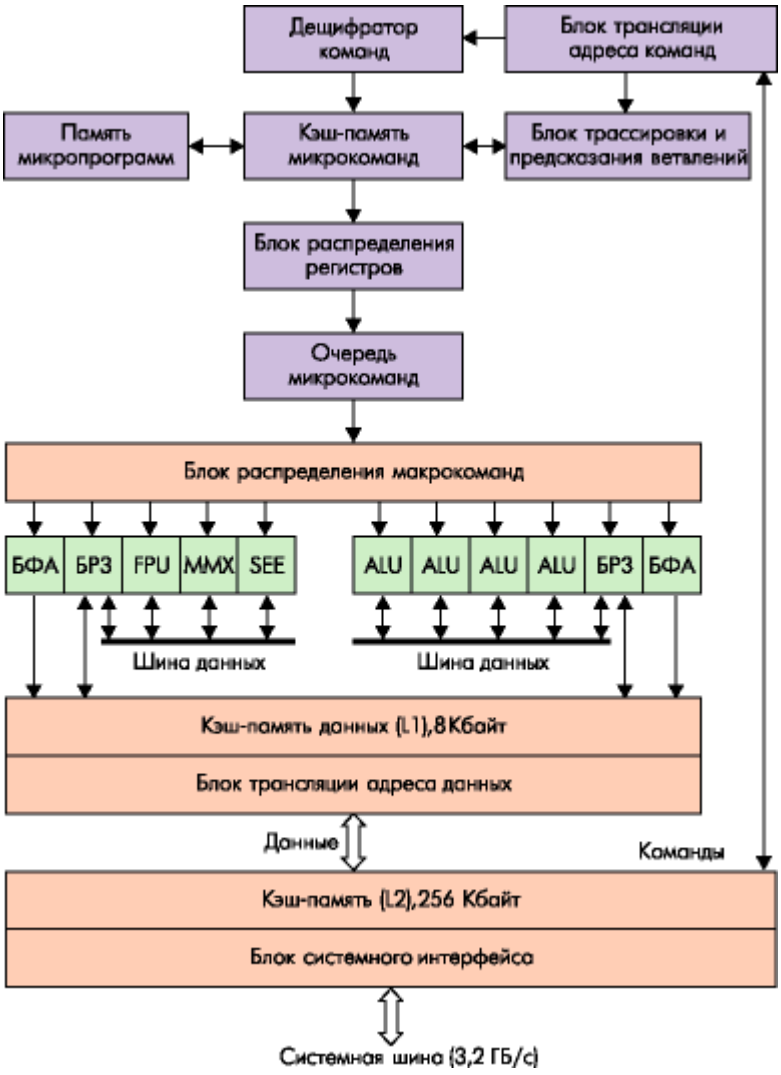


Рис. 4. Обобщенная структура Pentium 4

Гарвардская внутренняя структура реализуется путём разделения потоков команд и данных, поступающих от системной шины через блок внешнего интерфейса и размещённую на кристалле процессора общую кэш-память 2-го уровня (L2- Advanced Transfer Cache) ёмкостью 256 Кбайт. Такое размещение позволяет сократить время выборки команд. Так как Pentium 4 рассчитан на обработку потоковых данных, скорость работы L2-кеша для него является одним из ключевых моментов. Поэтому, Intel увеличил пропускную способность кеша второго уровня в Pentium 4 в два раза. Это усовершенствование было сделано благодаря передаче данных из L2-кеша в каждом процессорном такте. Пропускная способность L2-кеша Pentium 4, работающего с частотой 1.4 ГГц имеет 44.8 Мбайт/с. L2 кэш имеет восемь областей ассоциативности и строки длиной 128 байт.

Блок внешнего интерфейса реализует обмен процессора с системной шиной, к которой подключается память, контроллеры ввода/вывода и другие активные устройства системы. Обмен по системной шине осуществляется с помощью 64-разрядной двунаправленной шины данных, 41-разрядной шины адреса, обеспечивающей адресацию до 64 Гбайт внешней памяти.

Дешифратор команд работает вместе с памятью микропрограмм, формируя последовательность микрокоманд, обеспечивающих выполнение поступивших команд. Декодированные команды загружаются в кэш-память микрокоманд, откуда они выбираются для исполнения. Вместо обычного L1 кеша, который в более ранних процессорах был разделен на область команд и область данных в Pentium 4 применен новый подход. Команды в L1 кэше не сохраняются, он предназначен теперь только для данных. Для кэширования инструкций теперь используется специальный кэш трассировки (Trace Cache), однако по сравнению с обычным L1-кешем он имеет много преимуществ, направленных на минимизацию простоев процессора при выполнении неправильных предсказаний переходов. В кэш трассировки сохраняются уже декодированные инструкции, т.е. в нем хранятся не классические x86 инструкции, а так называемые микрокоманды, более простые операции которыми непосредственно оперирует процессорное ядро. Сохранение в кэш трассировки микроопераций позволяет избежать повторного декодирования x86 инструкций при повторном выполнении того же участка программы или при неправильном предсказании переходов. Итак, после заполнения кэш трассировки практически любая команда будет храниться в ней в декодированом виде. Поэтому при поступлении очередной команды блок трассировки выбирает из этой кэш-памяти необходимые микрокоманды, обеспечивающие её выполнение.

Микрооперации в кэш трассировки сохраняются именно в том порядке, в каком они выполняются. Если в потоке команд оказывается команда условного перехода (ветвления программы), то включается механизм

предсказания ветвления, который формирует адрес следующей выбираемой команды до того, как будет определено условие выполнения перехода. Вероятность того, что переходы предсказываются неправильно, достаточно мала для того, чтобы отказаться от очевидного выигрыша, получаемого путем отказа от повторных декодирований и предсказаний переходов. Кэш-память может хранить до 12000 микрокоманд. После формирования потоков микрокоманд производится выделение регистров, необходимых для выполнения декодированных команд. Эта процедура реализуется блоком распределения регистров, который выделяет для каждого указанного в команде логического регистра (регистра целочисленных операндов EAX, ECX и других, регистра операндов с плавающей точкой ST0-ST7 или регистра блоков MMX, SSE, рис. 2) один из 128 физических регистров, входящих в состав блоков регистров замещения (БРЗ). Эта процедура позволяет выполнять команды, использующие одни и те же логические регистры, одновременно или с изменением их последовательности. Выбранные микрокоманды размещаются в очереди микрокоманд. В ней содержатся микрокоманды, реализующие выполнение 126 поступивших и декодированных команд, которые затем направляются в исполнительные устройства по мере готовности операндов. Отметим, что в процессорах Pentium III в очереди находятся микрокоманды для 40 поступивших команд. Значительное увеличение числа команд, стоящих в очереди, позволяет более эффективно организовать поток их исполнения, изменяя последовательность выполнения команд и выделяя команды, которые могут выполняться параллельно. Эти функции реализует блок распределения микрокоманд. Он выбирает микрокоманды из очереди не в порядке их поступления, а по мере готовности соответствующих операндов и исполнительных устройств. В результате команды, поступившие позже, могут быть выполнены до ранее выбранных команд (внеочередное выполнение). При этом реализуется одновременное выполнение нескольких микрокоманд (команд) в параллельно работающих исполнительных устройствах. Таким образом,

естественный порядок следования команд нарушается, чтобы обеспечить более полную загрузку параллельно включенных исполнительных устройств и повысить производительность процессора.

Архитектура многопроцессорных систем

Классификация многопроцессорных систем

Многопроцессорные системы по классификации Флинна относятся к архитектурам типа MIMD(Multiple Instruction Multipl Data) с множественным потоком команд при множественном потоке данных (МКМД). В многопроцессорной системе каждый процессор выполняет свою программу достаточно независимо от других процессоров. Процессоры в ходе решения общей задачи должны связываться друг с другом в соответствии с графом взаимодействия её параллельных ветвей. Это вызывает необходимость более подробно производить классификацию систем типа MIMD.

В многопроцессорных системах с общей памятью (сильносвязанных) имеется память данных и команд, доступная всем процессорам. С общей памятью процессоры связываются с помощью коммуникационной среды, основой которой может быть либо общая шина (ОШ), либо множество шин (МШ), либо перекрёстный коммутатор (ПК).

В противоположность этому варианту в слабосвязанных многопроцессорных системах (машинах с распределённой памятью) вся память разделена между процессорами и каждый блок памяти доступен только локальному процессору. Память и процессор образуют фактически независимые вычислительные модули (вычислительные узлы), которые связываются между собой при помощи высокоскоростной сети обмена с коммутацией сообщений.

Сообщение – это блок информации, сформированный процессом-отправителем таким образом, чтобы он был понятен процессу- получателю. Сообщение состоит из заголовка фиксированной длины и набора данных определённого типа обычно переменной длины. В заголовок, как правило, включают следующую информацию:

- адрес - это поле, предназначенное для идентификации процессоров (вычислительных узлов), участвующих в процедуре обмена. Адрес процессора или вычислительного узла является уникальным и состоит из двух частей - адреса процессора - отправителя и адреса процессора - получателя;
- управляющие поля, в которые могут входить символы синхронизации, отмечающие начало и конец передаваемого блока (кадра) данных, символы, обозначающие тип данных, длину передаваемых данных и др.

В качестве топологической модели коммуникационной среды применяют линейные или кольцевые моноканалы, звездообразную конфигурацию, плоскую решётку, n- мерный тор, либо n- кубическую (гиперкубическую) сеть.

Базовой моделью вычислений на MIMD-системах является совокупность независимых процессов, время от времени обращающихся к разделяемым данным. Основана она на распределенных вычислениях, в которых программа делится на довольно большое число параллельных задач.

В настоящее время всё большее внимание разработчики проявляют к архитектурам типа MIMD. Это находит объяснение главным образом существованием двух факторов:

1) в архитектурах MIMD используются высокотехнологичные, дешёвые, выпускаемые массово микропроцессоры, что позволяет оптимизировать соотношение стоимость/производительность;

2) архитектура MIMD дает большую гибкость и при наличии соответствующей поддержки со стороны аппаратных средств и программного обеспечения, поскольку может работать и как однопрограммная система, обеспечивая высокопроизводительную обработку данных для одной прикладной задачи, и как многопрограммная, выполняющая множество задач параллельно, а также как некоторая комбинация этих возможностей.

Одной из отличительных особенностей многопроцессорной вычислительной системы является коммуникационная среда, с помощью которой процессоры соединяются друг с другом или с памятью. Топология коммуникационной среды настолько важна для многопроцессорной системы, что многие характеристики производительности и другие оценки выражаются отношением времени обработки к времени обмена, которые в общем случае зависят от алгоритмов решаемых задач и порождаемых ими вычислительных процессов.

Существуют две основные модели межпроцессорного обмена: одна основана на передаче сообщений, другая - на использовании общей памяти.

В многопроцессорных системах с общей памятью один процессор осуществляет запись в конкретную ячейку, а другой процессор производит считывание из этой ячейки памяти. Чтобы обеспечить согласованность данных и синхронизацию процессов, обмен часто реализуется по принципу взаимно исключающего доступа к общей памяти методом "почтового ящика".

В архитектурах с распределённой памятью непосредственное разделение памяти невозможно. Вместо этого процессоры получают доступ к совместно используемым данным посредством передачи сообщений по сети

обмена. Эффективность схемы коммуникаций зависит от протоколов обмена, каналов обмена и пропускной способности памяти. Такие системы часто называют системами с передачей сообщений.

Каждый из этих механизмов обмена имеет свои преимущества. Для обмена в общей памяти это включает:

- совместимость с хорошо понятными и используемыми в однопроцессорных системах механизмами взаимодействия процессора с основной памятью;
- простота программирования, особенно это заметно в тех случаях, когда процедуры обмена между процессорами сложные или динамически меняются во время выполнения. Подобные преимущества упрощают конструирование компилятора;
- более низкая задержка обмена и лучшее использование полосы пропускания при обмене малыми порциями данных;
- возможность использования аппаратно управляемого кэширования для снижения частоты удаленного обмена, допускающая кэширование всех данных как разделяемых, так и неразделяемых.

Основные преимущества обмена с помощью передачи сообщений являются:

- аппаратура может быть более простой, особенно по сравнению с моделью разделяемой памяти, которая поддерживает масштабируемую когерентность кэш-памяти;
- процедуры обмена понятны, принуждают программистов (или компиляторы) уделять внимание обмену, который обычно имеет высокую, связанную с ним, стоимость.

Часто, в системах с общей памятью затраты времени на обмен не учитываются, так как проблемы обмена в значительной степени скрыты от программиста. Однако накладные расходы на обмен в этих системах имеются и определяются в основном конфликтами при доступе процессоров и других устройств к общим шинам и блокам основной памяти. Чем больше процессоров добавляется в систему, тем больше процессов соперничают при использовании одних и тех же данных и шины, что может привести к значительным задержкам хода вычислительного процесса и, следовательно, к потерям общей производительности. Причём с увеличением числа процессоров в системе возможно появление эффекта насыщения при котором рост числа процессоров не приведёт к росту производительности, и даже наоборот к её падению. Модель системы с общей памятью очень удобна для программирования, поскольку пользователь практически не задумывается о процедурах распараллеливания (они большей частью ложаться на

компилятор) и процедурах взаимодействия параллельных процессов (они производятся аппаратными средствами посредством общей памяти).

В сетях с коммутацией сообщений по мере возрастания требований к обмену следует учитывать возможность перегрузки сети. Здесь межпроцессорный обмен связывает сетевые ресурсы: каналы, процессоры, буферы сообщений. Объем передаваемой информации может быть сокращен за счет тщательного разбиения задачи на параллельные ветви и тщательного диспетчирования процесса их исполнения.

Таким образом, существующие MIMD-системы распадаются на два основных класса в зависимости от количества объединяемых процессоров, которое определяет и способ организации памяти и методику их межсоединений.

К первому классу относятся системы с разделяемой (общей) основной памятью (Shared Memory multiProcessing, SMP), объединяющие до нескольких (2-16) процессоров, число которых зависит от типа применяемой коммуникационной среды. Сравнительно небольшое количество процессоров в таких системах позволяет иметь одну централизованную общую память и зачастую объединить процессоры и память с помощью лишь одной шины. При наличии у процессоров кэш-памяти достаточного объема высокопроизводительная шина и общая память могут удовлетворить обращения к памяти, поступающие от нескольких процессоров.

Поскольку имеется единственная память с одним и тем же временем доступа, эти системы называют симметричными, а иногда - UMA (Uniform Memory Access). Симметричная архитектура предполагает однородность процессоров и единообразную схему их включения в многопроцессорную систему. Такой способ организации со сравнительно небольшой разделяемой памятью в настоящее время является наиболее популярным. Структура подобной системы представлена на рис. 2.1.

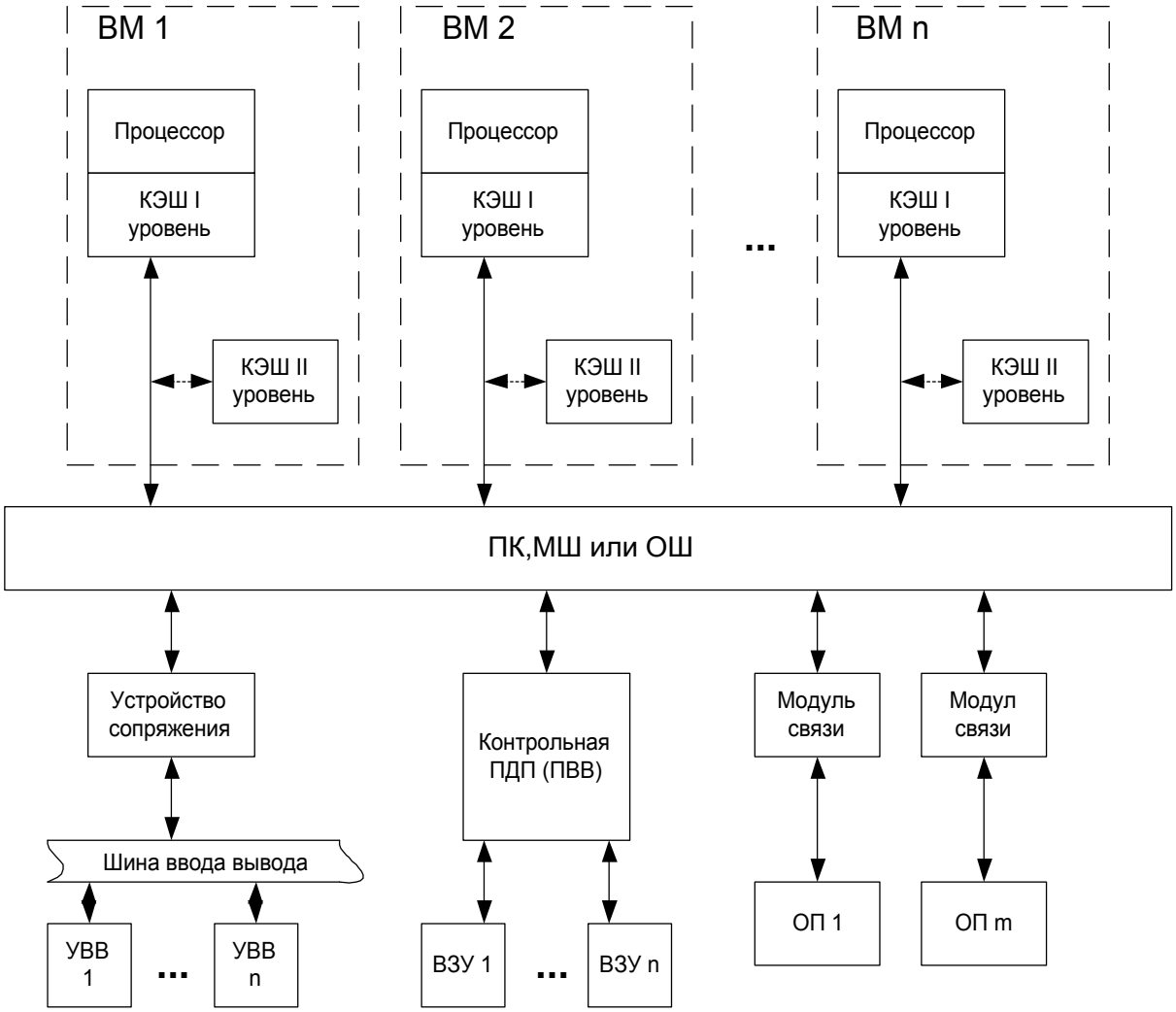


Рис. 2.1. Архитектура многопроцессорной системы с разделяемой (общей) памятью.

Второй класс составляют крупномасштабные вычислительные системы с распределенной памятью. Подобные ВС получили название систем с массовым параллелизмом (Mass-Parallel Processing, MPP). Для того чтобы подключать в систему большое количество процессоров необходимо физически разделять основную память и распределять её между ними. В противном случае пропускной способности памяти просто может не хватить для удовлетворения запросов, поступающих от очень большого числа процессоров. Естественно при таком подходе также требуется реализовать связь процессоров между собой. На рис. 2.2 показана структура такой системы.

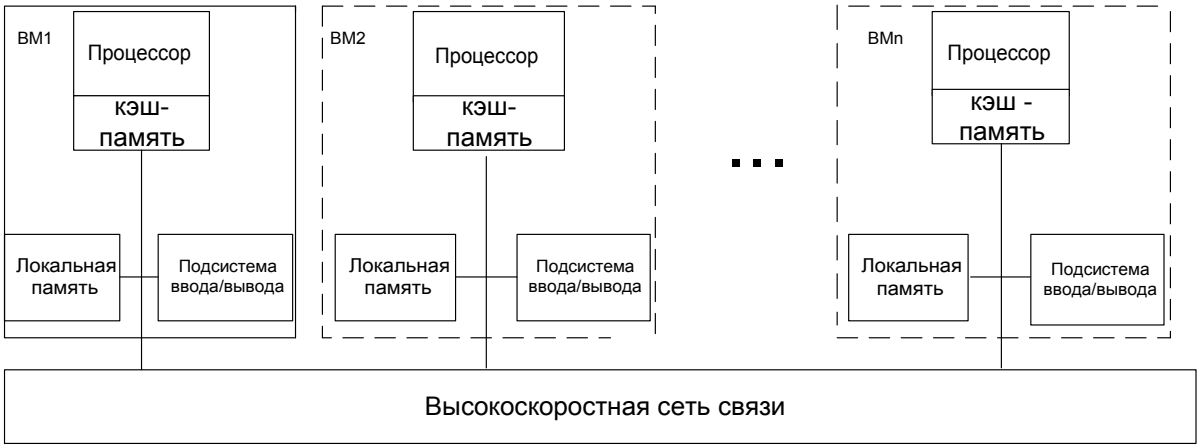


Рис. 2.2. Архитектура многопроцессорной системы с распределенной памятью.

Адресное пространство в таких системах состоит из отдельных адресных пространств, которые логически не связаны, и доступ к которым не может быть осуществлен аппаратно другим процессором. Фактически каждый модуль процессор-память представляет собой отдельный компьютер, поэтому такая структура в какой-то степени приближена к многопроцессорным системам.

MPP - система менее эффективна с точки зрения пользователя из-за усложнённой процедуры программирования, которая связана с применением специальных коммуникационных библиотек для организации взаимодействия между вычислительными узлами (процессами). Необходимость же реализации модели распределенной памяти объясняется тем, что масштабируемость (способность системы к наращиванию числа процессоров) систем с общей памятью ограничена пропускной способностью памяти и коммуникационной среды.

Вообще распределение памяти между отдельными узлами системы имеет два главных преимущества. Во-первых, это эффективный с точки зрения стоимости способ увеличения пропускной способности памяти, поскольку большинство обращений могут выполняться параллельно к локальной памяти в каждом узле. Во- вторых уменьшается задержка обращения к локальной памяти из-за отсутствия конфликтов при доступе к ней. Поэтому совершенно естественно появление промежуточного класса систем, объединяющего достоинства первого и второго классов. Память в таких системах распределена по вычислительным узлам и одновременно является доступной для всех процессоров. Такие ВС называются системами с распределенной разделяемой (общей) памятью (DSM - Distributed Shared Memory), а иногда NUMA (Non-Uniform Memory Access), поскольку время

доступа зависит от расположения данных в подсистеме памяти (Рис.2.3). Если данные находятся в локальной памяти местного вычислительного узла, то время доступа к ним минимально, если в локальной памяти удалённого вычислительного узла, то время доступа увеличивается в несколько раз.

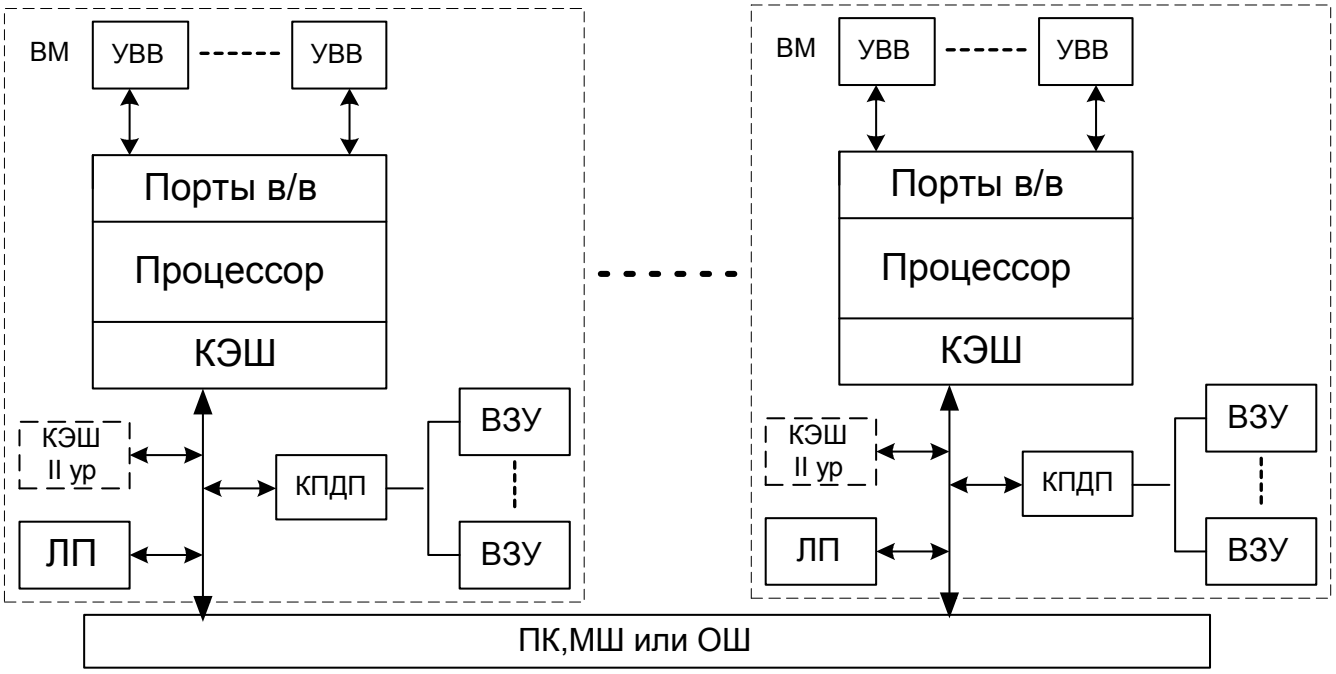


Рис.2.3. Архитектура многопроцессорной системы с распределённой разделяемой памятью.

Хотя структурно DSM и MPP-системы сходны, однако технически они реализуются по-разному. В DSM-системах физически отдельные, распределённые по вычислительным узлам, устройства памяти могут представляться логически как единое адресное пространство, что означает, что любой процессор может выполнять обращения к любым ячейкам памяти, предполагая, что он имеет соответствующие права доступа. Коммуникационная же среда и вовсе другая, так как она полностью соответствует структурам SMP-систем. Поскольку, в связи с принципами локальности, вычислительный процесс в основном развивается внутри вычислительного узла и редко обращается к удалённой памяти, что резко снижает объём передаваемых данных по коммуникационной среде, то масштабируемость таких систем повышается по сравнению с SMP-системами, и число процессоров может достигать 32-х.

Что касается устройств ввода/вывода и внешних запоминающих устройств, то они также как и память, либо распределяются по узлам, либо находятся в общем пользовании.

2.2 Организация коммуникационной среды в системах с разделяемой памятью.

Коммуникационную среду, реализующую множество соединений между процессорами или вычислительными узлами в многопроцессорных системах, называют коммутатором. По способу реализации различают коммутаторы с временной и пространственной коммутацией.

При временной коммутации передача информации осуществляется методами разделения времени или мультиплексирования. Такой коммутатор называется в простейшем случае общей шиной (ОШ). Структура многопроцессорной системы с общей шиной представлена на рис. 2.4.

В каждый момент времени ОШ способна, передавать лишь одно сообщение, т. е. она представляет собой разделенную во времени шину. Это говорит о возможности возникновения конфликтных ситуаций тогда, когда нескольким модулям одновременно необходимо связаться со своими абонентами - вычислительными узлами. С целью разрешения конфликтов

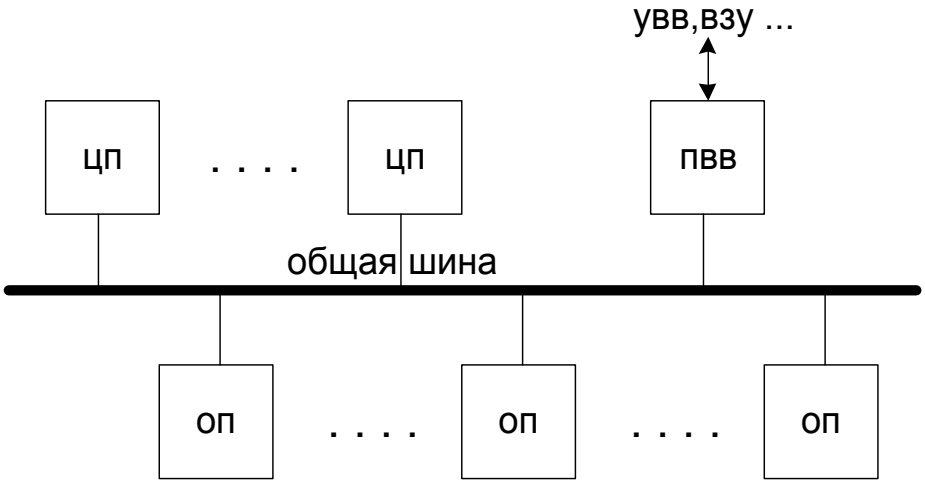


Рис.2.4. Структура системы с коммуникационной средой на основе общей шины.

Всем абонентам, могут быть назначены определенные приоритеты. Свободная общая шина удовлетворяет запросы абонента (процессора, периферийных устройств и др.) с наивысшим приоритетом.

Процедура занятия ОШ абонентами, являющимися источниками информации, заключается в их "борьбе" за ОШ. Достоинством коммутаторов с ОШ является простота организаций и гибкость (простое добавление и изъятие модулей). Очевидно, что такой коммутатор не может обеспечить

высокую пропускную способность. Для всех потоков информации здесь только один путь, поэтому временные задержки при передаче данных значительны. Из-за этого оказываются низкими общая производительность, поскольку различные пары абонентов не могут работать одновременно, и надежность системы, так как отказ в единственном пути передачи информации ведет к отказу всей системы. Пространственную коммутацию осуществляет перекрестный коммутатор (ПК). Он представляет собой многополюсник с M входами и N выходами, допускающими одновременное установление любого количества соединений между заданными входами и выходами.

Структуру системы с перекрестным коммутатором можно представить так как показано на рис. 2.5. Между любыми двумя абонентами здесь устанавливается физический контакт на все время передачи информации. При этом возможные конфликты между модулями разрешаются в логических схемах коммутационной матрицы, что существенно усложняет аппаратуру коммутаторов. В отличие от систем с ОШ, где во-первых коммутационного оборудования много меньше, а во-вторых эти функции могут быть возложены на вычислительные узлы или на центральный арбитр шины.

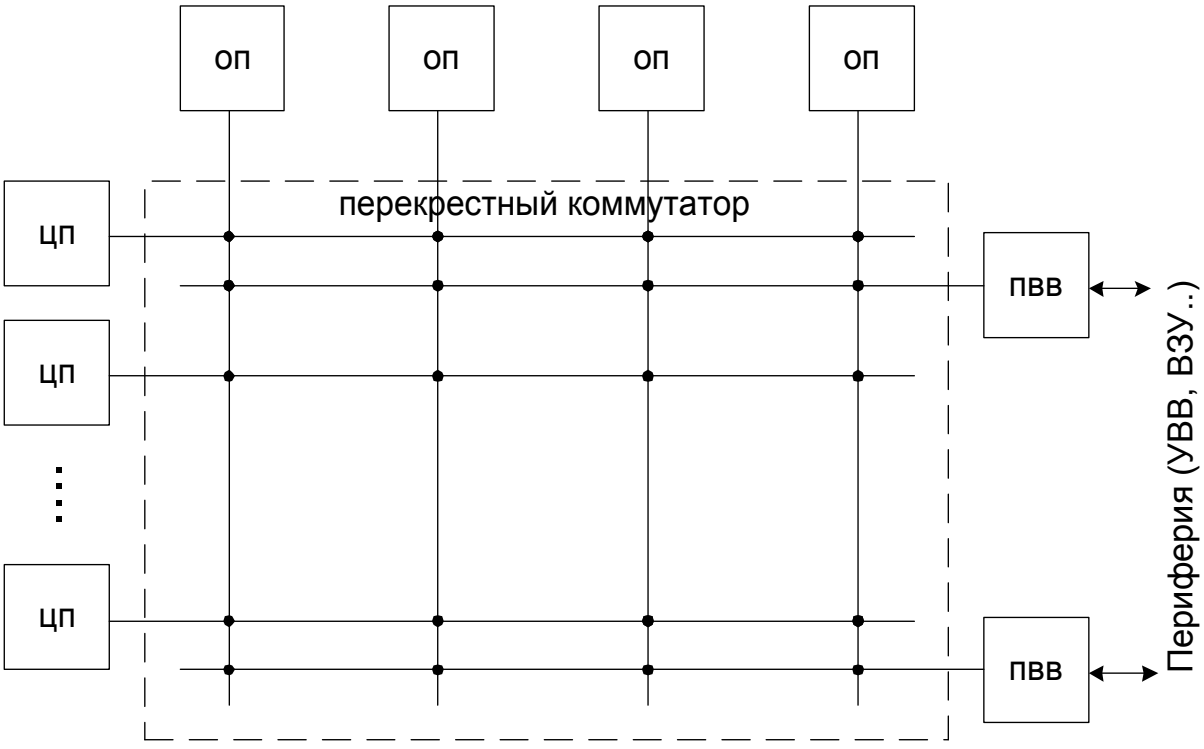


Рис.2.5. Многопроцессорная система с коммуникационной средой на основе перекрёстного коммутатора.

У перекрестного коммутатора в противоположность общей шине имеется больше достоинств, но есть и существенные недостатки, к которым относятся сложность внутренних связей, ведущая к усложнению аппаратуры

и плохая масштабируемость, ограниченная числом входов и выходов коммутатора.

Достоинствами ПК являются: возможность установления нескольких одновременных путей передачи информации; обеспечение большей производительности и надежности многопроцессорной системы.

Более экономичной схемой пространственной коммутации по сравнению с ПК является схема, использующая многовходовую память (рис. 2.6).

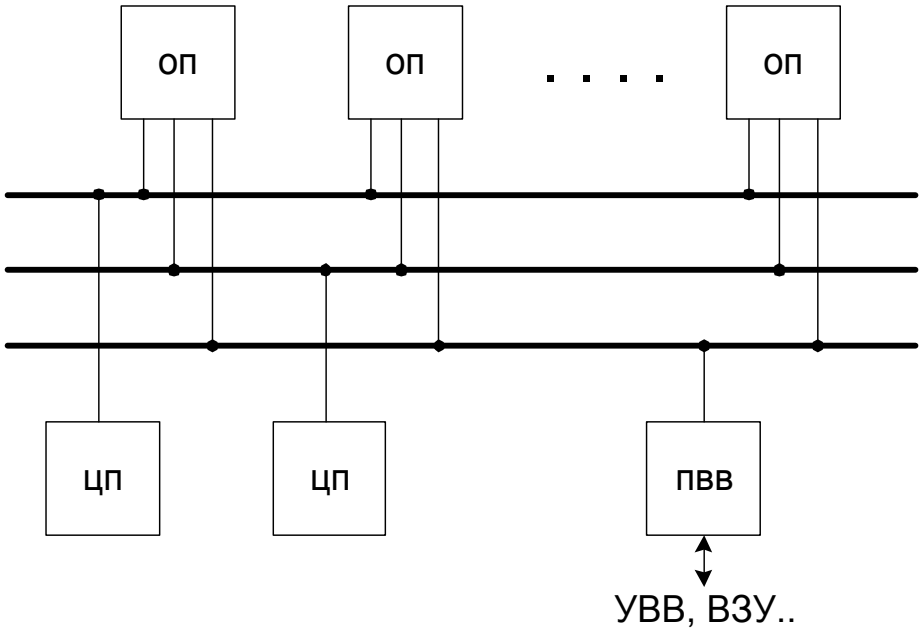


Рис.2.6. Многопроцессорная система с многошинной коммуникационной средой.

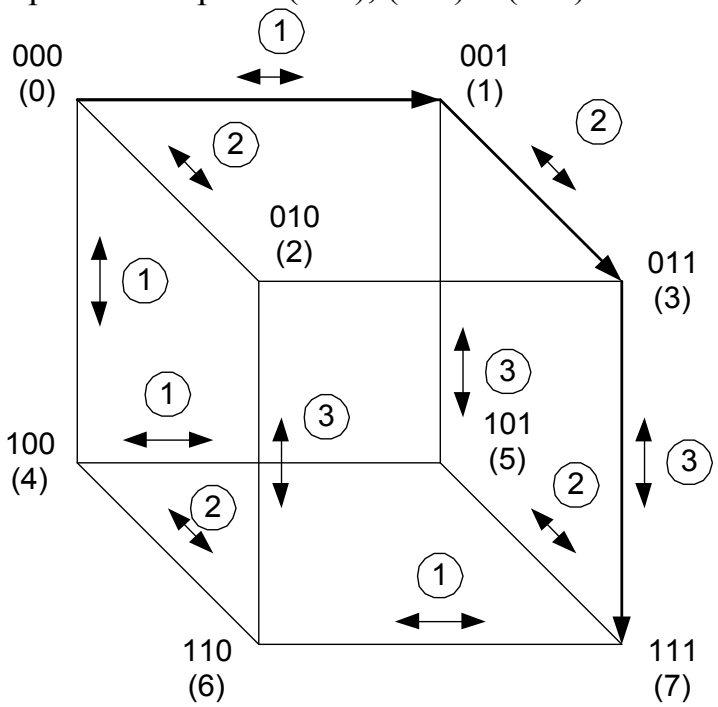
Здесь имеет место меньшее количество точек, в которых нужно разрешать конфликты. Максимально возможная конфигурация системы (количество подключаемых процессоров) ограничивается числом входов блоков памяти.

2.3. Организация коммуникационной среды в системах с распределённой памятью.

К настоящему времени разработано множество схем соединения, осуществляющих связь между произвольными процессорами с помощью коммуникационной среды из многоступенчатых переключателей, размещенных в log2 *n* ступенях. В качестве примеров этих схем можно назвать сеть Омега и *n*-кубическую (гиперкубическую) сеть.

Принцип действия п-кубической сети отражен на рис. 2.7, *a*, а пример его структурной реализации на многоступенчатом переключателе показан на рис. 2.7,*b*. На рисунке представлена система соединения восьми процессоров.

Как видно из рис. 2.7, *а*, всем процессорам присваиваются двоичные номера *(Xm-1…X1Xo)*. Процессор с номером *(Xm-1 ... X1Xo)* соединяется с *m* процессорами, для номеров которых расстояние Хемминга равняется 1. В примере на рис. 2.7, *а* процессор (000) может быть соединен с каждым из процессоров с номерами (001), (010) и (100), а процессор (010) - с любым из процессоров с номерами (011), (000) и (110).



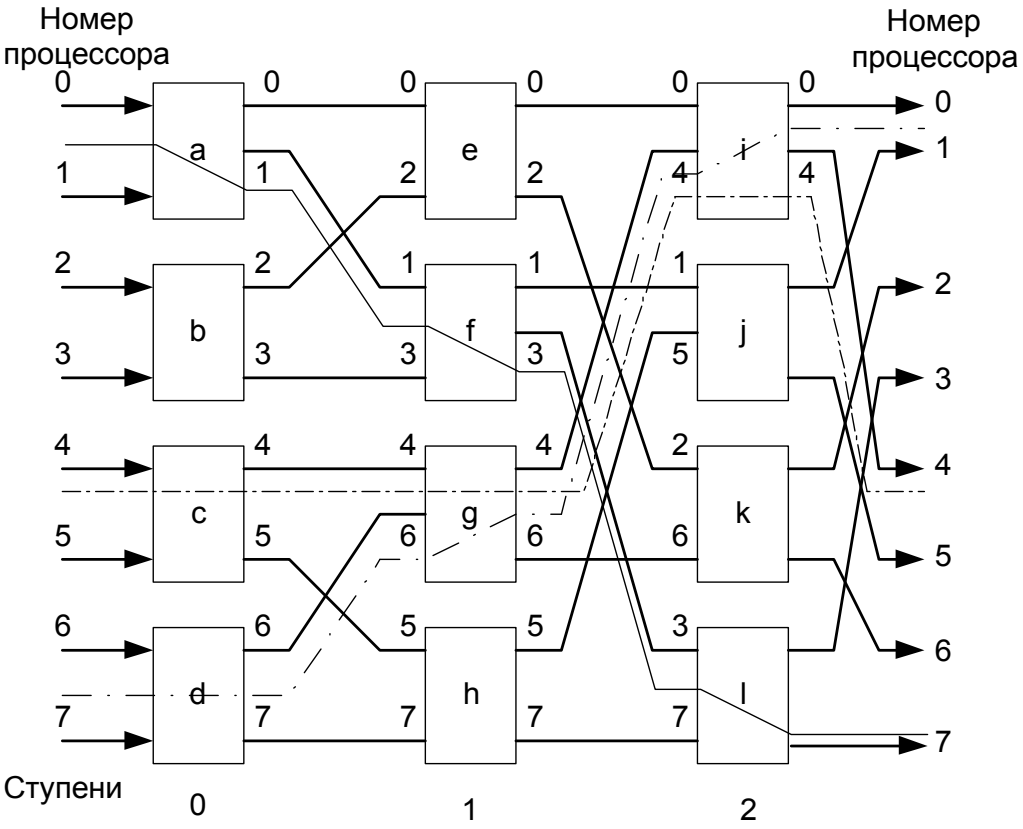


Рис.2.7. Коммуникационная среда на основе гиперкубической сети.

Рассмотрим теперь передачу данных от исходного процессора *(Xm-1... X1Xo)* в процессор-адресат *(Ym-1 ... Y1Y0)*. Сначала сравниваются биты Xo и Y0 младших разрядов номеров процессоров. При *Xo = Y0* данные из процессора *(Xm-1 ... X1Xo*), передаются в процессор (*Xm-1 ... X1Xo)* (на рис. 2.7,*a* это обозначено знаком 1 в кружке). При *Xo = Yo* данные не передаются. Далее, в номерах процессоров(*Xm-1 ... X1Xo)* и *(Ym-1…Y1Y0)* сравниваются биты *X1* и *Y1* и осуществляется аналогичная процедура (на рис. 2.7,*a* обозначена индексом 2 в кружке). Этот процесс повторяется со всеми парами битов до самого старшего разряда; в результате *m*-кратного повторения описанной процедуры данные поступают в процессор *(Ym-1 ...Y1Y0).* Путь передачи данных от процессора (000) в процессор (111) на рис. 2.7,*a* показан утолщенной линией. Рассмотрим реализацию данного способа с использованием многоступенчатого переключателя с *m* ступенями (log2 *n* ступеней).

Как видно из рис. 2.7,*b* , на каждой ступени размещается *n*/2 переключателей. На каждый из переключателей ступени 0 поступают данные от пары процессоров с двоичными номерами, различающимися значениями младших разрядов (например, (000) и (001), (010) и (Oil) и т. д.). В общем случае на каждый из переключателей ступени подаются данные от двух процессоров с номерами, различающимися значением *i*-го бита (например, на ступени 1 это (000) и (010), (001) и (Oil) и т.д. Для переключателей ступени *i* сравниваются бит *Xi* номера процессора-источника и бит *Yi* номера процессора-получателя; при *Xi=Yi* переключатель устанавливается в режим прямой связи, а при *Xi* не равным *Yi* - в режим диагональной связи. Таким образом, реализуется отмеченный ранее принцип *n*-кубической коммутации («прямая связь», когда данные не передаются, и «диагональная связь», когда данные передаются). Например, при передаче данных из процессора (000) в процессор (111) управление переключателями производится в соответствии с пунктирной линией передачи на рис. 2.7,*b*. Поскольку для всех переключателей управление их режимов производится на основании сравнения *Xi* и *Yi,* необходимо снабжать данные заголовком, представляющим собой поразрядную сумму *X(Xm-1 ... X1Xo)* и *Y{Ym-1...Y1Yo)* по модулю 2. Со стороны процессора - источника вычисляется *Zi = Xi + Yi,* где (+ - исключающее ИЛИ), и каждый переключатель при *Zi* = 0 устанавливается в режим прямой связи, а при *Zi* = 1 - в режим диагональной связи. Длина заголовка данных не превышает *m* бит.

При такой многоступенчатой коммутации возможна связь между любыми процессорами. Однако при одновременной организации связи между более чем двумя процессорами, как показано на рис. 2.7,*b* штрихпунктирной линией, возможны конфликты переключателей (блокирование). На рис. 2.7,*b* показана ситуация, когда одновременно передаются данные от процессора 4 к процессору 0 и от процессора 6 к процессору 4. При возникновении конфликтов данные запоминаются в буфере переключателя и ожидают своей очереди на передачу.

2.4. Когерентность кэш- памяти в SMP- системах.

Требования, предъявляемые современными процессорами к полосе пропускания памяти можно существенно сократить путем применения больших многоуровневых кэшей. Тогда, если эти требования снижаются, то несколько процессоров смогут разделять доступ к одной и той же памяти. Начиная с 1980 года эта идея, подкрепленная широким распространением микропроцессоров, стимулировала многих разработчиков на создание небольших мультипроцессоров, в которых несколько процессоров разделяют одну физическую память, соединенную с ними с помощью разделяемой шины. Из-за малого размера процессоров и заметного сокращения требуемой полосы пропускания шины, достигнутого за счет возможности реализации достаточно большой кэш-памяти, такие машины стали исключительно эффективными по стоимости. В первых разработках подобного рода машин удавалось разместить весь процессор и кэш на одной плате, которая затем вставлялась в заднюю панель, с помощью которой реализовывалась шинная архитектура. Современные конструкции позволяют разместить до четырех процессоров на одной плате.

В такой машине кэши могут содержать как разделяемые, так и частные данные. Частные данные - это данные, которые используются одним процессором, в то время как разделяемые данные используются многими процессорами, по существу обеспечивая обмен между ними. Когда кэшируется элемент частных данных, их значение переносится в кэш для сокращения среднего времени доступа, а также требуемой полосы пропускания. Поскольку никакой другой процессор не использует эти данные, этот процесс идентичен процессу для однопроцессорной машины с кэш-памятью. Если кэшируются разделяемые данные, то разделяемое значение реплицируется и может содержаться в нескольких кэшах. Кроме сокращения задержки доступа и требуемой полосы пропускания такая репликация данных способствует также общему сокращению количества обменов. Однако кэширование разделяемых данных вызывает новую проблему: когерентность кэш-памяти.

Проблема, о которой идет речь, возникает из-за того, что значение элемента данных в памяти, хранящееся в двух разных процессорах, доступно этим процессорам только через их индивидуальные кэши. На рис. 2.8 показан простой пример, иллюстрирующий эту проблему.

Проблема когерентности памяти для мультипроцессоров и устройств ввода/вывода имеет много аспектов. Обычно в малых мультипроцессорах используется аппаратный механизм, называемый протоколом, позволяющий решить эту проблему. Такие протоколы называются протоколами когерентности кэш-памяти. Существуют два класса таких протоколов:

1. Протоколы на основе справочника (directory based). Информация о состоянии блока физической памяти содержится только в одном месте, называемом справочником (физически справочник может быть распределен по узлам системы). Этот подход будет рассмотрен в разд. 10.3.

2. Протоколы наблюдения (snooping). Каждый кэш, который содержит копию данных некоторого блока физической памяти, имеет также соответствующую копию служебной информации о его состоянии. Централизованная система записей отсутствует. Обычно кэши расположены на общей (разделяемой) шине и контроллеры всех кэшей наблюдают за шиной (просматривают ее) для определения того, не содержат ли они копию соответствующего блока.

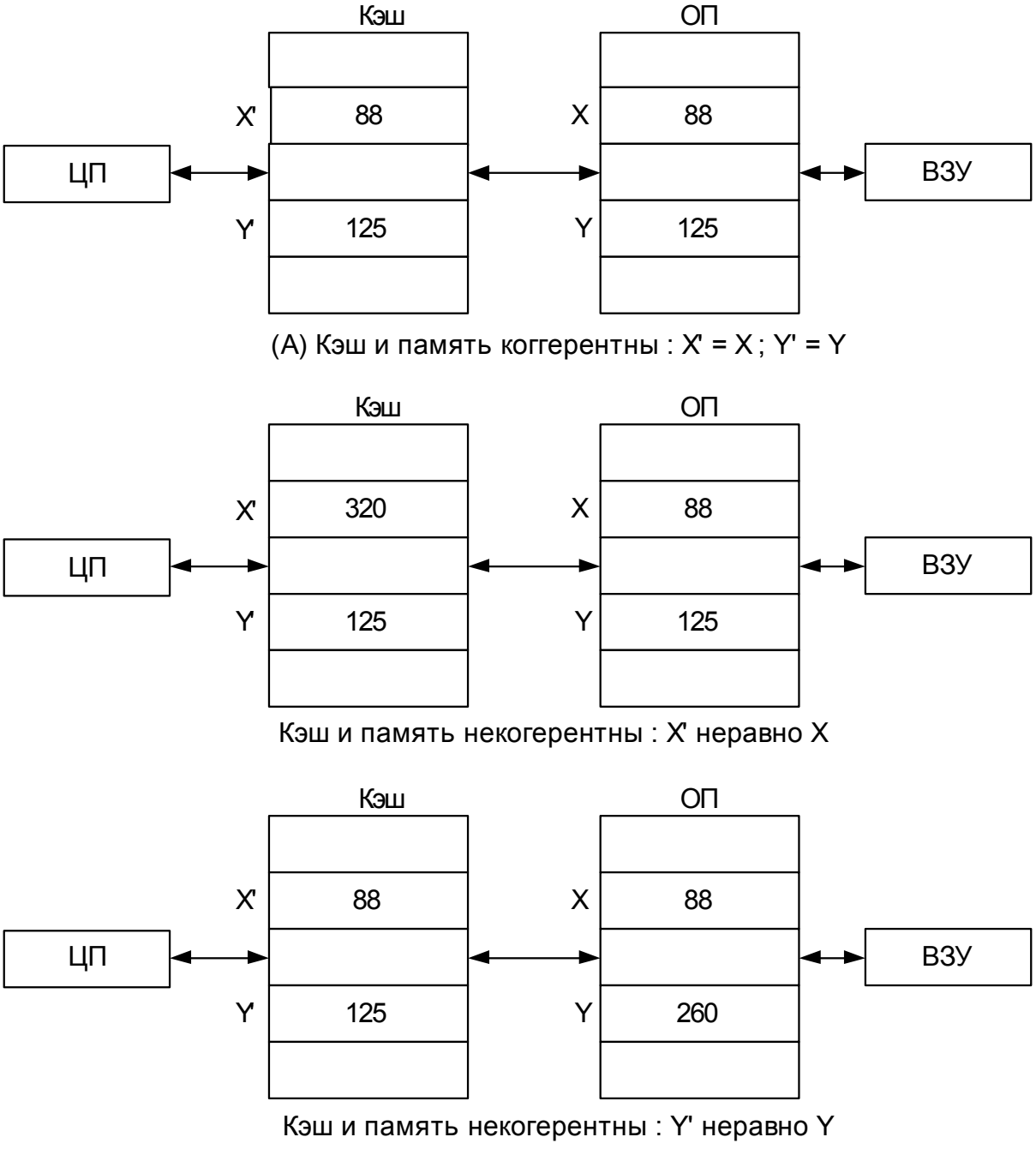


Рис. 2.8. Иллюстрация проблемы когерентности кэш-памяти

а) Когерентное состояние кэша и основной памяти.

б) Предполагается использование кэш-памяти с обратной записью, когда ЦП записывает значение 320 в ячейку X. В результате X’ содержит новое значение, а в основной памяти осталось старое значение 88. При попытке вывода X из памяти будет получено старое значение.

в) Внешняя память вводит в ячейку памяти Y новое значение 260, а в кэш-памяти осталось старое значение Y.

В мультипроцессорных системах, использующих процессоры с кэш-памятью, подсоединенные к централизованной общей памяти, протоколы наблюдения приобрели популярность, поскольку для опроса состояния кэшей они могут использовать заранее существующее физическое соединение - шину памяти.

Неформально, проблема когерентности памяти состоит в необходимости гарантировать, что любое считывание элемента данных возвращает последнее по времени записанное в него значение. Это определение не совсем корректно, поскольку невозможно требовать, чтобы операция считывания мгновенно видела значение, записанное в этот элемент данных некоторым другим процессором. Если, например, операция записи на одном процессоре предшествует операции чтения той же ячейки на другом процессоре в пределах очень короткого интервала времени, то невозможно гарантировать, что чтение вернет записанное значение данных, поскольку в этот момент времени записываемые данные могут даже не покинуть процессор. Вопрос о том, когда точно записываемое значение должно быть доступно процессору, выполняющему чтение, определяется выбранной моделью согласованного (непротиворечивого) состояния памяти и связан с реализацией синхронизации параллельных вычислений. Поэтому с целью упрощения предположим, что мы требуем только, чтобы записанное операцией записи значение было доступно операции чтения, возникшей немного позже записи и что операции записи данного процессора всегда видны в порядке их выполнения.

С этим простым определением согласованного состояния памяти мы можем гарантировать когерентность путем обеспечения двух свойств:

1. Операция чтения ячейки памяти одним процессором, которая следует за операцией записи в ту же ячейку памяти другим процессором получит записанное значение, если операции чтения и записи достаточно отделены друг от друга по времени.

2. Операции записи в одну и ту же ячейку памяти выполняются строго последовательно (иногда говорят, что они сериализованы): это означает, что две подряд идущие операции записи в одну и ту же ячейку памяти будут наблюдаться другими процессорами именно в том порядке, в

котором они появляются в программе процессора, выполняющего эти операции записи.

Первое свойство очевидно связано с определением когерентного (согласованного) состояния памяти: если бы процессор всегда бы считывал только старое значение данных, мы сказали бы, что память некогерентна.

Необходимость строго последовательного выполнения операций записи является более тонким, но также очень важным свойством. Представим себе, что строго последовательное выполнение операций записи не соблюдается. Тогда процессор P1 может записать данные в ячейку, а затем в эту ячейку выполнит запись процессор P2. Строго последовательное выполнение операций записи гарантирует два важных следствия для этой последовательности операций записи. Во-первых, оно гарантирует, что каждый процессор в машине в некоторый момент времени будет наблюдать запись, выполняемую процессором P2. Если последовательность операций записи не соблюдается, то может возникнуть ситуация, когда какой-нибудь процессор будет наблюдать сначала операцию записи процессора P2, а затем операцию записи процессора P1, и будет хранить это записанное P1 значение неограниченно долго. Более тонкая проблема возникает с поддержанием разумной модели порядка выполнения программ и когерентности памяти для пользователя: представьте, что третий процессор постоянно читает ту же самую ячейку памяти, в которую записывают процессоры P1 и P2; он должен наблюдать сначала значение, записанное P1, а затем значение, записанное P2. Возможно он никогда не сможет увидеть значения, записанного P1, поскольку запись от P2 возникла раньше чтения. Если он даже видит значение, записанное P1, он должен видеть значение, записанное P2, при последующем чтении. Подобным образом любой другой процессор, который может наблюдать за значениями, записываемыми как P1, так и P2, должен наблюдать идентичное поведение. Простейший способ добиться таких свойств заключается в строгом соблюдении порядка операций записи, чтобы все записи в одну и ту же ячейку могли наблюдаться в том же самом порядке. Это свойство называется последовательным выполнением (сериализацией) операций записи (write serialization). Вопрос о том, когда процессор должен увидеть значение, записанное другим процессором достаточно сложен и имеет заметное воздействие на производительность, особенно в больших машинах.

Альтернативные протоколы

Имеются две методики поддержания описанной выше когерентности. Один из методов заключается в том, чтобы гарантировать, что процессор должен получить исключительные права доступа к элементу данных перед выполнением записи в этот элемент данных. Этот тип протоколов называется протоколом записи с аннулированием (write ivalidate protocol), поскольку при выполнении записи он аннулирует другие копии. Это наиболее часто используемый протокол как в схемах на основе справочников, так и в схемах наблюдения. Исключительное право доступа гарантирует, что во время выполнения записи не существует никаких других копий элемента данных, в которые можно писать или из которых можно читать: все другие кэшированные копии элемента данных аннулированы. Чтобы увидеть, как такой протокол обеспечивает когерентность, рассмотрим операцию записи, вслед за которой следует операция чтения другим процессором. Поскольку запись требует исключительного права доступа, любая копия, поддерживаемая читающим процессором должна быть аннулирована (в соответствии с названием протокола). Таким образом, когда возникает операция чтения, произойдет промах кэш-памяти, который вынуждает выполнить выборку новой копии данных. Для выполнения операции записи мы можем потребовать, чтобы процессор имел достоверную (valid) копию данных в своей кэш-памяти прежде, чем выполнять в нее запись. Таким образом, если оба процессора попытаются записать в один и тот же элемент данных одновременно, один из них выиграет состязание у второго (мы вскоре увидим, как принять решение, кто из них выиграет) и вызывает аннулирование его копии. Другой процессор для завершения своей операции записи должен сначала получить новую копию данных, которая теперь уже должна содержать обновленное значение.

Альтернативой протоколу записи с аннулированием является обновление всех копий элемента данных в случае записи в этот элемент данных. Этот тип протокола называется протоколом записи с обновлением (write update protocol) или протоколом записи с трансляцией (write broadcast protocol). Обычно в этом протоколе для снижения требований к полосе пропускания полезно отслеживать, является ли слово в кэш-памяти разделяемым объектом, или нет, а именно, содержится ли оно в других кэшах. Если нет, то нет никакой необходимости обновлять другой кэш или транслировать в него обновленные данные.

Разница в производительности между протоколами записи с обновлением и с аннулированием определяется тремя характеристиками:

1. Несколько последовательных операций записи в одно и то же слово, не перемежающихся операциями чтения, требуют нескольких операций трансляции при использовании протокола записи с обновлением, но только одной начальной операции аннулирования при использовании протокола записи с аннулированием.

2. При наличии многословных блоков в кэш-памяти каждое слово, записываемое в блок кэша, требует трансляции при использовании протокола записи с обновлением, в то время как только первая запись в любое слово блока нуждается в генерации операции аннулирования при использовании протокола записи с аннулированием. Протокол записи с аннулированием работает на уровне блоков кэш-памяти, в то время как протокол записи с обновлением должен работать на уровне отдельных слов (или байтов, если выполняется запись байта).

3. Задержка между записью слова в одном процессоре и чтением записанного значения другим процессором обычно меньше при использовании схемы записи с обновлением, поскольку записанные данные немедленно транслируются в процессор, выполняющий чтение (предполагается, что этот процессор имеет копию данных). Для сравнения, при использовании протокола записи с аннулированием в процессоре, выполняющим чтение, сначала произойдет аннулирование его копии, затем будет производиться чтение данных и его приостановка до тех пор, пока обновленная копия блока не станет доступной и не вернется в процессор.

Эти две схемы во многом похожи на схемы работы кэш-памяти со сквозной записью и с записью с обратным копированием. Также как и схема задержанной записи с обратным копированием требует меньшей полосы пропускания памяти, так как она использует преимущества операций над целым блоком, протокол записи с аннулированием обычно требует менее тяжелого трафика, чем протокол записи с обновлением, поскольку несколько записей в один и тот же блок кэш-памяти не требуют трансляции каждой записи. При сквозной записи память обновляется почти мгновенно после записи (возможно с некоторой задержкой в буфере записи). Подобным образом при использовании протокола записи с обновлением другие копии обновляются так быстро, насколько это возможно. Наиболее важное отличие в производительности протоколов записи с аннулированием и с обновлением связано с характеристиками прикладных программ и с выбором размера блока.

Основы реализации

Ключевым моментом реализации в многопроцессорных системах с небольшим числом процессоров как схемы записи с аннулированием, так и схемы записи с обновлением данных, является использование для выполнения этих операций механизма шины. Для выполнения операции обновления или аннулирования процессор просто захватывает шину и

транслирует по ней адрес, по которому должно производиться обновление или аннулирование данных. Все процессоры непрерывно наблюдают за шиной, контролируя появляющиеся на ней адреса. Процессоры проверяют не находится ли в их кэш-памяти адрес, появившийся на шине. Если это так, то соответствующие данные в кэше либо аннулируются, либо обновляются в зависимости от используемого протокола. Последовательный порядок обращений, присущий шине, обеспечивает также строго последовательное выполнение операций записи, поскольку когда два процессора конкурируют за выполнение записи в одну и ту же ячейку, один из них должен получить доступ к шине раньше другого. Один процессор, получив доступ к шине, вызовет необходимость обновления или аннулирования копий в других процессорах. В любом случае, все записи будут выполняться строго последовательно. Один из выводов, который следует сделать из анализа этой схемы заключается в том, что запись в разделяемый элемент данных не может закончиться до тех пор, пока она не захватит доступ к шине.

В дополнение к аннулированию или обновлению соответствующих копий блока кэш-памяти, в который производилась запись, мы должны также разместить элемент данных, если при записи происходит промах кэш-памяти. В кэш-памяти со сквозной записью последнее значение элемента данных найти легко, поскольку все записываемые данные всегда посылаются также и в память, из которой последнее записанное значение элемента данных может быть выбрано (наличие буферов записи может привести к некоторому усложнению).

Однако для кэш-памяти с обратным копированием задача нахождения последнего значения элемента данных сложнее, поскольку это значение скорее всего находится в кэше, а не в памяти. В этом случае используется та же самая схема наблюдения, что и при записи: каждый процессор наблюдает и контролирует адреса, помещаемые на шину. Если процессор обнаруживает, что он имеет модифицированную ("грязную") копию блока кэш-памяти, то именно он должен обеспечить пересылку этого блока в ответ на запрос чтения и вызвать отмену обращения к основной памяти. Поскольку кэши с обратным копированием предъявляют меньшие требования к полосе пропускания памяти, они намного предпочтительнее в мультипроцессорах, несмотря на некоторое увеличение сложности. Поэтому далее мы рассмотрим вопросы реализации кэш-памяти с обратным копированием.

Для реализации процесса наблюдения могут быть использованы обычные теги кэша. Более того, упоминавшийся ранее бит достоверности (valid bit), позволяет легко реализовать аннулирование. Промахи операций чтения, вызванные либо аннулированием, либо каким-нибудь другим событием, также не сложны для понимания, поскольку они просто основаны на возможности наблюдения. Для операций записи мы хотели бы также знать, имеются ли другие кэшированные копии блока, поскольку в случае

отсутствия таких копий, запись можно не посылать на шину, что сокращает время на выполнение записи, а также требуемую полосу пропускания.

Чтобы отследить, является ли блок разделяемым, мы можем ввести дополнительный бит состояния (shared), связанный с каждым блоком, точно также как это делалось для битов достоверности (valid) и модификации (modified или dirty) блока. Добавив бит состояния, определяющий является ли блок разделяемым, мы можем решить вопрос о том, должна ли запись генерировать операцию аннулирования в протоколе с аннулированием, или операцию трансляции при использовании протокола с обновлением. Если происходит запись в блок, находящийся в состоянии "разделяемый" при использовании протокола записи с аннулированием, кэш формирует на шине операцию аннулирования и помечает блок как частный (private). Никаких последующих операций аннулирования этого блока данный процессор посылать больше не будет. Процессор с исключительной (exclusive) копией блока кэш-памяти обычно называется "владельцем" (owner) блока кэш-памяти.

При использовании протокола записи с обновлением, если блок находится в состоянии "разделяемый", то каждая запись в этот блок должна транслироваться. В случае протокола с аннулированием, когда посылается операция аннулирования, состояние блока меняется с "разделяемый" на "неразделяемый" (или "частный"). Позже, если другой процессор запросит этот блок, состояние снова должно измениться на "разделяемый". Поскольку наш наблюдающий кэш видит также все промахи, он знает, когда этот блок кэша запрашивается другим процессором, и его состояние должно стать "разделяемый".

Поскольку любая транзакция на шине контролирует адресные теги кэша, потенциально это может приводить к конфликтам с обращениями к кэшу со стороны процессора. Число таких потенциальных конфликтов можно снизить применением одного из двух методов: дублированием тегов, или использованием многоуровневых кэшей с "охватом" (inclusion), в которых уровни, находящиеся ближе к процессору являются поднабором уровней, находящихся дальше от него. Если теги дублируются, то обращения процессора и наблюдение за шиной могут выполняться параллельно. Конечно, если при обращении процессора происходит промах, он должен будет выполнять арбитраж с механизмом наблюдения для обновления обоих наборов тегов. Точно также, если механизм наблюдения за шиной находит совпадающий тег, ему будет нужно проводить арбитраж и обращаться к обоим наборам тегов кэша (для выполнения аннулирования или обновления бита "разделяемый"), возможно также и к массиву данных в кэше, для нахождения копии блока. Таким образом, при использовании схемы дублирования тегов процессор должен приостановиться только в том случае, если он выполняет обращение к кэшу в тот же самый момент времени, когда

механизм наблюдения обнаружил копию в кэше. Более того, активность механизма наблюдения задерживается только когда кэш имеет дело с промахом.

Наименование	Тип протокола	Стратегия записи в память	Уникальные свойства Применение
Одиночная запись	Запись с аннулированием	Обратное копирование при первой записи	Первый описанный в литературе протокол наблюдения -
Synapse N+1	Запись с аннулированием	Обратное копирование	Точное состояние, где "владельцем является память" Машины Synapse Первые машины с когерентной кэш-памятью
Berkely	Запись с аннулированием	Обратное копирование	Состояние "разделяемый" Машина SPUR университета Berkely
Illinois	Запись с аннулированием	Обратное копирование	Состояние "приватный"; может передавать данные из любого кэша Серии Power и Challenge компании Silicon Graphics

"Firefly"	Запись с трансляцией	Обратное копирование для "приватных" блоков и сквозная запись для "разделяемых"	Обновление памяти во время трансляции SPARC center 2000

Рис. 2.9. Примеры протоколов наблюдения

Если процессор использует многоуровневый кэш со свойствами охвата, тогда каждая строка в основном кэше имеется и во вторичном кэше. Таким образом, активность по наблюдению может быть связана с кэшем второго уровня, в то время как большинство активностей процессора могут быть связаны с первичным кэшем. Если механизм наблюдения получает попадание во вторичный кэш, тогда он должен выполнять арбитраж за первичный кэш, чтобы обновить состояние и возможно найти данные, что обычно будет приводить к приостановке процессора. Такое решение было принято во многих современных системах, поскольку многоуровневый кэш позволяет существенно снизить требований к полосе пропускания. Иногда может быть даже полезно дублировать теги во вторичном кэше, чтобы еще больше сократить количество конфликтов между активностями процессора и механизма наблюдения.

В реальных системах существует много вариаций схем когерентности кэша, в зависимости от того используется ли схема на основе аннулирования или обновления, построена ли кэш-память на принципах сквозной или обратной записи, когда происходит обновление, а также имеет ли место состояние "владения" и как оно реализуется. На рис. 2.9 представлены несколько протоколов с наблюдением и некоторые машины, которые используют эти протоколы.

2.5. Когерентность кэш- памяти в MPP-системах.

Существуют два различных способа построения крупномасштабных систем с распределенной памятью. Простейший способ заключается в том, чтобы исключить аппаратные механизмы, обеспечивающие когерентность кэш-памяти, и сосредоточить внимание на создании масштабируемой системы памяти. Несколько компаний разработали такого типа машины.

Наиболее известным примером такой системы является компьютер T3D компании Cray Research. В этих машинах память распределяется между узлами (процессорными элементами) и все узлы соединяются между собой посредством того или иного типа сети. Доступ к памяти может быть локальным или удаленным. Специальные контроллеры, размещаемые в узлах сети, могут на основе анализа адреса обращения принять решение о том, находятся ли требуемые данные в локальной памяти данного узла, или размещаются в памяти удаленного узла. В последнем случае контроллеру удаленной памяти посылается сообщение для обращения к требуемым данным.

Чтобы обойти проблемы когерентности, разделяемые (общие) данные не кэшируются. Конечно, с помощью программного обеспечения можно реализовать некоторую схему кэширования разделяемых данных путем их копирования из общего адресного пространства в локальную память конкретного узла. В этом случае когерентностью памяти также будет управлять программное обеспечение. Преимуществом такого подхода является практически минимально необходимая поддержка со стороны аппаратуры, хотя наличие, например, таких возможностей как блочное (групповое) копирование данных было бы весьма полезным. Недостатком такой организации является то, что механизмы программной поддержки когерентности подобного рода кэш-памяти компилятором весьма ограничены. Существующая в настоящее время методика в основном подходит для программ с хорошо структурированным параллелизмом на уровне программного цикла.

Машины с архитектурой, подобной Cray T3D, называют процессорами (машинами) с массовым параллелизмом (MPP Massively Parallel Processor). К машинам с массовым параллелизмом предъявляются взаимно исключающие требования. Чем больше объем устройства, тем большее число процессоров можно расположить в нем, тем длиннее каналы передачи управления и данных, а значит и меньше тактовая частота. Происшедшее возрастание нормы массивности для больших машин до 512 и даже 64К процессоров обусловлено не ростом размеров машины, а повышением степени интеграции схем, позволившей за последние годы резко повысить плотность размещения элементов в устройствах. Топология сети обмена между процессорами в такого рода системах может быть различной. На рис. 2.10 приведены характеристики сети обмена для некоторых коммерческих MPP.

В частности, чтобы предотвратить появление узкого горла в системе, связанного с единым справочником, можно распределить части этого справочника вместе с устройствами распределенной локальной памяти. Таким образом можно добиться того, что обращения к разным справочникам (частям единого справочника) могут выполняться параллельно, точно также как обращения к локальной памяти в распределенной памяти могут

выполняться параллельно, существенно увеличивая общую полосу пропускания памяти. В распределенном справочнике сохраняется главное свойство подобных схем, заключающееся в том, что состояние любого разделяемого блока данных всегда находится во вполне определенном известном месте. На рис.2.11 показан общий вид подобного рода системы с распределенной памятью.

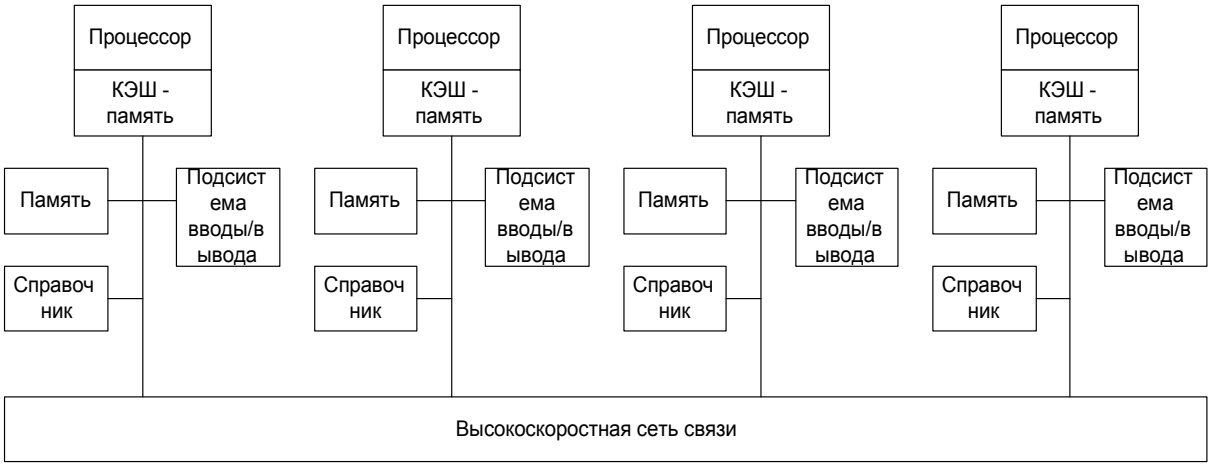


Рис. 2.11. Архитектура системы с распределенной памятью и распределенными по узлам справочниками.

2.6. Организация прерываний в мультипроцессорных системах

Рассмотрим реализацию прерываний в наиболее простых симметричных многопроцессорных системах, в которых используется несколько процессоров, объединенных общей шиной. Каждый процессор выполняет свою задачу, задаваемую операционной системой (ОС). При этом процессоры совместно используют общие ресурсы системы (память, внешние устройства), обращение к которым регулируется ОС. В каждый момент времени один из процессоров является ведущим (master) - только он имеет доступ к системной шине. Другие процессоры в случае необходимости обращения к шине выдают соответствующий запрос. Эти запросы анализируются специальным устройством - арбитром шины, который работает под управлением ОС. В соответствии с определенным алгоритмом арбитр предоставляет доступ к шине одному из запросивших процессоров, который становится таким образом ведущим. Поддержку функционирования таких мультипроцессорных систем обеспечивает ряд современных ОС (Windows NT, Novell NetWare и другие). Чаще всего симметричные мультипроцессорные системы содержат два или четыре процессора.

Характерным примером является система прерываний, реализованная в процессорах фирмы Intel. Так, например, процессоры семейства P6(PentiumII, PentiumIII, Celeron и др.) имеют ряд средств для поддержки работы

мультипроцессорных систем, обеспечивая для процессоров взаимный доступ к содержимому внутренней кэш-памяти данных (снупинг), возможность блокировки доступа к шине при выполнении ряда процедур и другие возможности. Различные модели этого семейства позволяют организовать эффективную работу двух- или четырехпроцессорных систем.

Одной из наиболее серьезных проблем при реализации мультипроцессорных систем является организация обслуживания внешних (аппаратных) прерываний. Классическая организация обслуживания с помощью контроллера прерываний, подающего сигнал запроса INTR и формирующего код команды INT n, с реализацией процессором цикла подтверждения прерывания ориентирована на использование в однопроцессорной системе. Для обеспечения функционирования мультипроцессорных систем в процессоры семейства P6 введен программируемый контроллер прерываний с расширенными возможностями APIC (APIC – Advanced Programmable Interrupt Controller)

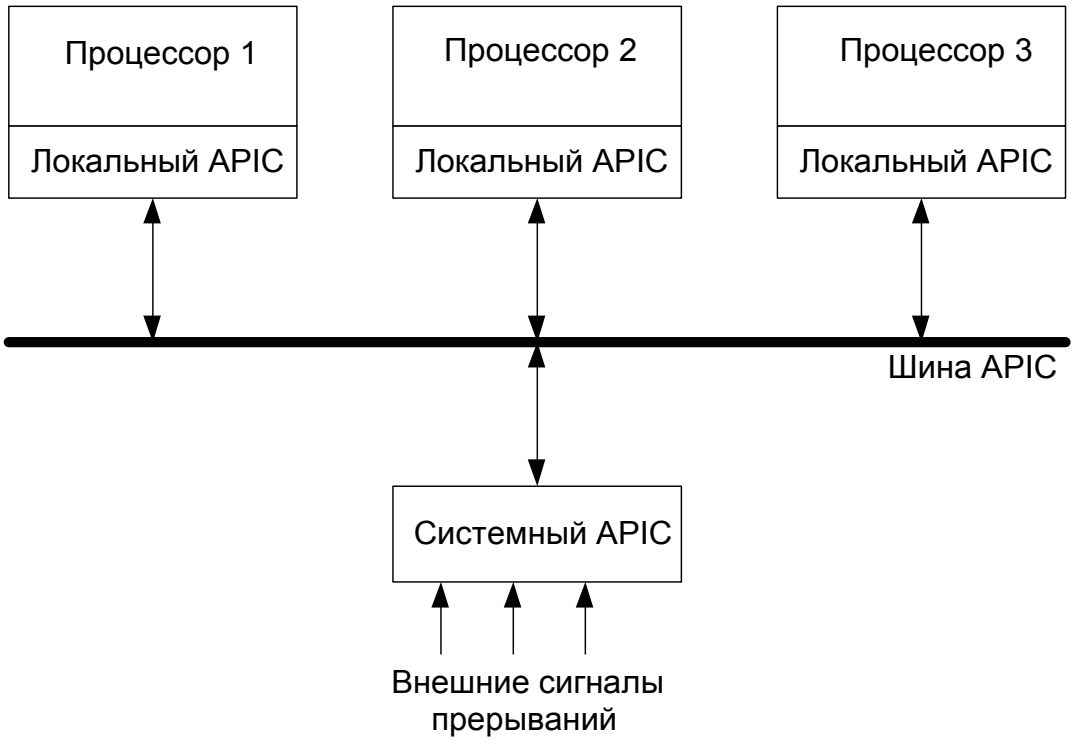


Рис.2.12. Схема прерываний в многопроцессорной системе.

Внутренние контроллеры прерываний связаны между собой по специальной APIC-шине (рис.2.12). Общие внешние запросы прерываний поступают на системный APIC - контроллер, который реализован в виде отдельной микросхемы, разработанной и поставляемой компанией Intel. Каждый из процессоров содержит локальный APIC, имеющий две входных лини LINT 0, LINT 1, на которые поступают локальные запросы прерывания, обслуживаемые только данным процессором. При работе в однопроцессорной системе APIC отключается, и выводы LINT 1-0 используются для подачи запросов немаскируемого NMI и маскируемого INTR прерываний.

Общие запросы прерывания поступают на системный APIC, который после их анализа выдает соответствующие послания на внутреннюю APIC - шину. Эта шина содержит три линии, на одну из которых (PICCLK) выдается синхросигнал, а две других (PICD 1-0) служат для последовательного обмена информацией в процессе организации обслуживания поступивших запросов. При этом для внешних устройств, формирующих запросы прерывания, мультипроцессорная система выглядит как один процессор, а процедура обслуживания запросов соответствует процедуре, выполняемой серийным контроллером прерываний Intel 8259A, который широко используется в современных системах.

2.7. Организация межпроцессорного обмена в системах с разделяемой памятью и общей шиной

Рассмотрим принципы организации обмена между вычислительными модулями на примере многопроцессорной системы с разделяемой памятью, использующей в качестве коммуникационной среды общую шину.

2.7.1. Структура системы с разделяемой памятью

Структура многопроцессорной системы с общей шиной и разделяемой (общей) памятью (рис.2.13) содержит n вычислительных модулей $ВМ_1,...,ВМ_n$, m модулей оперативной памяти $МОП_1,...,МОП_m$, работающих под управлением контроллера. В составе вычислительных модулей находится центральный процессор (ЦП) и шинный интерфейс, сопрягающий его с общей шиной.

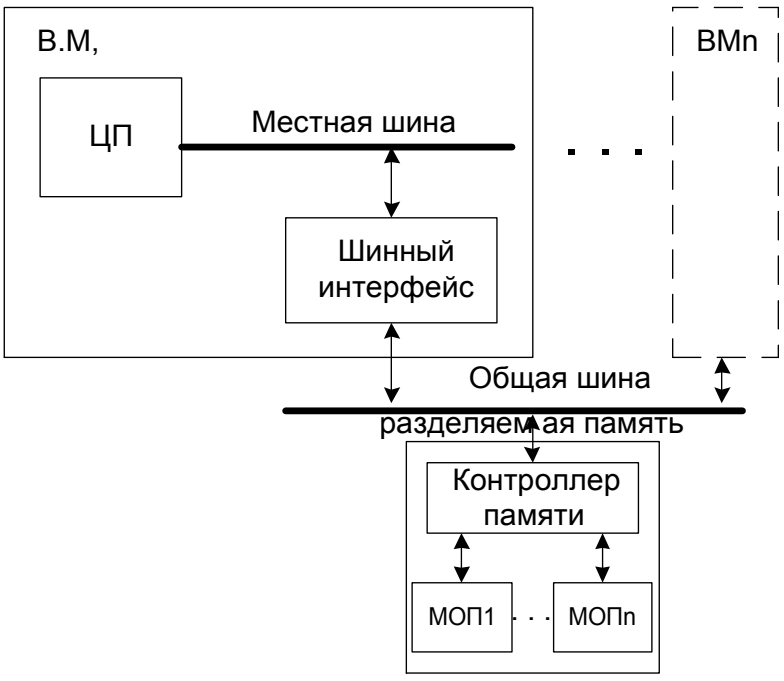


Рис.2.13. Структура системы с общей шиной и разделяемой памятью

Функции шинного интерфейса следующие:

- управление занятием/освобождением общей шины;
- управление обменом между вычислительным модулем и общей памятью;
- распознавание собственного адреса, выставленного на общую шину контроллером разделяемой памяти или другим устройством;
- организация внешних прерываний от запросов из системного контролера прерываний;
- буферизация передаваемых данных.

Каждому вычислительному модулю присваивают уникальный номер, который будем называть собственным адресом. Протоколом обмена предусматривается распознавание собственного адреса, когда сообщение передаётся извне в данный вычислительный модуль. Для этого шинный интерфейс постоянно отслеживает код присвоенного ему адреса. Как только он обнаружит собственный адрес, производится коммутация вычислительного модуля на общую шину и осуществляется приём данных с ОШ в приемный буфер интерфейса или в процессор.

Будем считать, что обмен процессор - память может производиться пословно или группами слов. Если применяется обмен без буферизации (в шинном интерфейсе и контроллере памяти отсутствуют буферы данных), то используется обычный машинный цикл (его ещё называют циклом шины) чтения или записи в память. По окончании цикла шины память освобождается и может быть передана в распоряжение другому вычислительному модулю.

Время обмена составит $t_B = \tau + t_{MC}$, где τ - время занятия общей шины; $t_{мс}$ - машинный цикл процессора, связанный с обращением к памяти. Время занятия τ зависит от способа управления общей шиной. Цикл обращения к памяти, в свою очередь, зависит от типа процессора и не зависит от быстродействия модуля памяти.

Из этого следует, что при способе обмена без буферизации сообщений в системе можно иметь только один модуль общей памяти, т.к. параллельная работа нескольких модулей невозможна. Это обстоятельство не позволяет достичь высокой степени параллельности выполнения работ из-за частых конфликтов, которые будут возникать между вычислительными модулями в борьбе за доступ к общей памяти. В связи с этим потенциальная производительность таких систем низкая.

Чтобы обеспечить параллельность работы модулей памяти, необходимо значительно увеличить пропускную способность общей шины и общей(разделяемой) памяти.

Первое достигается за счёт сокращения цикла шины, связанного с записью (считыванием) данных в память. Для этого передаваемые сообщения

буферизуются в быстрых регистрах шинного интерфейса. В режиме записи процессор инициирует обмен, передав в буферную память шинного интерфейса адрес ячейки памяти, управляющую информацию и данные. Контроллер шинного интерфейса самостоятельно, после получения доступа к общей шине, быстро передаёт всю информацию в буфер общей памяти. Для этих целей в контроллер разделяемой памяти так же как и в устройство шинного интерфейса вычислительного модуля включают быстродействующий буфер небольшого объёма. В этом случае цикл шины может быть значительно меньше цикла памяти. Следовательно, в течение цикла памяти возможна передача нескольких слов по общей шине. Поэтому число модулей памяти может быть увеличено во столько раз, во сколько цикл памяти больше цикла шины. Такой способ обмена называют с буферизацией передаваемых данных.

Для повышения пропускной способности разделяемой памяти применяют её расслоение на ряд независимых модулей МОП1-МОПn (см. рис.3.9) с использованием чередования адресов.

Время записи составит $t_W = \tau + t_{BF}$, где t_{BF} - время обращения к буферному регистру.

При чтении данных процессором из разделяемой памяти в режиме с буферизацией потребуется уже два, но таких же коротких, как и при записи, цикла шины. Вычислительный модуль, требующий данные из общей памяти, выставляет собственный адрес, адрес модуля памяти, адрес ячейки памяти внутри модуля и необходимые управляющие сигналы. Эта информация запоминается в быстродействующей буферной памяти шинного интерфейса. Контроллер шинного интерфейса самостоятельно организует их отправку адресату, т.е. контроллеру разделяемой памяти. Контроллер производит выборку операнда из адресуемого модуля памяти и в свою очередь формирует кадр обмена с вычислительным модулем-адресантом. В него включают адрес вычислительного модуля- адресанта(берут его из сообщения, которое было направлено в память) и прочитанные из памяти данные .

Время, необходимое для чтения данных из общей памяти, составит $t_R = 2(\tau + t_{BF})$. Среднее время обмена вычислительного модуля за время выполнения программы, ели в ней содержится h команд записи и g команд чтения, определится выражением

$$t_{обм} = \frac{ht_W + gt_R}{h+g}.$$

Подставив в (3.25) выражения для t_W и t_R, получим:

$$t_B = \tau + t_{BF} + \frac{g(\tau + t_{RF})}{h+g}.$$

При таком способе обмена совмещается работа процессора, памяти и интерфейсного оборудования, улучшаются возможности масштабирования и, как следствие, может быть увеличена общая производительность системы.

Пропускную способность общей шины может варьироваться, поскольку она зависит от ширины шины, которая определяется величиной, обратной числу циклов шины, необходимых для передачи одного слова (байта) данных.

Если обозначить ширину шины буквой *b*, то при *b*=1 число линий в шине равно числу разрядов в информационной части передаваемого сообщения. Для небольших многопроцессорных систем, как правило, *b<1*. В современных крупных многопроцессорных системах b>>1, где разрядность только системной шины данных составляет 128 или более разрядов.

От ширины шины зависят характеристики межмодульного интерфейса. Чем меньше ширина шины, тем меньше затраты на изготовление интерфейсного оборудования, однако ниже пропускная способность, и следовательно, потенциальная производительность и масштабируемость системы.

2.7.2 Структура системы с разделяемой распределённой памятью

Структура системы представлена на рис.2.14 и содержит в своём составе процессор, локальную память, подключённую на местную шину и шинный интерфейс

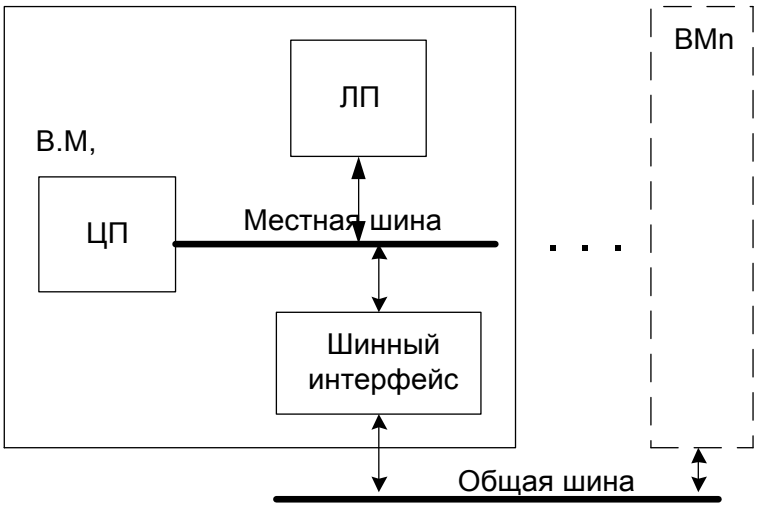


Рис.2.14. Структура системы с разделяемой распределённой памятью

Функции шинного интерфейса зависят от способа организации управления системой и могут включать:

- организацию доступа к памяти удаленного вычислительного модуля;
- управление занятием/освобождением общей шины;
- организацию внешних прерываний;

- управление локальной памятью при доступе к ней со стороны удалённого вычислительного модуля;
- распознавание собственного адреса, выставленного на общую шину удалённым вычислительным модулем;
- буферизацию передаваемых данных.

Коммутация адреса, данных и управляющей информации может производиться с буферизацией или без неё, в зависимости от принятого способа связи между вычислительными модулями. Коммутаторы шинного интерфейса должны обеспечивать режим высокого выходного сопротивления для отключения вычислительного модуля от общей шины на период местных обращений процессора в локальную память. Освобождённая ОШ может быть предоставлена для осуществления связи другим вычислительным модулям.

Обращения вычислительных модулей в удалённую память может производится либо прямым доступом, либо некоторым другим способом, например, по прерыванию. Для организации обращения в удалённую память вычислительный модуль-источник выставляет на общую шину адрес вычислительного модуля - приёмника, адрес ячейки локальной памяти удалённого вычислительного модуля, данные и управляющую информацию, т.е. формирует кадр обмена. Шинный интерфейс вычислительного модуля - приёмника, обнаружив собственный адрес, вырабатывает сигнал ЗАПРОС ПРЯМОГО ДОСТУПА или ЗАПРОС ПРЕРЫВАНИЯ, с помощью которого приостанавливается работа местного процессора на период времени обращения к его локальной памяти внешнего процессора. По сигналу ПОДТВЕРЖДЕНИЕ ДОСТУПА шинный интерфейс производит коммутацию общей шины на местную. По окончании обмена вычислительный модуль - приёмник возвращается в исходное состояние, которое предшествовало внешнему обращению. Такой способ организации вычислительных модулей возможно использование способов связи как с буферизацией, так и без буферизации данных. В обоих случаях могут возникать конфликты за доступ к разделяемой памяти при одновременном запросе её со стороны удалённого и местного процессоров. Разрешение конфликтных ситуаций может производится схемой приоритетов, причём высший приоритет должны иметь запросы, поступающие из общей шины, чтобы не загружать её дополнительной передачей информации о состоянии памяти удалённого вычислительного модуля. Это требование связано с обеспечением высокой пропускной способности общей шины, являющейся «узким местом» системы.

Чтобы минимизировать конфликты за доступ к локальной памяти со стороны местного и удалёного процессоров, её выполняют с расслоением. Ещё большей минимизации конфликтов достигают применением в вычислительных модулях кэш – памяти разных уровней. На рисунке 2.15 представлен вариант построения вычислительного модуля с кэш-памятью.

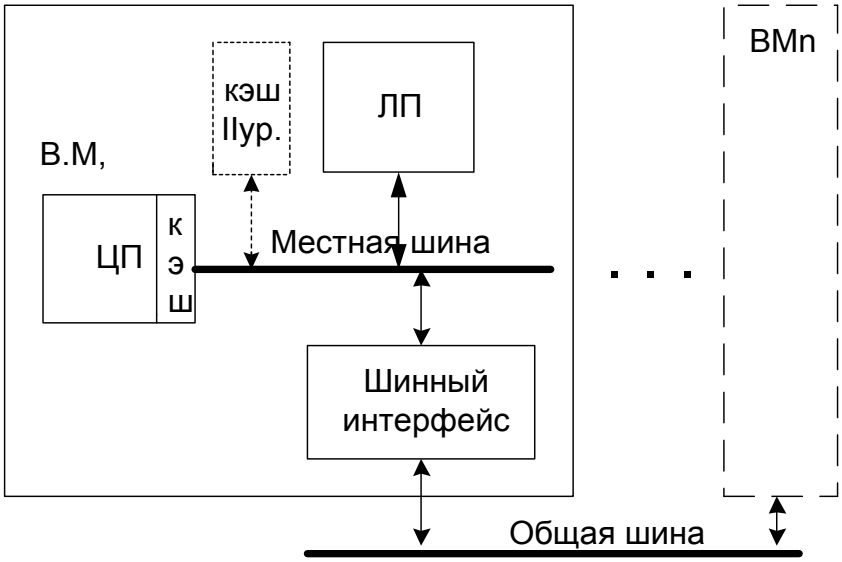


Рис.2.15. Структура вычислительного модуля с кэш-памятью для систем с разделяемой распределённой памятью

Поскольку, в соответствии с принципами временной и пространственной локальности, основная доля обращений при обработке программ вычислительным модулем будет приходиться на кэш-память, то загрузка локальной памяти сокращается. Отсюда следует уменьшение препятствий при доступе к локальной памяти со стороны удалённых вычислительных модулей. Этот же подход применим и к структурам с разделяемой памятью (рис.2.16).

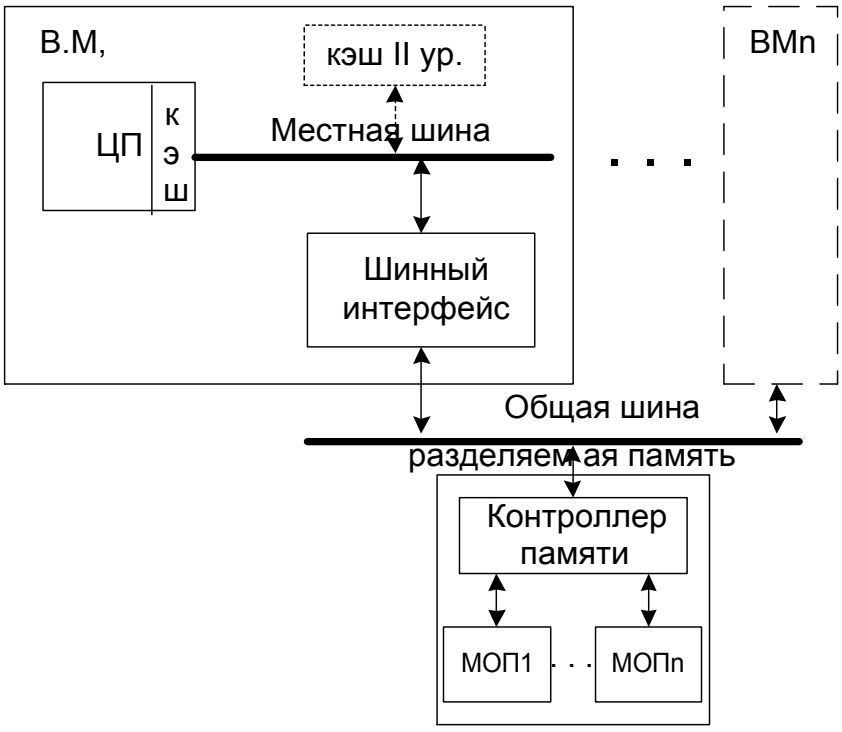


Рис. 2.16. Структура вычислительного модуля с кэш-памятью для систем с разделяемой памятью

Мультипроцессорные операционные системы

Перейдем теперь от обсуждения аппаратного обеспечения мультипроцессоров к рассмотрению программного обеспечения, в частности к обсуждению мультипроцессорных операционных систем. Возможны различные варианты организации данных систем. Ниже будут рассмотрены три из них.

1) Операционные системы с раздельным выполнением заданий

Простейший способ организации мультипроцессорных операционных систем состоит в том, чтобы статически разделить оперативную память по числу центральных процессоров и дать каждому центральному процессору свою собственную память с собственной копией операционной системы. В результате *n* центральных процессоров будут работать как *n* независимых компьютеров. В качестве очевидного варианта оптимизации можно позволить всем центральным процессорам совместно использовать код операционной системы и хранить только индивидуальные копии данных (рис. 8.7). Квадратики, помеченные словом Data, означают приватные данные операционной системы для каждого центрального процессора.

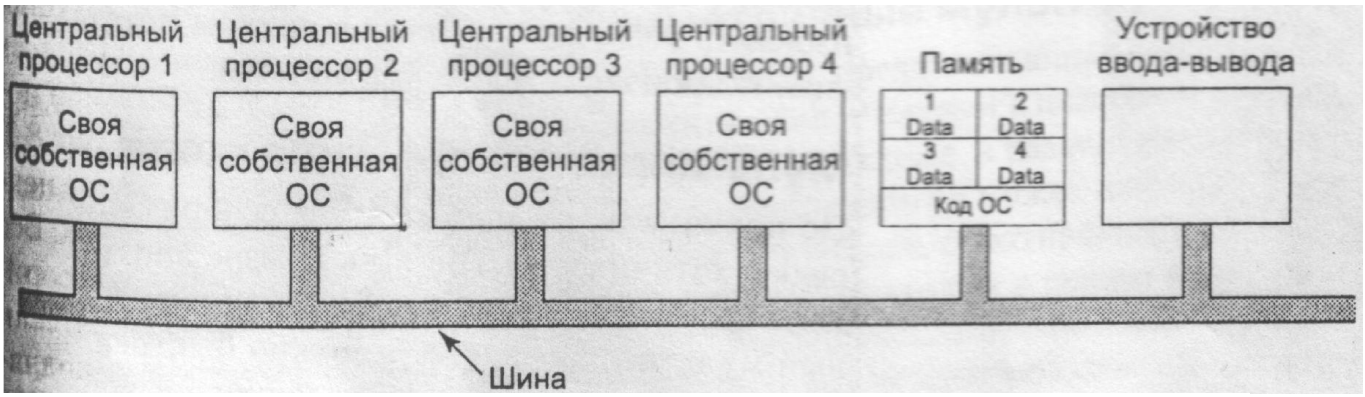


Рис.8.7. Разделение памяти мультипроцессора между четырьмя центральными процессорами с общей копией кода операционной системы

Тем не менее такая схема лучше, чем *n* независимых компьютеров, так как она позволяет всем машинам совместно использовать набор дисков и других устройств ввода-вывода, а также обеспечивает гибкое совместное использование памяти. Например, если требуется запустить большую программу, одному из центральных процессоров может быть выделена большая порция памяти на время выполнения этой программы. Кроме того,

процессы могут эффективно общаться друг с другом, если одному процессу будет позволено писать данные в память, а другой процесс будет их считывать в этом месте. Но с точки зрения операционной системы наличие операционной системы у каждого центрального процессора является крайне примитивным подходом.

Следует отметить четыре аспекта данной схемы, возможно, не являющихся очевидными. Во-первых, когда процесс обращается к системному вызову, системный вызов перехватывается и обрабатывается его собственным центральным процессором при помощи структур данных в таблицах операционной системы.

Во-вторых, поскольку у каждой операционной системы есть свои собственные таблицы, у нее есть также и свой набор процессов, которые она сама планирует совместного использования процессов нет. Если пользователь регистрируется на центральном процессоре 1, то все его процессы работают на центральном процессоре 1. В результате может случиться так, что центральный процессор 1 окажется загружен работой, тогда как центральный процессор 2 будет простаивать.

В-третьих, совместного использования страниц также нет. Может случиться так, что у центрального процессора 2 много свободных страниц, в то время как цен- тральный процессор 1 будет постоянно заниматься свопингом. И нет никакого способа занять свободные страницы у соседнего процессора, так как выделение памяти статически фиксировано.

В-четвертых, и это хуже всего, если операционная система поддерживает буферный кэш недавно использованных дисковых блоков, то каждая операционная система будет выполнять это независимо от остальных. Таким образом, может случиться так, что некоторый блок диска будет присутствовать в нескольких буферах одновременно, причем в нескольких буферах сразу он может оказаться модифицированным, что приведет к порче данных на диске. Единственный способ избежать этого заключается в полном отказе от блочного кэша, что значительно снизит производительность системы.

Мультипроцессоры типа «ведущий-ведомый»

По причине приведенных выше соображений такая модель теперь используется редко, хотя она применялась на заре эпохи мультипроцессоров, когда ставилась цель просто перенести существующие операционные системы на какой-либо новый мультипроцессор как можно быстрее. Вторая модель показана на рис. 8.8. Здесь используется всего одна копия операционной системы, находящаяся на центральном процессоре 1 и отсутствующая на других центральных процессорах, системные вызовы перенаправляются для обработки на центральный процессор Центральный процессор 1 может также выполнять процессы пользователя, если у него будет оставаться для этого время. Такая схема называется «хозяин-подчиненный», так как центральный процессор 1 является - хозяином», то есть ведущим, а все остальные процессоры — подчиненными, или ведомыми.

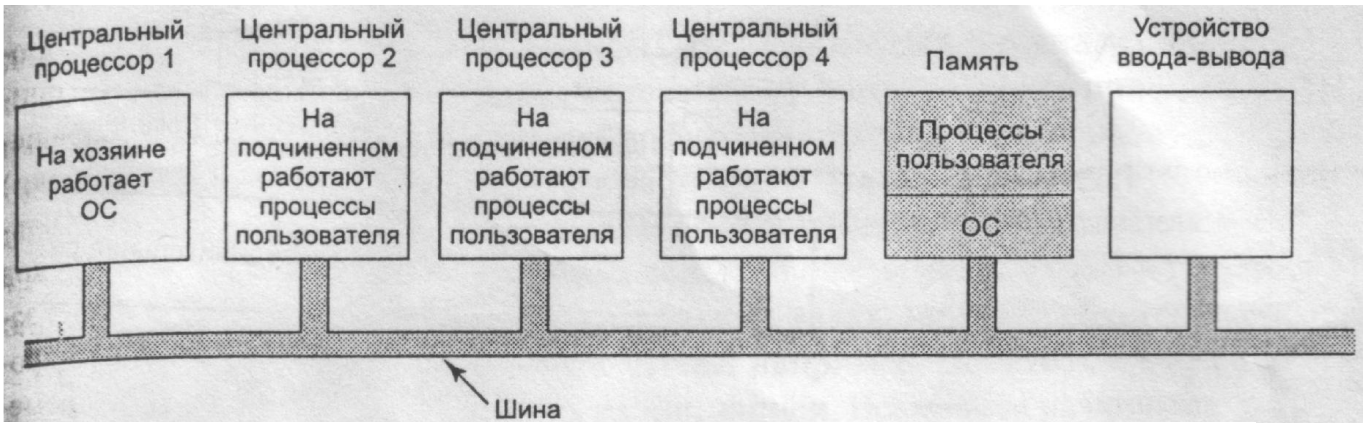


Рис.8.8. Модель мультипроцессора «хозяин-подчинённый»

Модель мультипроцессора «хозяин-подчиненный» позволяет решить большинство проблем первой модели. В этой модели используется единая структура данных (например, один общий список или набор приоритетных списков), учитывающая готовые процессы. Когда центральный процессор переходит в состояние простоя, он запрашивает у операционной системы процесс, который можно обрабатывать, и при наличии готовых процессов операционная система назначает этому процессору процесс. Поэтому при такой организации никогда не может служиться так, что один центральный процессор будет простаивать, в то время как другой центральный процессор перегружен. Страницы памяти могут динамичес**ки** предоставляться всем процессам. Кроме того, в такой системе есть всего один «общий буферный кэш блочных устройств, поэтому дискам не грозит порча данных, как в предыдущей модели при попытке использования блочного кэша. Недостаток этой модели состоит в том, что при большом количестве центральных процессоров хозяин может стать узким местом системы. Ведь ему приходится обрабатывать все системные вызовы от всех центральных процессоров. Например, если обработка системных вызовов занимает 10 % времени, тогда 10 центральных .процессоров завалят хозяина работой, а при 20 центральных процессорах хозяин уже не будет успевать их обрабатывать, и система начнет простаивать. Следовательно, такая модель проста и работоспособна для небольших мультипроцессоров, но на больших она работать не может.

Операционные системы с симметричной обработкой

Третья модель, представляющая собой симметричные мультипроцессоры (SMP, Symmetric Multiprocessor), позволяет устранить перекос предыдущей модели. Как и в предыдущей схеме, в памяти находится всего одна копия операционной системы, но выполнять ее может любой процессор. При системном вызове на центральном процессоре, обратившемся к системе с системным вызовом, происходит прерывание с переходом в режим ядра и обработкой системного вызова. Модель симметричного мультипроцессора показана на рис. 8.9. Эта модель обеспечивает динамический баланс процессов и памяти, поскольку имеется всего один набор таблиц операционной системы. Она также позволяет

избежать простоя системы, связанного с перегрузкой ведущего центрального процессора («хозяина»), так как в ней нет ведущего центрального процессора и все же данная модель имеет собственные проблемы. В частности, если код операционной системы будет выполняться одновременно на двух или более центральных процессорах, произойдет катастрофа. Представьте себе два центральных процессора, одновременно берущих один и тот же процесс для запуска или запрашивающих одну и ту же свободную страницу памяти. Простейший способ разрешения подобных проблем заключается в связывании мьютекса (то есть блокировки) с операционной системой, в результате чего вся система превращается в одну большую критическую область. Когда центральный процессор хочет выполнять код операционной системы, он должен сначала получить мьютекс. Если мьютекс; блокирован, процессор вынужден ждать. Таким образом, любой центральный процессор может выполнить код операционной системы, но в каждый момент в ни только один из них будет делать это.

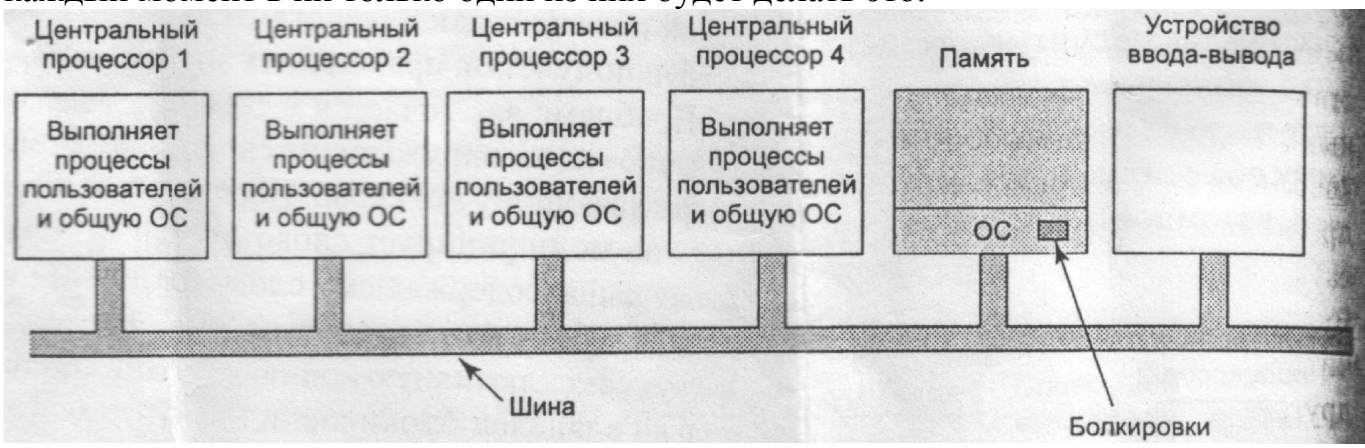


Рис.8.9. Модель симметричного мультипроцессора

Такая модель работает, но она практически так же плоха, как *и* модель «хозяин-подчиненный». Опять же предположим, что 10 % всего процессорного времени расходуется на выполнение самой операционной системы. При 20 центральных процессорах большинство процессоров будут подолгу стоять в длинных очередях, ожидая разрешения доступа к мьютексу. К счастью, такую ситуацию легко исправить. Многие части операционной системы независимы друг от друга. Например, один центральный процессор может заниматься планированием, в то время как другой центральный процессор будет выполнять обращение к файловой системе, а третий обрабатывать страничное прерывание.

Такое наблюдение приводит к расщеплению операционной системы на независимые критические области, не взаимодействующие друг с другом. У критической области есть свой мьютекс, поэтом только один центральный процессор может выполнять ее в каждый момент времени. Таким образом можно достичь большей степени распараллеливания. Однако может случиться, что некоторые таблицы, например таблица процессов, используются в нескольких критических областях. Так, таблица процессов требуется для планирования, но также для выполнения системного вызова

fork и для обработки сигналов. Для каждой таблицы, использующейся несколькими критическими областями, требуется свой собственный мьютекс. Итак, в каждый момент времени каждая критическая область может выполняться только одним центральным процессором, а также к каждой критической таблице может быть предоставлен доступ только полному центральному процессору.

Подобная организация используется в большинстве современных мультипроцессоров. Сложность написания операционной системы для такой машины заключается не в том, что программный код сильно отличается от обычной операционной системы. Нет. Самым сложным является расщепление операционной системы на критические области, которые могут выполняться параллельно на разных центральных процессорах, не мешая друг другу даже косвенно. Кроме того, каждая таблица, используемая двумя и более критическими областями, должна быть отдельно защищена мьютексом, а все программы, пользующиеся этой таблицей, должны корректно использовать мьютекс.

Кроме того, следует уделить особое внимание вопросу избежания взаимоблокировок. Если двум критическим областям одновременно потребуются таблица *A* и таблица *B* и они затребуют эти таблицы в разном порядке, то рано или поздно возникнет взаимоблокировка, причину появления которой будет очень трудно определить. Теоретически всем таблицам можно поставить в соответствие целые числа и потребовать от всех критических областей запрашивать эти таблицы по порядку номеров. Такая стратегия позволяет избежать взаимоблокировок, но она требует от программиста детального исследования того, какие таблицы потребуются каждой критической области, чтобы запросить их в правильном порядке.

При изменении программы со временем критической области может потребоваться новая таблица, которая не была нужна ранее. Если изменением программы занимается новый программист, не понимающий всей логики системы, он может поддаться искушению просто захватить мьютекс в тот момент, когда требуется таблица, и отпустить его, когда таблица более не нужна. Однако именно такие простые и кажущиеся разумными действия могут привести к взаимоблокировке, что с точки зрения пользователя выглядит как зависание системы. Очень не просто грамотно спроектировать систему, но поддерживать ее в правильном состоянии в течение нескольких лет и при этом совершенствовать систему меняющимся штатом программистов крайне трудно.