Г.И.Шпаковский

ОРГАНИЗАЦИЯ ПАРАЛЛЕЛЬНЫХ ЭВМ И СУПЕРСКАЛЯРНЫХ ПРОЦЕССОРОВ

Допущено
Министерством образования
и науки Республики Беларусь
в качестве учебного пособия для
студентов вузов, обучающихся по специальностям
Н.08.01.00 Прикладная математика
Н.08.02.00 Информатика
Т.10.03.00 Вычислительные машины,
системы и сети

Минск
1996

ББК 32973я73
Ш83
УДК 681.3(075.8)

Рецензенты: кафедра программного обеспечения информационных технологий Белорусского государственного университета информатики и радиоэлектроники;
доктор технических наук, профессор М.К.Буза.

Шпаковский Г.И.

Ш83 Организация параллельных ЭВМ и суперскалярных процессоров: Учеб. пособие. — Мн.: Белгосуниверситет, 1996. — 296 с.: ил.

ISBN 985-6144-50-6.

Рассмотрены организация (структуры, коммутация, управление), программное обеспечение (языки, компиляторы, распараллеливатели, операционные системы) и методы создания параллельных алгоритмов для векторных и многопроцессорных ЭВМ.

Подробно изложена архитектура транспьютеров и суперскалярных процессоров (Pentium, Alpha, Power PC), имеется большой объем примеров и справочного материала.

Для студентов старших курсов вузов, специализирующихся в области ЭВМ и прикладной математики, аспирантов и широкого круга инженеров, связанных с разработкой, эксплуатацией и программированием быстродействующих ЭВМ, ПЭВМ и рабочих станций.

Ш 2404010000 **ББК 32973я73**

ISBN 985-6144-50-7.

СПИСОК ОСНОВНЫХ СОКРАЩЕНИЙ

АЛУ — арифметико-логическое устройство
БА — буфер адреса
БАО — буфер адресов операндов
БВП — блок вычитания порядков
БД — база данных
БИС — большая интегральная схема
БК — буфер команд
БМ — базовая машина
БО — буфер операндов
БОЗ — буфер операндов при записи
БОП — блок очереди процессов
БОЧ — буфер операндов при чтении
БС — блок событий
Бсд — блок сдвига
Бсл — блок сложения
БС РОН — блок состояния регистров общего назначения
ВА — вычисление абсолютного адреса
ВО — выборка операнда
ВШ — внутренняя шина
ДУ — дешифрация и управление
ДУЧП — дифференциальное уравнение в частных производных
ИГ — информационный граф
К — коммутатор
КНП — кодовый номер перестановки
КСН — кодовое слово настройки
КЭ — коммутационный элемент
МД — магистраль данных
МУ — магистраль управления
МП — микропроцессор
ОП — оперативная память
ОС — операционная система
ПВВ — процессор ввода-вывода
ПК — память команд
ПМ — процессорная матрица

ПО — память операндов
ПРП — параллельный реляционный процессор
ПП — процессорное поле
ПЭ — процессорный элемент
ПЭВМ — персональная ЭВМ
РА — регистр адреса
РОН — регистр общего назначения
СБИС — сверхбольшая интегральная схема
СМ — системная магистраль
СПО — системное программное обеспечение
ССМП — суперскалярный МП
СчАК — счетчик адресов команд
УВА — устройство вычисления адреса
УВК — устройство выборки команд
УВО — устройство выборки операндов
УК — универсальный коммутатор
УПК — указатель номера пропущенной команды
УУ — устройство управления
ЦМП — центральная многоблочная память
ЦП — центральный процессор
ЦУУ — центральное устройство управления
ЯВУ — язык высокого уровня
ЯПФ — ярусно-параллельная форма
CISC — процессор со сложной системой команд
RISC — процессор с сокращенной системой команд

ПРЕДИСЛОВИЕ

Одним из основных условий успешного развития современного общества является высокий уровень оснащенности средствами вычислительной техники. Причем от этих средств требуется быстродействие в сотни миллионов операций в секунду. Такое быстродействие может быть достигнуто за счет элементной базы и параллельной организации ЭВМ. Из-за принципиальных ограничений все труднее становится получать быстродействие первым путем, поэтому широко распространено создание параллельных ЭВМ различных типов и назначений.

Под параллельной ЭВМ будем понимать ЭВМ, состоящую из множества связанных определенным образом вычислительных блоков, которые способны функционировать совместно и одновременно выполнять множество арифметико-логических операций, принадлежащих одной задаче (программе).

Понятие "организация параллельных ЭВМ и процессоров" включает совокупность свойств, определяющих состав и связи оборудования (структуру ЭВМ), средства программирования (языки, трансляторы, операционные системы), типы используемых вычислительных алгоритмов (алгоритмику).

Существуют различные схемы классификации параллельных ЭВМ.

В некоторых схемах отдельные типы ЭВМ выпадают из числа параллельных. В частности, по структуре конвейерные ЭВМ могут быть отнесены к машинам последовательного действия, но они имеют большую вычислительную мощность и используют параллельную алгоритмику и программирование, поэтому конвейерные ЭВМ также рассматриваются в данном учебном пособии.

Кроме конвейерных ЭВМ в пособии уделяется внимание организации всех широко распространенных типов ЭВМ: процессорным матрицам, многопроцессорным ЭВМ и машинам с управлением от потока данных.

Особое внимание уделено процессорам с параллельной организацией: транспьютерам и суперскалярным микропроцессорам ввиду их широкого использования в ПЭВМ и рабочих станциях.

Изучение параллельных ЭВМ проводится по всем составляющим понятия "организация": структуре, языковому обеспечению, алгоритмике.

Определенные сложности в написании книги возникли вследствие неустоявшейся терминологии, поэтому из множества вариантов некоторого понятия выбирался наиболее распространенный.

Издание написано на основе чтения автором курса лекций "Организация параллельных ЭВМ" в Белорусском государственном университете и в дополнение ранее изданного в 1989 году учебного пособия “Архитектура параллельных ЭВМ” [1]. Предназначено в первую очередь для студентов старших курсов вузов, специализирующихся в области вычислительной техники, а также может быть полезно и студентам-математикам, аспирантам, инженерам. В тексте имеется большое количество примеров. Часть примеров демонстрирует возможность и особенности применения параллелизма в широком круге предметных областей. Примеры математизированы, что позволяет представить их в компактной форме. Другая часть примеров носит детализирующий характер и предназначена для всестороннего рассмотрения некоторой структуры или алгоритма.

Для усвоения изложенных сведений требуется определенная подготовка по структурам последовательных ЭВМ, методам вычислительной математики и языкам программирования.

Автор выражает благодарность преподавателям и сотрудникам кафедры информатики Белорусского государственного университета за обсуждение и содержательные советы, рецензентам: доктору технических наук, профессору М.К.Бузе, доктору технических наук, профессору А.Е.Леусенко за ценные замечания, способствовавшие улучшению данной книги, а также А.И.Кохнюку за большой труд по оформлению издания.

Глава 1

ПРИНЦИПЫ ПАРАЛЛЕЛЬНОЙ ОБРАБОТКИ

§ 1.1. Формы параллелизма в алгоритмах и программах

Известны два основных метода повышения быстродействия ЭВМ: использование все более совершенной элементной базы и параллельное выполнение вычислительных операций.

Максимальная вычислительная мощность интегральной схемы определяется выражением [2]

$$V_{\max} = f \cdot N/a,$$

где f — частота переключения вентилей в микросхеме; N — количество вентилей; a — число переключений, необходимых для выполнения одной арифметико-логической операции.

Известно, что энергия переключения полупроводникового вентиля

$$E = \tau \cdot p,$$

где τ и p — время и мощность переключения вентиля соответственно.

Так как $N = D/p$, где D — допустимая тепловая мощность, рассеиваемая микросхемой, то произведение $f \cdot N$ можно выразить через энергию переключения

$$f \cdot N = D/E.$$

Данные свидетельствуют о том, что могут быть получены достаточно большие значения быстродействия. Однако функциональная схема СБИС и БИС обычно реализует принцип действия последовательных ЭВМ, когда в единицу времени исполняется только одна команда. При этом в работе участвует лишь малая часть вентилей микросхемы, и в результате быстродействие такой последовательной микросхемы $V_{\text{пс}} << V_{\max}$. Поскольку требования к вычислительной мощности неуклонно растут и превосходят не только $V_{\text{пс}}$, но и $V_{\max}$, то необходимо научиться использовать всю потенциальную вычислительную мощность $V_{\max}$ микросхемы. Более того, по ряду фундаментальных физических и технологических

причин не приходится ожидать резкого уменьшения E (роста V_{max}). Следовательно, для увеличения вычислительной мощности ЭВМ необходимо научиться использовать суммарную мощность множества СБИС и БИС, т. е. применять параллелизм.

Параллелизм — это возможность одновременного выполнения нескольких арифметико-логических или служебных операций.

В процессе развития вычислительной техники роль параллельной обработки информации менялась. Если ранее применение параллельных ЭВМ диктовалось необходимостью увеличения надежности оборудования управляющих систем, то современные параллельные ЭВМ используются для ускорения счета в различных прикладных областях.

Особое значение приобретает параллельная обработка для ЭВМ пятого поколения, предназначенных для выполнения алгоритмов искусственного интеллекта. Такие алгоритмы часто носят комбинаторный характер и требуют большой вычислительной мощности.

На стадии постановки задачи параллелизм не определен, он появляется только после выбора метода вычислений. В зависимости от характера этого метода и профессиональной подготовки пользователя параллелизм алгоритма (то есть количество одновременно выполняемых операций) может меняться в довольно больших пределах.

Параллелизм используемой ЭВМ также меняется в широких пределах и зависит в первую очередь от числа процессоров, способа размещения данных, методов коммутации и способов синхронизации процессов.

Язык программирования является средством переноса параллелизма алгоритма на параллелизм ЭВМ, и тип языка может в сильной степени влиять на результат переноса.

Для сравнения параллельных алгоритмов необходимо уметь оценивать степень параллелизма. Одновременное выполнение операций возможно, если они логически независимы. Таким образом, описание зависимостей операций по данным полностью определяет параллелизм некоторого метода вычислений.

Программные объекты A и B (команды, операторы, программы) являются независимыми и могут выполняться параллельно, если выполняется следующее условие:

$$(InB \wedge OutA) \vee (InA \wedge OutB) \wedge (OutA \wedge OutB) = \varnothing, \qquad (1.1)$$

где In(A) — набор входных, а Out(A) — набор выходных переменных объекта A. Если условие (1.1) не выполняется, то между A и B существует зависимость и они не могут выполняться параллельно.

Если условие (1.1) нарушается в первом терме, то такая зависимость называется прямой. Приведем пример:

A: R = R1 + R2
B: Z = R + C

Здесь операторы A и B не могут выполняться одновременно, так как результат A является операндом B.

Если условие нарушено во втором терме, то такая зависимость называется обратной:

A: R = R1 + R2
B: R1 = C1 + C2

Здесь операторы A и B не могут выполняться одновременно, так как выполнение B вызывает изменение операнда в A.

Наконец, если условие не выполняется в третьем терме, то такая зависимость называется конкуренционной:

A: R = R1 + R2
B: R = C1 + C2

Здесь одновременное выполнение операторов дает неопределенный результат.

Увеличение параллелизма любой программы заключается в поиске и устранении указанных зависимостей.

Наиболее общей формой представления этих зависимостей является *информационный граф* задачи (ИГ). Пример ИГ, описывающего логику конкретной задачи, точнее порядок выполнения операций в задаче, приведен на рис. 1.1. В своей первоначальной форме ИГ, тем не менее, не используется ни математиком, ни программистом, ни ЭВМ. На этапе разработки метода вычислений ИГ представляется в виде формул и блок-схем алгоритма, на этапе программирования — в виде последовательности операторов на языке высокого уровня (ЯВУ) или машинном языке, на этапе счета — в виде порядка выборки команд устройством управления (УУ) процессора. На каждом этапе логика задачи (ИГ) отображается в

форме, наиболее пригодной для восприятия человеком или машиной. Однако поскольку на всех уровнях представления задачи логика ее исполнения неизменна, то ИГ, восстановленные из различных форм представления конкретной задачи, должны совпадать. Некоторое отличие копий ИГ возможно за счет введения промежуточных переменных, используемых для удобства записи, работы с памятью и регистрами. Таким образом, ИГ является внутренней характеристикой любой формы представления задачи.

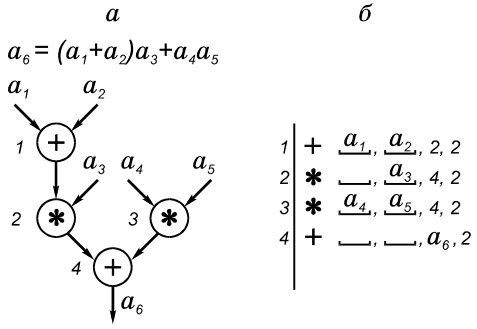


Рис.1.1. Информационный граф математического выражения (*а*) и порядок выполнения операций в выражении (*б*)

Более определенной формой представления параллелизма является *ярусно-параллельная форма* (ЯПФ) [3]: алгоритм вычислений представляется в виде ярусов, причем в нулевой ярус входят операторы (ветви), не зависящие друг от друга, в первый ярус — операторы, зависящие только от нулевого яруса, во второй — от первого яруса и т. д.

Для ЯПФ характерны параметры, в той или иной мере отражающие степень параллелизма метода вычислений: b_i — ширина i-го яруса; B — ширина графа ЯПФ (максимальная ширина яруса, т. е. максимум из b_i, $i = 1, 2, ...$); l_i — длина яруса (время операций) и L длина графа; ε — коэффициент заполнения ярусов; θ — коэффициент разброса указанных параметров и т. д.

Главной задачей настоящего издания является изучение связи между классами задач и классами параллельных ЭВМ. Но представление параллелизма в виде большого набора количественных характеристик ЯПФ сильно затрудняет данную задачу. Поэтому желательно упростить представление ЯПФ, разделив их на типичные виды, группы, которые в дальнейшем будем называть формами параллелизма. *Форма параллелизма* обычно достаточно просто

характеризует некоторый класс прикладных задач и предъявляет определенные требования к структуре, необходимой для решения этого класса задач параллельной ЭВМ.

Изучение ряда алгоритмов и программ показало, что можно выделить следующие основные формы параллелизма: *естественный* или *векторный параллелизм*; *параллелизм независимых ветвей*; *параллелизм смежных операций* или *скалярный параллелизм*.

Векторный параллелизм. Наиболее распространенной в обработке структур данных является векторная операция (естественный параллелизм). *Вектор* — одномерный массив, который образуется из многомерного массива, если один из индексов не фиксирован и пробегает все значения в диапазоне его изменения. В параллельных языках этот индекс обычно обозначается знаком *. Пусть, например, A, B, C — двумерные массивы. Тогда можно записать такую векторную операцию:

C (*, j) = A (*, j) + B (*, j).

На последовательных языках эта операция выражается с помощью цикла DO:

```
      DO 1 I = 1,N
1     C(I,J) = A(I,J) + B(I,J)
```

Возможны операции и большей размерности, чем векторные: над матрицами и многомерными массивами. Однако в параллельные ЯВУ включаются только векторные операции (сложение, умножение, сравнение и т. д.), потому что они носят универсальный характер, тогда как операции более высокого уровня специфичны и возможное их число слишком велико для включения в любой язык. Операции высоких порядков выражаются в виде некоторой процедуры вычислений через векторные операции.

Области применения векторных операций над массивами обширны: цифровая обработка сигналов (цифровые фильтры); механика, моделирование сплошных сред; метеорология; оптимизация; задачи движения; расчеты электрических характеристик БИС и т. д.

Рассмотрим решение линейной системы уравнений:

$$a_{11}x_1 + a_{12}x_2 + \ldots + a_{1n}x_n = b_1,$$
$$a_{21}x_1 + a_{22}x_2 + \ldots + a_{2n}x_n = b_2,$$
$$\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots$$
$$a_{n1}x_1 + a_{n2}x_2 + \ldots + a_{nn}x_n = b_n.$$

Для решения таких систем уравнений при положительно определенной матрице коэффициентов используется метод простой итерации [4]:

$$x_1^{(k+1)} = c_{11}x_1^{(k)} + c_{12}x_2^{(k)} + \ldots + c_{1n}x_n^{(k)} + d_1$$
$$x_2^{(k+1)} = c_{21}x_1^{(k)} + c_{22}x_2^{(k)} + \ldots + c_{2n}x_n^{(k)} + d_2$$
$$\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots$$
$$x_n^{(k+1)} = c_{n1}x_1^{(k)} + c_{n2}x_2^{(k)} + \ldots + c_{nn}x_n^{(k)} + d_n.$$

Пусть *C(N,N)* — матрица коэффициентов $c_{i,j}$ системы уравнений; *D (N)* — вектор $d_1, d_2, \ldots, d_n$; *XK(N)* — вектор $x_1^{(k)}, x_2^{(k)}, \ldots, x_n^{(k)}$ (в исходный момент хранит начальное приближение); *XK1(N)* — вектор $x_1^{(k+1)}, x_2^{(k+1)}, \ldots, x_n^{(k+1)}$; ε — заданная погрешность вычислений. Тогда программа для параллельной ЭВМ может выглядеть следующим образом:

```
    DIMENSION C(N,N), D(N), XK1(N), XK(N)
    XK(*) = начальные значения
    XK1(*) = D(*)
4   DO 1 I = 1,N
1   XK1(*) = XK1(*) + C(*,I)*XK(I)
    IF (ABS(XK1(*)-XK(*))-ε) 2,2,3
3   XK(*) = XK1(*)
    GO TO 4
2   Вывод XK1(*)
    STOP
```

В этой программе многократно использована параллельная обработка элементов векторов (практически во всех строках программы).

Параллелизм независимых ветвей. Суть параллелизма независимых ветвей состоит в том, что в программе решения большой задачи могут быть выделены независимые программные части — ветви. Под независимостью ветвей в первую очередь понимается независимость по данным.

В параллельных языках запуск *параллельных ветвей* осуществляется с помощью оператора FORK M1, M2 , ..., ML, где M1, M2, ..., ML — имена независимых ветвей. Каждая ветвь заканчивается оператором JOIN (R,K), выполнение которого вызывает вычитание единицы из ячейки памяти R. Так как в R предварительно записано число, равное количеству ветвей, то при последнем срабатывании оператора JOIN (все ветви выполнены) в R оказывается нуль и управление передается на оператор K. Иногда в JOIN описывается подмножество ветвей, при выполнении которого срабатывает этот оператор.

В последовательных языках аналогов операторам FORK и JOIN нет и ветви выполняются друг за другом.

Рассмотрим пример задачи с параллелизмом ветвей.

Пусть задана система уравнений

$$
\begin{aligned}
&f_1(x_1, x_2, \mathrm{K}\ , x_n) = 0,\\
&f_2(x_1, x_2, \ldots, x_n) = 0,\\
&\ldots\ldots\ldots\ldots\ldots\ldots\\
&f_n(x_1, x_2, \ldots, x_n) = 0.
\end{aligned}
$$

Эту систему можно вычислять методом итераций по следующим формулам (n=3):

$$
\begin{aligned}
x_1^{(k+1)} &= F_1(x_1^{(k)}, x_2^{(k)}, x_3^{(k)}),\\
x_2^{(k+1)} &= F_2(x_1^{(k)}, x_2^{(k)}, x_3^{(k)}),\\
x_3^{(k+1)} &= F_3(x_1^{(k)}, x_2^{(k)}, x_3^{(k)}).
\end{aligned}
$$

Функции $F_1 ... F_3$ из-за их различной программной реализации должны вычисляться отдельными программными сегментами, которые можно использовать как ветви параллельной программы.

Соответствующая параллельная программа имеет следующий вид:

```
L     FORK M1, M2, M3
M1    Z1 = F1 (X1, X2, X3)
      JOIN (R, K)
M2    Z2 = F2 (X1, X2, X3)
      JOIN (R,K)
M3    Z3 = F3 (X1, X2, X3)
      JOIN (R, K)
K     IF (ABS(Z1-X1)<ε)AND(ABS(Z2-X2)<ε)AND(ABS(Z3-
      X3)<ε)
          THEN вывод результатов; STOP
          ELSE X1=Z1; X2=Z2; X3=Z3; GO TO L
```

Если при выполнении оператора JOIN оказалось, что $R \neq 0$, то вычисления в данной ветви останавливаются, но могут быть запущены повторно, если условие в операторе K не выполняется.

Для приведенного примера характерны две особенности: 1) присутствует *синхронизация процессов*, для которой используются оператор JOIN и ячейка R; состояние R = 0 свидетельствует об окончании процессов разной длительности; 2) производится *обмен данными* (обращение за X1...X3 из разных ветвей).

Скалярный параллелизм. При исполнении программы регулярно встречаются ситуации, когда исходные данные для *i*-й операции вырабатываются заранее, например, при выполнении (*i* - 2)-й или (*i* - 3)-й операции. Тогда при соответствующем построении вычислительной системы можно совместить во времени выполнение *i*-й операции с выполнением (*i* - 1)-й, (*i* - 2)-й, ... операций. В таком понимании скалярный параллелизм похож на параллелизм независимых ветвей, однако они очень отличаются длиной ветвей и требуют разных вычислительных систем. Рассмотрим пример.

Пусть имеется программа для расчета ширины запрещенной зоны транзистора и в этой программе есть участок — определение энергии примесей по формуле

$$E = \frac{mq^4\pi^2}{8\varepsilon_0^2\varepsilon^2 h^2}.$$

Тогда последовательная программа для вычисления E будет такой:

```
F1 = M * Q ** 4 * P ** 2
F2 = 8 * E0 ** 2 * E ** 2 * H ** 2
E = F1/F2
```

Здесь имеется параллелизм, но при записи на Фортране (показано выше) или Ассемблере у нас нет возможности явно отразить его. Явное представление параллелизма для вычисления E задается ЯПФ (см. рис. 1.2).

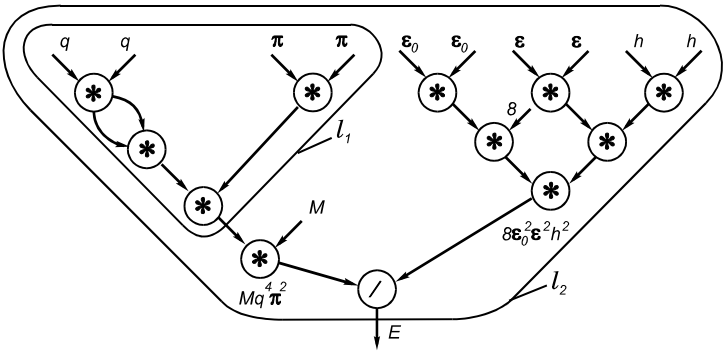


Рис. 1.2. ЯПФ вычисления величины E

Ширина параллелизма первого яруса этой ЯПФ (первый такт) сильно зависит от числа операций, включаемых в состав ЯПФ. Так, в примере для $l_1 = 4$ параллелизм первого такта равен двум, для $l_1 = 12$ параллелизм равен пяти.

Поскольку это параллелизм очень коротких ветвей и с помощью операторов FORK и JOIN описан быть не может (вся программа будет состоять в основном из этих операторов), данный вид параллелизма должен автоматически выявляться аппаратурой ЭВМ в процессе выполнения машинной программы.

Для скалярного параллелизма часто используют термин мелкозернистый параллелизм (МЗП), в отличие от крупнозернистого параллелизма (КЗП), к которому относят векторный параллелизм и параллелизм независимых ветвей.

§ 1.2. Организация и эффективность параллельных ЭВМ

Критерии классификации параллельных ЭВМ могут быть разными: вид соединения процессоров, способ функционирования процессорного поля, область применения и т.д.

Одна из наиболее известных *классификаций параллельных ЭВМ* предложена Флинном [2] и отражает форму реализуемого ЭВМ параллелизма. Основными понятиями классификации являются "поток команд" и "поток данных". Под *потоком команд* упрощенно понимают последовательность команд одной программы. *Поток данных* — это последовательность данных, обрабатываемых одной программой.

Согласно классификации Флинна имеется четыре больших класса ЭВМ:

1) ОКОД — *одиночный поток команд-одиночный поток данных*. Это последовательные ЭВМ, в которых выполняется единственная программа, т. е. имеется только один счетчик команд и одно арифметико-логическое устройство (АЛУ). Сюда можно, например, отнести все модели ЕС ЭВМ;

2) ОКМД — *одиночный поток команд-множественный поток данных*. В таких ЭВМ выполняется единственная программа, но каждая ее команда обрабатывает много чисел. Это соответствует векторной форме параллелизма. В классе ОКМД будут рассмотрены два типа ЭВМ: с арифметическими конвейерами (векторно-конвейерные ЭВМ) и процессорные матрицы. К классу ОКМД-ЭВМ относятся машины CRAY-1, ПС-2000, ILLIAC-IV;

3) МКОД — *множественный поток команд-одиночный поток данных*. Подразумевается, что в данном классе несколько команд одновременно работает с одним элементом данных, однако эта позиция классификации Флинна на практике не нашла применения;

4) МКМД — *множественный поток команд-множественный поток данных.* В таких ЭВМ одновременно и независимо друг от друга выполняется несколько программных ветвей, в определенные промежутки времени обменивающихся данными. К машинам этого класса относится ЭВМ "Эльбрус". В классе МКМД будут рассмотрены машины двух типов: с управлением от потока команд (IF — instruction flow) и управлением от потока данных (DF — data flow). Если в ЭВМ первого типа используется традиционное выполнение команд по ходу их расположения в программе, то при-

менение ЭВМ второго типа предполагает активацию операторов по мере их текущей готовности.

В случае DF все узлы информационного графа задачи представляются в виде отдельных операторов:

КОП О1, О2, А3, БС,

где О1, О2 — поля для приема первого и второго операндов от других операторов; А3 — адрес (имя) оператора, куда посылается результат; БС — блок событий. В БС записывается число, равное количеству операндов, которое нужно принять, чтобы начать выполнение данного оператора. После приема очередного операнда из БС вычитается единица; когда в БС оказывается нуль, оператор начинает выполняться (см. рис. 1.1). Программа полностью повторяет ИГ, но ее операторы могут располагаться в памяти произвольно. Выполняться они будут независимо от начального расположения в следующем порядке:

такты	номера операторов
1	1,3
2	2
3	4

Во всех случаях последовательность вычислений определяется связями операторов, поэтому оператор 2 никогда не будет выполнен раньше оператора 1, оператор 4 — раньше оператора 3, т. е. сохраняется логика ИГ. Вычислительный процесс управляется готовностью данных. Это и есть *управление потоком данных*. Считается, что такая форма представления ИГ обеспечивает наибольший потенциальный параллелизм.

Наибольшее распространение на практике получили следующие виды параллельных ЭВМ: ОКМД-ЭВМ с конвейерной организацией, ОКМД-ЭВМ на основе процессорных матриц, МКМД-ЭВМ с общей памятью, МКМД-ЭВМ с локальной памятью (в частности, на транспьютерах) и суперскалярные ЭВМ. Именно эти типы ЭВМ и будут рассмотрены в последующих главах.

В зависимости от быстродействия параллельные ЭВМ можно разделить на суперЭВМ и ЭВМ с относительно небольшим быстродействием: минисуперЭВМ, рабочие станции, персональные суперЭВМ.

Вопрос об эффективности параллельных ЭВМ возникает на разных стадиях исследования и разработки ЭВМ. Следует различать эффективность параллельных ЭВМ в процессе их функционирования и эффективность параллельных алгоритмов. Эффективность алгоритмов рассматривается в главе 6.

В зависимости от стадии разработки полезными оказываются различные *характеристики эффективности* ЭВМ. Рассмотрим следующие характеристики:

1. *Ускорение r* параллельной системы, которое используется на начальных этапах проектирования или в научных исследованиях для оценки предельных возможностей архитектуры.

2. *Быстродействие V*, которое является главной характеристикой при конкретном проектировании или выборе существующей параллельной ЭВМ под класс пользовательских задач.

Ускорение определяется выражением:

$$r = T_1 / T_n ,$$

где T_1 — время решения задачи на однопроцессорной системе, а T_n — время решения той же задачи на n — процессорной системе.

Пусть $W = W_{ск} + W_{пр}$, где W — общее число операций в задаче, $W_{пр}$ — число операций, которые можно выполнять параллельно, а W_{ck} — число скалярных (нераспараллеливаемых) операций.

Обозначим также через t время выполнения одной операции. Тогда получаем известный *закон Амдала* [4]:

$$r = \frac{W \cdot t}{(W_{ck} + \frac{W_n}{n}) \cdot t} = \frac{1}{a + \frac{1-a}{n}} \xrightarrow[n\to\infty]{} \frac{1}{a} . \qquad (1.2)$$

Здесь $a = W_{ck} / W$ — удельный вес скалярных операций.

Закон Амдала определяет принципиально важные для параллельных вычислений положения:

1. Ускорение вычислений зависит как от потенциального параллелизма задачи (величина 1-a), так и от параметров аппаратуры (числа процессоров n).

2. Предельное ускорение вычислений определяется свойствами задачи.

Пусть, например, a = 0,2 (что является реальным значением), тогда ускорение не может превосходить 5 при любом числе процессоров, то есть максимальное ускорение определяется потенциальным параллелизмом задачи. Очевидной является чрезвычайно высокая чувствительность ускорения к изменению величины a.

Выражение (1.2) определяет ускорение только одного уровня вычислительной системы. Однако, реальные системы являются многоуровневыми как с точки зрения программных конструкций, так и по аппаратуре, что и показано в таблице 1.1.

Реальные параллельные ЭВМ обычно используют параллелизм нескольких уровней и полное ускорение такой ЭВМ R можно в первом приближении описать выражением:

$$R = \prod_{i=1}^{M} r_i , \qquad (1.3)$$

где M — число вложенных уровней вычислений, используемых для распараллеливания, а r_i — собственное ускорение уровня i, определяемое параллелизмом соответствующих данному уровню объектов: независимых задач, программ, ветвей, итераций цикла, операторов, отдельных операций выражения.

Таблица 1.1.

Уровни программных конструкций
и соответствующие им структуры ЭВМ

№ п/п	Уровни программных конструкций для параллельного исполнения	Структуры вычислительных систем
1.	Независимые задачи и программы	Многомашинные системы
2.	Программы и подпрограммы одной задачи	МКМД-ЭВМ с общей и локальной памятью
3.	Ветви и циклы одной программы	МКМД-ЭВМ, ОКМД конвейерного типа или процессорные матрицы
4.	Операторы, выражения, линейные участки программ	Системы для суперскалярных вычислений

Наиболее важной характеристикой параллельных ЭВМ при проектировании или выборе новой ЭВМ является быстродействие. Различают следующие разновидности характеристик быстродействия:

1. Номинальное (максимальное, пиковое) быстродействие:

$$V_н = n / \sum_{i=1}^{k} \gamma_i t_i \qquad (1.4.)$$

Здесь n — число процессоров или АЛУ; k — число различных команд в списке команд ЭВМ; γ_i — удельный вес команд i-го типа в программе, t_i — время выполнения команды типа i. Если t_i задавать в тактах, то выражение (1.4) будет определять архитектурную скорость, измеряемую числом команд, выполняемых за один такт. Этот параметр особенно важен для суперскалярных процессоров. Веса команд определяют путем сбора статистики по частотам команд в реальных программах. Известны смеси команд Гибсона, Флинна и ряд других.

Для микропроцессоров с сокращенной системой команд, в которых большинство команд выполняется за один такт, вычисление $V_н$ упрощается:

$$V_н = n / t = n \cdot f ,$$

где f — частота синхронизации микропроцессора, а n — число одновременно выполняемых команд (для суперскалярных микропроцессоров).

Очевидно, что номинальное быстродействие определяет только свойства оборудования, причем, в предположении, что оно полностью загружено в каждом такте, что, конечно, далеко от действительности.

В качестве единицы быстродействия используются:

а) МИПС (MIPS-Million Instructions per Second) — как для целочисленных операций, так и для смеси команд, входящих в состав эталонной программы;

б) Мегафлопс (MFlops-Million Float Instructions per Second) — как для операций с плавающей запятой, так и для смеси команд с преобладанием плавающих команд.

2. Реальное быстродействие ЭВМ $V_{\text{р}}$ определяется с учетом всех факторов, сопутствующих выполнению пользовательских программ. Наилучшим способом определения $V_{\text{р}}$ было бы выполнение реальных пользовательских задач и измерение времени их выполнения, тогда можно было бы считать, что

$$V_{\text{р}} = (\sum_{i=1}^{k} z_i) / T_к,$$

где k — число выполненных задач; z_i — число выполненных в i-й задаче команд; T_k — время решения k задач.

Реальное быстродействие обычно в 5-10 раз меньше номинального.

Однако, при проектировании или выборе наиболее подходящей ЭВМ среди уже существующих, программы пользователя для этой ЭВМ еще не написаны. В этих случаях для характеристики быстродействия $V_{\text{р}}$ используются стандартные *тестовые программы*, называемые эталонами (benchmarks). Эти программы могут быть созданы искусственным путем с учетом статистики реальных программ, могут быть специальным набором частей реальных программ, могут быть смесью двух предыдущих типов программ. Главные требования к этим тестовым программам состоят в следующем:

1. Тестовые программы должны учитывать эффективность работы операционной системы, компиляторов, системы ввода вывода, всех видов памяти (включая КЭШ), процессоров, то есть тех частей ЭВМ, которые работают при выполнении реальных программ.

2. Тестовые программы должны быть достаточно точными, то есть давать результаты, которые будут близки к результатам, полученным при выполнении реальных программ. Наиболее известны следующие тестовые программы: LINPACK, Livermore, Fortran Kernel, SPECint, SPECfp и др.

§ 1.3. Основные этапы развития параллельной обработки

Идея параллельной обработки возникла одновременно с появлением первых вычислительных машин. В начале 50-х гг. американский математик Дж. фон Нейман предложил архитектуру последовательной ЭВМ, которая приобрела классические формы и применяется практически во всех современных ЭВМ. Однако фон Нейман разработал также принцип построения процессорной матрицы, в которой каждый элемент был соединен с четырьмя соседними и имел 29 состояний. Он теоретически показал, что такая матрица может выполнять все операции, поскольку она моделирует поведение машины Тьюринга.

Практическая реализация основных идей параллельной обработки началась только в 60-х гг. нашего столетия. Это связано с появлением транзистора, который благодаря малым размерам и высокой надежности по сравнению с электронными лампами позволил строить машины, состоящие из большого количества логических элементов, что принципиально необходимо для реализации любой формы параллелизма.

Появление параллельных ЭВМ с различной организацией (конвейерные ЭВМ, процессорные матрицы, многопроцессорные ЭВМ) вызвано различными причинами. Совершенствование этих ЭВМ происходило по внутренним законам развития данного типа машин.

Рассмотрим основные этапы совершенствования параллельных ЭВМ для каждого типа структур.

Конвейерные ЭВМ. Основополагающим моментом для развития конвейерных ЭВМ явилось обоснование академиком

С.А.Лебедевым в 1956 г. метода, названного “принципом водопровода” (позже он стал называться *конвейером*). Прежде всего был реализован конвейер команд, на основании которого практически одновременно были построены советская ЭВМ БЭСМ-6 (1957-1966 гг., разработка Института точной механики и вычислительной техники АН СССР) и английская машина ATLAS (1957-1963 гг.) [5,6]. *Конвейер команд* предполагал наличие многоблочной памяти и секционированного процессора, в котором на разных этапах обработки находилось несколько команд.

Конвейер команд позволил получить в ЭВМ БЭСМ-6 быстродействие в 1 млн оп/с. В дальнейшем конвейеры команд совершенствовались и стали необходимым элементом всех быстродействующих ЭВМ, в частности, использовались в известных семействах ЭВМ IBM/360 и ЕС ЭВМ.

Следующим заметным шагом в развитии конвейерной обработки, реализованном в ЭВМ CDC-6600 (1964 г.), было введение в состав процессора нескольких функциональных устройств, позволяющих одновременно выполнять несколько арифметико-логических операций: сложение, умножение, логические операции [5].

В конце 60-х гг. был введен в использование *арифметический конвейер*, который нашел наиболее полное воплощение в ЭВМ CRAY-1 (1972-1976 гг.) [2]. Арифметический конвейер предполагает разбиение цикла выполнения арифметико-логической операции на ряд этапов, для каждого из которых отводится собственное оборудование. Таким образом, на разных этапах обработки находится несколько чисел, что позволяет производить эффективную обработку вектора чисел.

Сочетание многофункциональности, арифметического конвейера для каждого функционального блока и малой длительности такта синхронизации позволяет получить быстродействие в десятки и сотни миллионов операций в секунду. Такие ЭВМ называются *супер*ЭВМ.

Процессорные матрицы. Идея получения сверхвысокого быстродействия в первую очередь связывалась с *процессорными матрицами* (ПМ). Предполагалось, что, увеличивая в нужной степени число процессорных элементов в матрице, можно получить любое заранее заданное быстродействие. Именно этим объясняет-

ся большой объем теоретических разработок по организации процессорных матриц.

В 60-х гг. в Институте математики Сибирского отделения АН СССР под руководством доктора технических наук Э.В.Евреинова сформировалось комплексное научное направление “Однородные системы, структуры и среды”, которое получило развитие не только в СССР, но и за рубежом. Книга “Однородные универсальные вычислительные системы высокой производительности”, изданная в 1966 г., была одной из первых в мире монографий, посвященных этой теме. В работах по однородным средам и системам исследовались матрицы с различными связями между элементами и конфигурацией, а также с элементами разной сложности и составом функций.

Поскольку в 60-е гг. логические схемы с большим уровнем интеграции отсутствовали, то напрямую реализовать принципы функционирования процессорной матрицы, содержащей множество элементарных процессоров, не представлялось возможным. Поэтому для проверки основных идей строились однородные системы из нескольких больших машин. Так, в 1966 г. была построена система Минск-222, спроектированная совместно с Институтом математики Сибирского отделения АН СССР и минским заводом ЭВМ им. Г.К.Орджоникидзе [5]. Система содержала до 16 соединенных в кольцо ЭВМ Минск-2. Для нее было разработано специальное математическое обеспечение.

Другое направление в развитии однородных сред, основанное на построении процессорных матриц, состоящих из крупных процессорных элементов с достаточно большой локальной памятью, возникло в США и связано с именами Унгера, Холланда, Слотника. Была создана ЭВМ ILLIAC-IV (1966-1975 гг.), которая надолго определила пути развития процессорных матриц [2]. В машине использовались матрицы 8×8 процессоров, каждый с быстродействием около 4 млн оп/с и памятью 16 кбайт. Для ILLIAC-IV были разработаны кроме Ассемблера еще несколько параллельных языков высокого уровня. ЭВМ ILLIAC-IV позволила отработать вопросы коммутации, связи с базовой ЭВМ, управления вычислительным процессом. Особенно ценным является опыт разработки параллельных алгоритмов вычислений, определивший области эффективного использования подобных машин.

Примерно в то же время в СССР была разработана близкая по принципам построения ЭВМ М-10. Затем процессорные матрицы стали разрабатываться и выпускаться в ряде стран в большом количестве. Широко известной советской ЭВМ этого класса являлась машина ПС-2000, разработанная Институтом проблем управления АН СССР и запущенная в 1982 г. в производство [7].

Одной из наиболее сложных частей процессорных матриц является *система коммутации*. Самые простые ее варианты — кольцо и регулярная матрица соединений, где каждый узел связан только с двумя или четырьмя соседними, сильно ограничивают класс эффективно решаемых задач. Поэтому на протяжении всего времени существования параллельной обработки разрабатывались коммутаторы с более широкими возможностями: многомерные кубы, универсальные коммутаторы, коммутационные среды [2]. В СССР работы, связанные с коммутаторами для ПМ, наибольшее развитие получили в Таганрогском радиотехническом институте [8].

Многопроцессорные ЭВМ. Одной из первых полномасштабных многопроцессорных систем явилась система D825 фирмы “BURROUGHS” [5]. Начиная с 1962 г. было выпущено большое число экземпляров и модификаций D825.

Выпуск первых многопроцессорных систем, в частности D825, диктовался необходимостью получения не высокого быстродействия, а высокой живучести ЭВМ, встраиваемых в военные командные системы и системы управления. С этой точки зрения параллельные ЭВМ считались наиболее перспективными.

Система D825 содержала до четырех ПЭ и 16 модулей памяти, соединенных матричным коммутатором, который допускал одновременное соединение любого процессора с любым блоком памяти.

Существует мнение, что система D825 получила широкое распространение потому, что для нее впервые была разработана полноценная операционная система ASOR, обеспечившая синхронизацию процессов и распределение ресурсов.

В дальнейшем в СССР и на Западе были разработаны многопроцессорные системы [5], в которых все большее внимание уделялось операционным системам, языкам программирования, параллельной вычислительной математике.

Совершенствование микроэлектронной элементной базы, появление в 80-х годах БИС и СБИС позволили перейти к реализации структур с очень большим количеством ПЭ. Появились разработки по систолическим массивам, реализации многопроцессорных систем с программируемой архитектурой [7], ЭВМ с управлением от потока данных.

Большая плотность упаковки транзисторов на кристалле позволила разместить в одной микросхеме несколько АЛУ. Это позволило реализовать принцип суперскалярной обработки.

Если в последовательной ЭВМ пользователь в процессе программирования задачи в основном следит за логикой метода вычислений, то в параллельной ЭВМ ему приходится дополнительно заниматься размещением данных и синхронизацией вычислительных процессов. Это означает, что процесс программирования существенно усложняется, особенно если программирование ведется на уровне Ассемблера, чтобы повысить эффективность использования дорогостоящих параллельных ЭВМ. Такое программирование доступно только профессионально подготовленным программистам.

По мере удешевления и более широкого распространения параллельных ЭВМ доступ к ним получили пользователи, которые не являются профессиональными программистами, поэтому вопрос о системах программирования значительно обострился. Для облегчения программирования для каждой ЭВМ стали создаваться параллельные ЯВУ. Так, для ILLIAC-IV было создано несколько языков: TRANQUIL, IVTRAN, GLYPNIR; для ЭВМ Эльбрус — язык ЭЛЬ-76 и адаптированы языки Алгол-68, Фортран, Паскаль и PL-1. Применение ЯВУ освобождает пользователя от знания многих деталей структуры конкретной ЭВМ. Однако полностью освободиться от влияния структуры ЭВМ на язык не удается, поэтому возникла проблема *мобильности программного обеспечения*, под которой понимают возможность переноса программ с одной параллельной ЭВМ на другую с минимум переделок или вообще без них.

Очевидным средством обеспечения мобильности является разработка и использование (в качестве стандартных) проблемно-ориентированных языков, включающих средства описания параллельных вычислений. К ним относятся языки Алгол-68, ADA, Фортран-8X.

Большое внимание уделяется разработке автоматических распараллеливателей последовательных программ для исполнения на параллельных машинах различных типов [9, 10].

Наиболее радикальный путь автоматизации программирования вообще и параллельного в частности состоит в использовании методов искусственного интеллекта. В СССР это направление получило широкое развитие, разработаны системы программирования ДИЛОС, СПОРА, ПРИЗ [11].

Контрольные вопросы

1. Дайте определение параллелизма. Какие виды зависимостей существуют между объектами в программе?
2. Что такое информационный граф и ярусно-параллельная форма задачи? Назовите основные виды параллелизма.
3. Объясните сущность векторного параллелизма.
4. В чем состоит параллелизм независимых ветвей?
5. Что такое скалярный параллелизм?
6. Приведите классификацию параллельных ЭВМ по Флинну.
7. Чем отличается управление потоком команд от управления потоком данных?
8. В чем состоит сущность закона Амдала? Что такое ускорение?
9. Назовите характеристики эффективности параллельных ЭВМ и процессоров.

Глава 2

СТРУКТУРЫ ЭВМ С ОДИНОЧНЫМ ПОТОКОМ КОМАНД

§ 2.1. Конвейерные процессоры для скалярной обработки

Одним из самых простых и наиболее распространенных способов повышения быстродействия процессоров является *конвейеризация* процесса вычислений. Большим преимуществом конвейерных ЭВМ перед параллельными ЭВМ других типов является возможность использования пакетов программ, уже написанных для последовательных ЭВМ.

В любом процессоре машинная команда проходит ряд этапов обработки, например: выборку команды из оперативной памяти (ВК), вычисление абсолютного адреса операнда в оперативной памяти (ВА), выборку операнда из памяти (ВО), операцию в АЛУ.

В процессоре последовательной ЭВМ для выполнения этих функций используется единственное устройство, поэтому время выполнения команды

$$t_к = t_{ВК} + t_{ВА} + t_{ВО} + t_{АЛУ}.$$

Чтобы уменьшить $t_к$, можно для каждой функции ввести собственное оборудование (рис. 2.1). В таком процессоре любая команда последовательно проходит через все устройства, находясь на каждом этапе время Δt. Так, команда с номером i поступает в УВК, через время Δt она переходит в УВА, а в УВК поступает команда с номером $i + 1$, затем через время Δt команда i поступает в УВО, $i + 1$ — в УВА, $i + 2$ — в УВК и т. д. Наконец, команда i поступает в АЛУ и через время Δt вырабатывается результат. После этого через время Δt будет получен результат команды $i + 1$. Таким образом, несмотря на то, что общее время выполнения любой команды сохранилось, результаты вырабатываются через время $\Delta t = t_к/n$, где n — число этапов этого конвейера команд.

Описанный принцип построения процессора, действительно, напоминает конвейер сборочного завода, на котором изделие последовательно проходит ряд рабочих мест. На каждом из этих мест над изделием производится новая операция. Эффект ускорения

достигается за счет одновременной обработки ряда изделий на разных рабочих местах.

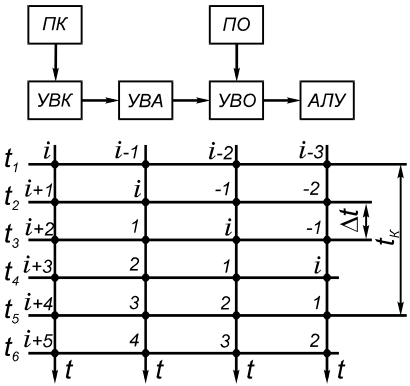


Рис.2.1. Схема и функционирование конвейера команд:
ПК — память команд; ПО — память операндов; УВК, УВА, УВО — устройства выборки команд, вычисление адреса, выборки операндов соответственно

Конвейерные процессоры применяются во всех без исключения старших моделях семейств ЭВМ.

Временная диаграмма на рис. 2.1 строилась при следующих упрощениях: в потоке выбираемых из ПК команд отсутствуют команды условных переходов; все команды имеют одинаковое время нахождения на разных этапах.

Наличие команд условного перехода будет вынуждать переход к командам, которые в данный момент отсутствуют в конвейере, что потребует опустошения и повторного заполнения конвейера из ПК, а неодинаковая длина команд приведет к приостановкам конвейера. Такой в общем случае асинхронный характер функционирования конвейера снижает указанные выше цифры быстродействия.

Стандартный способ увеличения быстродействия конвейерного процессора состоит в следующем: в существующем варианте конвейера выбирается устройство с наибольшим временем срабатывания и разделяется на два или более устройств с меньшим временем срабатывания каждое. При этом цикл конвейера Δt уменьшается. Если и после этого быстродействие конвейера недостаточно, снова выбирается наиболее медленное устройство и процесс повторяется.

В дальнейшем будет рассмотрена конвейеризация устройств процессора в таком порядке: АЛУ, УВК, УВА, УВО.

Арифметический конвейер можно построить для любых арифметико-логических операций: сложения, умножения, логиче-

ских операций. В частности, на рис. 2.2, а, показан конвейер для выполнения операции сложения двух чисел с плавающей запятой. Каждое число представлено в форме $A \cdot R^p$, где A — мантисса; R — основание системы счисления, p — порядок. Конвейер для умножения целых чисел изображен на рис. 2.2, б. Здесь каждым входом сумматора $\sum$ первого каскада управляет один разряд множителя. В зависимости от его значения на вход сумматора $\sum$ подаются два смежных сдвинутых частичных произведения. Число каскадов такого конвейерного умножителя равно $\log_2 r$, где r — разрядность чисел A_i, B_i.

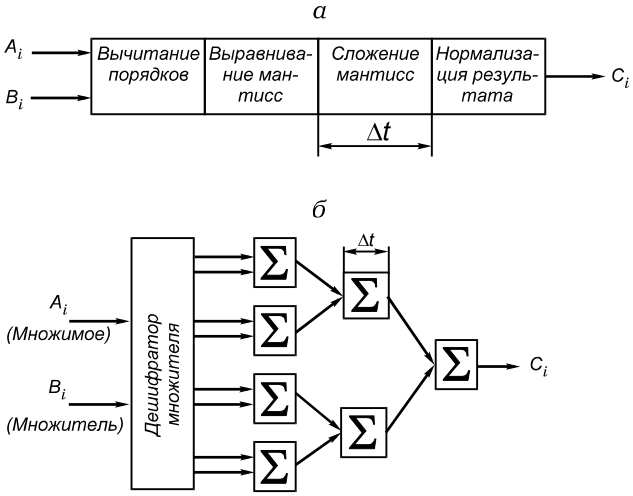


Рис. 2.2 Арифметические конвейеры для сложения (*а*) и умножения (*б*)

Как и в случае конвейера команд, числа поступают на вход конвейеризованного АЛУ вплотную друг за другом, поэтому результаты на выходе получаются с интервалом Δt. Для современных ЭВМ величина Δt стала меньше 10 нс, что соответствует быстродействию больше 100 млн оп/с, и цикл конвейера имеет тенденцию к дальнейшему уменьшению. Но при этом Δt не может стать меньше времени передачи данных с каскада на каскад.

Впервые арифметические конвейеры были использованы для целей обработки числовых векторов в ЭВМ STAR-100, запущенной в США в 1973 г. [5].

Если ставится задача построить быстродействующий конвейерный процессор с $V = 100$ млн оп/с, то цикл всех его устройств не должен превышать $\Delta t = 10$ нс. Выше было показано, как обеспечить данный цикл в АЛУ.

Рассмотрим, как обеспечить такую производительность в УВК. Поскольку на каждый полученный в АЛУ результат приходится одна выборка команды из ПК, то время выборки этой команды не должно превышать 10 нс. Но современные полупроводниковые запоминающие устройства большой емкости имеют цикл обращения $t_{пк} = 100...300$ нс, что во много раз больше требуемого цикла конвейера (10 нс). Выходом здесь является использование множества автономных по функционированию блоков памяти. Число этих блоков $N = t_{пк}/\Delta t$ и может достигать величины 8...64 (обычно кратно степени 2).

Организация работы такой многоблочной памяти может быть различной. Некоторые варианты памяти этого типа изображены на рис. 2.3.

Если память имеет организацию, предназначенную для чтения со сдвигом (рис. 2.3, а), то в регистры адреса (РА) блоков памяти 1...4 с интервалом Δt подается новый адрес из счетчика адресов команд (СчАК) ПК. С таким же сдвигом по времени на выходе ПК будут появляться команды, которые затем поступают в буфер команд (БК), представляющий собой совокупность быстрых регистров. При поступлении каждой новой команды на вход БК содержимое всех его регистров сдвигается вверх на одну позицию, и верхняя команда (самая старая) удаляется из БК.

В УВК имеется СчАК БК, который указывает положение в БК считываемой из УВА команды. При считывании из БК каждой команды его содержимое уменьшается на единицу, при добавлении в БК новой команды из ПК — увеличивается на единицу. За запросом новых команд в БК постоянно следит УВК, которое определяется величиной l. Если l становится меньше заданного уровня, то запускается СчАК ПК и производится выборка из ПК новых команд.

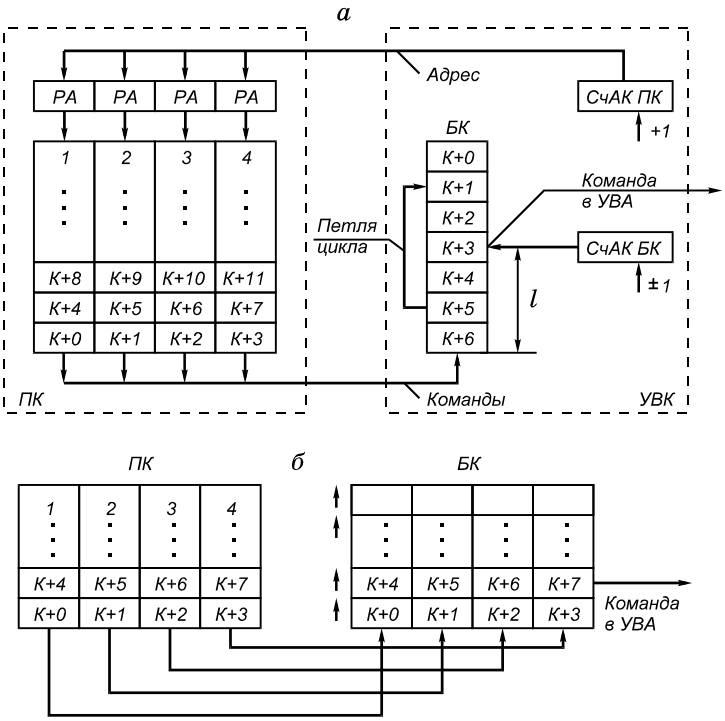


Рис. 2.3 Организация многоблочной памяти для выборки команд:
а — выборка со сдвигом во времени; *б* — выборка широким словом

Во многих задачах линейной алгебры и задачах решения систем уравнений в частных производных общее время выполнения программ определяется скоростью выполнения внутренних циклов, число повторений которых для задач большой размерности велико. Число же команд в петле цикла обычно невелико, и они полностью оказываются в БК. В таком случае выборка команд осуществляется только из БК, а ПК не используется. Это важно в структурах процессоров, где в качестве ПК и ПО используются одни и те же блоки памяти. В подобном случае выборка команд не будет создавать помех выборке операндов.

В схеме на рис. 2.3, б за один цикл памяти в БК заносится несколько команд (“широкое слово”), операции же в БК выполняют-

ся, как и ранее. Схемы на рис. 2.3 не имеют явного предпочтения друг перед другом.

Многоблочная структура памяти была впервые применена в ЭВМ STRETCH [5].

Относительно УВА следует отметить, что оно является простым АЛУ для сложения коротких целых чисел (адресов), поэтому получение малого Δt для этого устройства не составляет труда.

Сократить цикл работы УВО значительно сложнее. Здесь для уменьшения времени чтения операнда необходимо использовать многоблочную память. Однако между выборкой команд и выборкой операндов существует следующее принципиальное различие. Команды в программе и памяти располагаются в порядке линейного нарастания их номеров, поэтому во время исполнения текущей команды всегда можно вычислить адреса и выбрать (исключая команды переходов) любые следующие команды (см. рис. 2.3), что и приводит к уменьшению Δt при выборке команд. Однако строго упорядоченная выборка команд порождает неупорядоченную последовательность адресов для выборки операндов (рис. 2.4). Это означает, что выборка операндов для некоторой команды не может быть произведена заранее, до ее выборки. Следовательно, выборка операндов не может быть конвейеризована, поэтому для построения ПО используется не конвейерный, а поточный принцип организации многоблочной памяти (рис. 2.5).

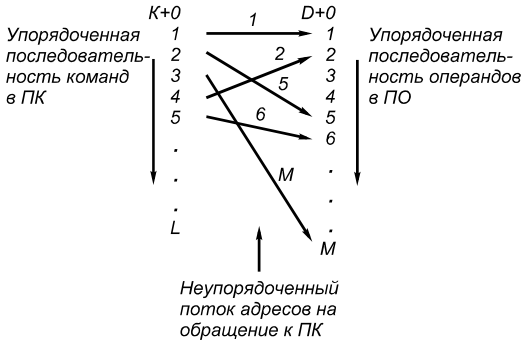


Рис. 2.4. Процесс генерации адресов операндов

Поступающие из УВА адреса операндов распределяются по блокам ПО. Поскольку

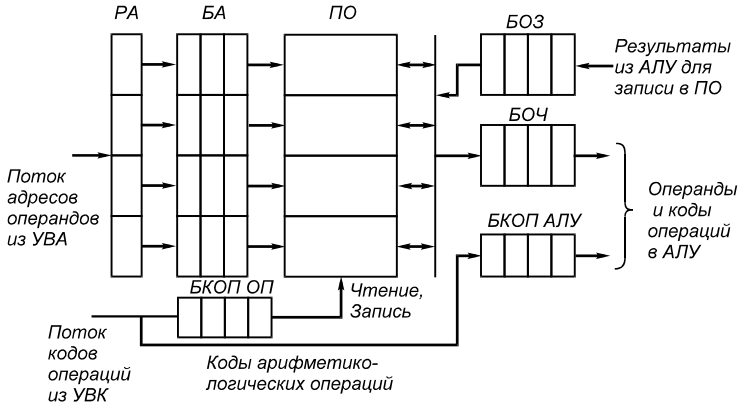


Рис. 2.5. Поточная организация УВО

распределение адресов носит достаточно случайный характер, в отдельных блоках памяти возможны очереди, для размещения которых введены буфера адресов (БА). Выбираемые из памяти операнды должны сразу поступать в АЛУ, однако вследствие неравномерности их появления из ПО и разной длительности исполнения операций в АЛУ вводятся буферные регистры операндов чтения (БОЧ) и записи (БОЗ). С каждой парой операндов связан свой код операции, который хранится также в буфере кода операций (БКОП). Таким образом, в буфере операндов (БО) и БКОП хранятся готовые к исполнению в АЛУ группы информации.

Качественная картина сокращения цикла выборки из многоблочной памяти одного операнда представлена на рис. 2.6. Время цикла одного блока памяти t выражено в относительных единицах.

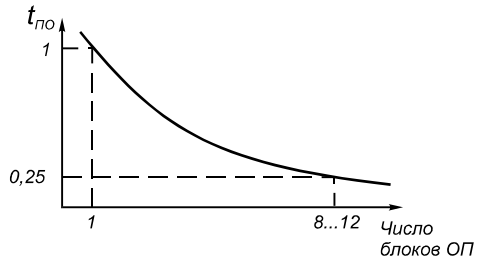


Рис. 2.6. Зависимость среднего времени выборки операнда из ПО от числа блоков ОП

Для уменьшения числа обращений к ПО, как правило, используются *регистры общего назначения* (РОН), расположенные в АЛУ (рис. 2.7). Обычно их 8...16. В этом случае АЛУ и УВО работают только с РОН, в результате чего в системе команд ЭВМ происходит дифференциация. Вместо команды с обращением в память за обоими операндами (рис. 2.8, а) появляется команда обмена между памятью и РОН (рис. 2.8, б) и команда для работы АЛУ с РОН (рис. 2.8, в). Качественная картина снижения числа обращений в ПО при увеличении РОН представлена на рис. 2.9.

Довольно сложную структуру памяти для выборки команд и операндов, содержащую множество блоков памяти, ряд буферов и других регистровых файлов, иногда заменяют единой, достаточно большой *сверхоперативной памятью*, называемой КЭШ-памятью (CACHE) или просто КЭШ. КЭШ-память имеет малый цикл обращения (десятки наносекунд) и

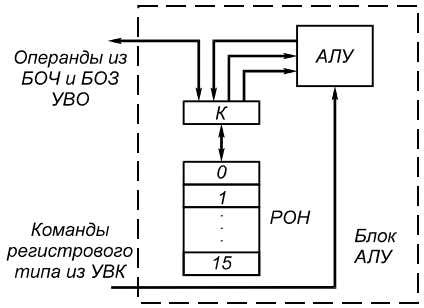


Рис. 2.7. Организация АЛУ с регистровой памятью: *К* — коммутатор

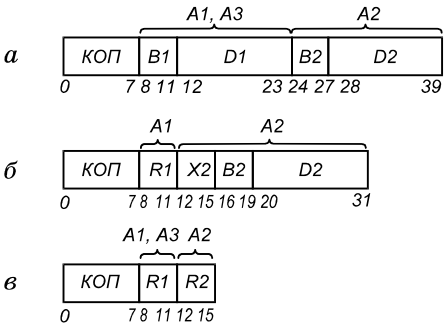


Рис. 2.8. Структура команды конвейерной ЭВМ

а — оба операнда размещены в памяти; *б* — один операнд размещен в памяти, другой в РОН; *в* — оба операнда размещены в РОН; *КОП* — код операции; *A*1, *A*2, *A*3 — абсолютные адреса первого, второго операндов и результата; *R*1, *R*2 — поля для указания номеров РОН, где размещены операнды; *Xi*, *Bi* — поля для указания номеров РОН, где хранятся индексный и базовый адреса; *Di* — смещение

большой (до 16...32 кбайт) объем (рис. 2.10). Основная память *M* разделена на *N* страниц (обычно объемом 1024 байт). КЭШ содержит *n* страниц такого же объема, причем *N*>>*n*. Любая страница КЭШ может заполняться информацией из *M* всего за несколько тактов. В каждой странице КЭШ расположены данные со смежными адресами, разные же страницы КЭШ могут не стыковаться по адресам.

При всяком обращении к *M* из УВК или УВО с помощью ассоциативной памяти производится проверка наличия искомой информации в КЭШ. Если информация имеется, производится обращение к КЭШ. При этом надо различать операции чтения и записи. В случае записи одновременно корректируется соответствующая ячейка памяти в *M*. Если нужный адрес обращения в КЭШ отсутствует, то из *M* в КЭШ производится запись новой страницы.

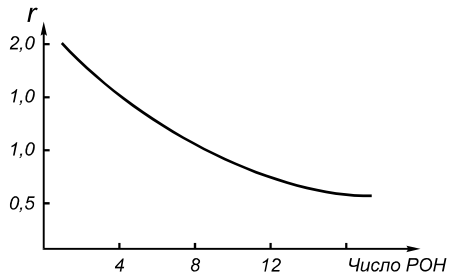


Рис. 2.9. Снижение числа обращения в ПО в зависимости от количества РОН:
r — число обращений за операндами в ПО в пересчете на одну выполненную команду

Рассмотрим два процесса: обращение к КЭШ и смену страницы в КЭШ.

При всяком обращении к *M* с помощью ассоциативной памяти делается проверка наличия данной страницы в КЭШ. Если она есть, то адрес в КЭШ будет таким:

$A=N_3$ (из ассоциативной памяти)$+N_2$ (из команды обращения).

В поле *Z* периодически заносится некоторое число. При каждом обращении к странице КЭШ из занесенного числа вычитается единица. Когда в КЭШ не оказывается нужногослова, требуется заменить страницу. Для этого просматриваются поля *Z*. Там, где число наибольшее, страница считается редко исполь-

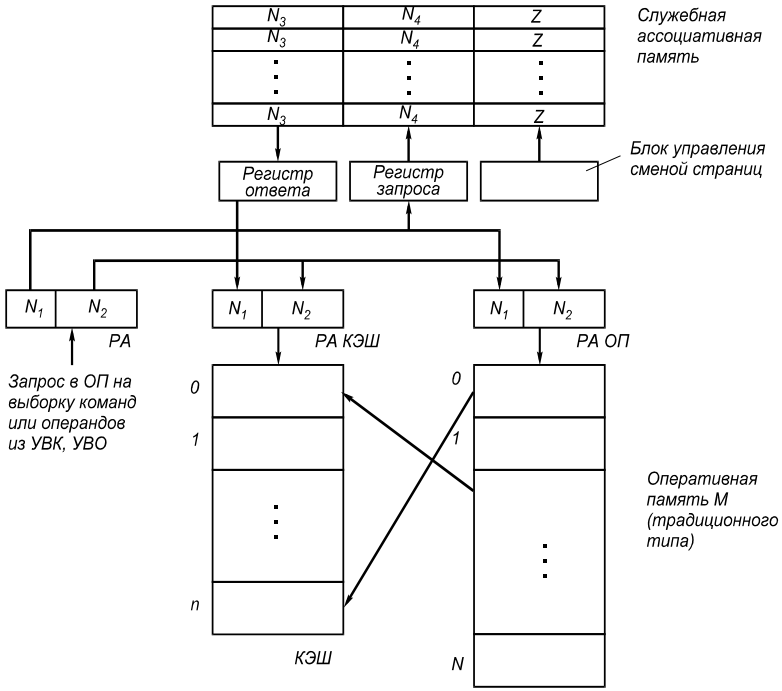


Рис. 2.10. Структура КЭШ-памяти:
N_1 и N_2 — номера страниц и слова, указанные в запросе на обращение к ОП; N_3 — номер страницы КЭШ; N_4 — номер страницы, размещенной в КЭШ

зуемой. Вместо нее из *М* вводится новая страница, и ассоциативная память корректируется. От обычного буфера КЭШ отличается большими размерами, что не позволяет добавлять новую информацию путем сдвига файла регистров, а также наличием несмежных страниц, так как приводит к специальной схеме обмена и обращения, обязательно использующей ассоциативную память для доступа к таблице. Кроме того, эффективность КЭШ зависит от характера распределения данных в *М*. Плохое распределение данных приводит к частой смене страниц.

Итак, мы изучили структуру отдельных устройств конвейерного процессора.

Теперь рассмотрим структуру и функционирование *скалярного конвейерного процессора* в целом (рис. 2.11).

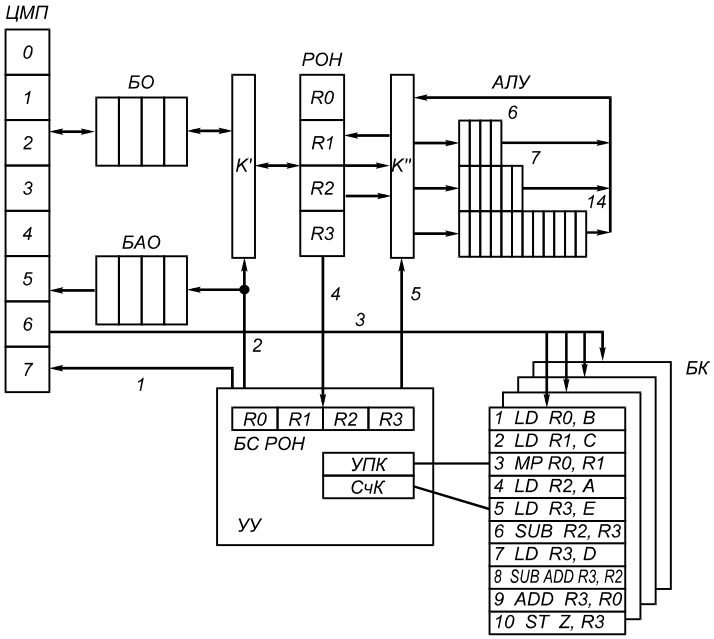


Рис. 2.11. Организация скалярного конвейерного процессора:
ЦМП — центральная многоблочная память; БАО — буфер адресов операндов; АЛУ — конвейеризованные арифметико-логические устройства для сложения, умножения и деления чисел с плавающей запятой; К', К''— коммутаторы памяти и АЛУ соответственно; БС РОН — блок состояния РОН; УПК — указатель номера пропущенной команды; СчК — счетчик команд; 1 — шина адреса команд; 2 — шина управления выполнением команд обращения к памяти; 3 — шина заполнения БК; 4 — шина смены состояний РОН; 5 — шина управления выполнением регистровых команд

Процессор содержит несколько конвейерных АЛУ. Это позволяет одновременно исполнять смежные арифметико-логические операции, что соответствует реализации не только параллелизма служебных операций, но и локального параллелизма. Для разных операций АЛУ имеют различную длину конвейера (на рис. 2.11 она равна 6, 7 и 14 позициям). В процессоре используются команды двух классов: команды обращения в память (см. рис. 2.8, б) и регистровые команды для работы с РОН (см. рис. 2.8, в). Буфер команд имеет многостраничную структуру, что позволяет во время

работы УУ с одной страницей производить заранее смену других страниц.

Для изучения работы процессора (см. рис. 2.11) использован отрезок программы, соответствующий вычислению выражения

$$Z = A - (B * C + D) - E.$$

В программе: *LD* — команда загрузки операнда из памяти в регистр; *MP*, *SUB, ADD* — команды умножения, вычитания и сложения соответственно; *ST* — команда записи операнда из регистра в память.

Состояние некоторых регистров при выполнении программы показано в табл. 2.1. В последней колонке таблицы приведен порядок запуска команд на исполнение. В частности, видно, что некоторые команды могут опережать по запуску команды, находящиеся в программе выше запущенной. Например, команды 4 и 5 выполняются ранее команды 3. Это возможно благодаря наличию в программе локального параллелизма и нескольких АЛУ в структуре процессора. Однако подобные “обгоны” не должны нарушать логики исполнения программы, задаваемой ее ИГ (рис. 2.12). Любая операция согласно рисунку может быть запущена только после того, как подготовлены соответствующие операнды. Это достигается путем запрета доступа в определенные РОН до окончания операции, в которой участвуют данные РОН. Состояния РОН отражены в специальном БС РОН.

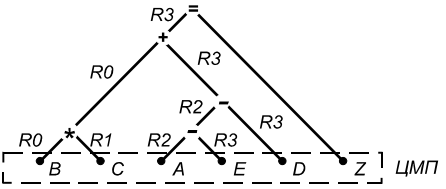


Рис. 2.12. Информационный граф программы:
Ri — регистр общего назначения

В табл. 2.1 приведено также описание нескольких тактов работы процессора. Принято, что выборка операнда из ЦМП занимает четыре такта. Кроме того, считается, что за один такт процессора устройство управления запускает на исполнение одну команду или просматривает в программе до четырех команд.

Таблица 2.1.

Порядок исполнения программы в скалярном конвейере

NN такта	Состояние РОН				Номер команды
	*R*0	*R*1	*R*2	*R*3	
1	4	-	-	-	1
2	3	4	-	-	2
3	2	3	4		4
4	1	2	3	4	5
5	*x*	1	2	3	-
6	7	-	1	2	3
7	6	-	*x*	1	-
8	5	-	6	*x*	6
9	4	-	5	4	7
10	3	-	4	3	-

Сделаем пояснения к таблице.

Такт 1. Анализ БС РОН показывает, что все РОН свободны, поэтому команда 1 запускается для исполнения в ЦМП. В столбец *R*0 записывается 4, что означает: *R*0 будет занят четыре такта. После исполнения каждого такта эта величина уменьшается на единицу. В структуре процессора занятость *R*0 описывается установкой разряда R0 БС РОН в 1, а затем сброс *R*0 в 0 по сигналу с шины 4, который появляется в такте 4 после получения операнда из памяти.

Такт 2. Запускается команда 2 и блокируется регистр *R*1.

Такт 3. Просматривается команда 3, она не может быть выполнена, так как после анализа БС РОН нужные для ее исполнения регистры *R*0 и *R*1 заблокированы. Команда 3 пропускается, а ее номер записывается в УПК. Производится анализ условий запуска следующей (по состоянию СчК) команды. Команда 4 может быть запущена и запускается.

Такт 4. Просмотр БК начинается с номера команды, записанной в УПК. Команда 3 не может быть запущена, поэтому запускается команда 5.

Такт 5. Команда 3 не может быть запущена, так как занят регистр *R*1, однако регистр *R*0 освободился и будет использоваться командой 3, он снова блокируется (символ *x*). Просмотр четырех

следующих команд показывает, что они не могут быть запущены, поэтому в такте 5 для исполнения выбирается новая команда.

Такт 6. Запускается команда 3.

В дальнейшем процесс происходит аналогично. Можно заметить, что за 10 тактов, описанных в табл. 2.1, в процессоре запущено семь команд, что соответствует $10/7 \approx 1{,}5$ такта на команду. Предположим, что такт процессора равен 10 нс. Тогда на выполнение одной команды тратится 15 нс, что соответствует быстродействию $V = 70$ млн оп/с.

§ 2.2. Конвейерные процессоры для векторной обработки

Как и ранее, будем считать, что вектор — это одномерный массив, который образуется из многомерного массива, если один из индексов не фиксирован и пробегает все значения в диапазоне его изменения. В некоторых задачах векторная форма параллелизма представлена естественным образом. В частности, рассмотрим задачу перемножения матриц.

Для перемножения матриц на последовательных ЭВМ неизменно применяется гнездо из трех циклов:

```
    DO 1 I = 1, L
    DO 1 J = 1, L
    DO 1 K = 1, L
1   C(I, J) = C(I, J) + A(I, K) * B(K, J)
```

Внутренний цикл может быть записан в виде отрезка программы на фортраноподобном параллельном языке:

```
R (*) = A (I, *) * B (*, J)                    (2.1)
C (I, J) = SUM R (*)
```

Здесь R(*), A(I, *), D(*, J) — векторы размерности L; первый оператор представляет бинарную операцию над векторами, а второй — унарную операцию SUM суммирования элементов вектора.

Для выполнения операций над векторами также используются арифметические конвейеры. Структура конвейерного процессора для обработки векторов показана на рис. 2.13.

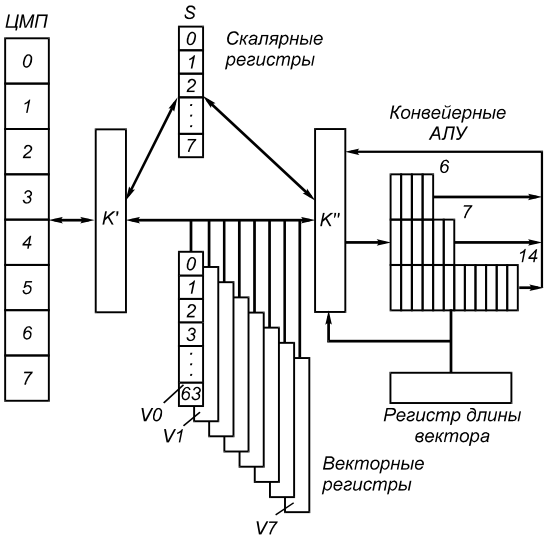


Рис. 2.13 Структура векторного конвейерного процессора

Структура устройства управления этого процессора не рассматривается, так как она аналогична структуре управления скалярного конвейерного процессора (отличие в том, что при выполнении векторной команды код операции команды не меняется).

Главная особенность векторного процессора — наличие ряда *векторных регистров V* (обычно до 8), каждый из которых позволяет хранить вектор длиною до 64 слов. Это своего рода РОН, значительно ускоряющие работу векторного процессора. Назначение остальных узлов такое же, как и узлов, изображенных на рис. 2.11.

Рассмотрим, как будет выполняться программа (2.1) в векторном процессоре. На условном языке ассемблерного уровня программа может быть представлена следующим образом:

```
1  LD L, Vi, A
2  LD L, Vj, B
3  MP Vk, Vi, Vj                                    (2.2)
4  SUM Sn, Vk
```

Операторы 1 и 2 соответствуют загрузке слов из памяти с начальными адресами A и B в регистры Vi, Vj; оператор 3 означает

поэлементное умножение векторов с размещением результата в регистре Vk; оператор 4 — суммирование вектора из Vk с размещением результата в Sn.

Соответственно этой программе векторные регистры сначала потактно заполняются из ЦМП, а затем слова из векторных регистров потактно (одна пара слов за такт) передаются в конвейерные АЛУ, где за каждый такт получается один результат.

Рассмотрим характеристики быстродействия векторного процессора на примере выполнения команды MP Vi, Vj, Vk. Число тактов, необходимое для выполнения команды, равно: $r = m_* + L$, где m_* — длина конвейера умножения.

Поскольку на умножение пары операндов затрачивается $k = (m_* + L)/L$ тактов, то быстродействие такого процессора

$$V = 1/(K\Delta t) = \frac{L}{(m_* + L)\Delta t},$$

где Δt — время одного такта работы конвейера.

Быстродействие конвейера зависит от величины L (рис. 2.14, кривая 1). При $L > m_*$ величина $K = 1$ и $V = 1/\Delta t$. Обычно для векторных процессоров стараются сделать Δt малым, в пределах 10...20 нс, поэтому быстродействие при выполнении векторных операций может достигать 50...10 млн оп/с.

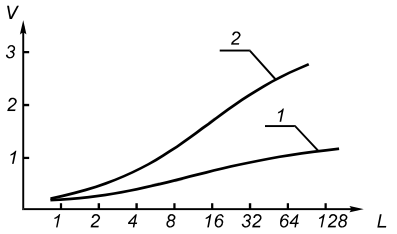


Рис.2.14. Зависимость быстродействия векторного процессора от длины вектора:

$m_{\text{ЦМП}} = 4$; $m_* = 7$; $m_+ = 6$

Важной особенностью векторных конвейерных процессоров, используемой для ускорения вычислений, является механизм зацепления. *Зацепление* — такой способ вычислений, когда одновременно над векторами выполняется несколько операций. В частности, в программе (2.2) можно одновременно производить выборку вектора из ЦМП, умножение векторов, суммирование эле-

ментов вектора. Поэтому программу можно переписать следующим образом:

```
LD L, Vi, A
ЗЦ Sn, Vi, B
```

Здесь команда зацепления (ЗЦ) задает одновременное выполнение операций в соответствии со схемой соединений (рис. 2.15).

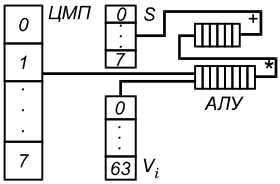


Рис. 2.15. Выполнение операции зацепления в векторном конвейерном процессоре

Для команды ЗЦ получаем:

$$r = m_{ЦМП} + m_* + m_+ + L,$$
$$K = (m_{ЦМП} + m_* + m_+ + L)/L,$$
$$V = n/(K\Delta t),$$

где n — число одновременно выполняемых операций. В случае команды ЗЦ n = 3 (рис. 2.14, кривая 2). При $L >> m_i$ и Δt = 10...20 нс в зацеплении быстродействие равно 150...300 млн оп/с. Такое быстродействие достигается не на всех векторных операциях. Для векторных ЭВМ существуют “неудобные” операции, в которых ход дальнейших вычислений определяется в зависимости от результата каждой очередной элементарной операции над одним или парой операндов. В подобных случаях L приближается к единице. К таким операциям относятся операции рассылки и сбора, которые можно определить следующими отрезками фортран-программ:

1) рассылка

```
    DO 1 I = 1, L
1   X(INDEX (I)) = Y (I)
```

2) сбор

```
    DO 1 I = 1, L
1   Y (I) = X (INDEX (I))
```

Целочисленный массив INDEX содержит адреса операндов, разбросанных произвольным образом в памяти процессоров. Операция рассылки распределяет упорядоченный набор элементов Y(I) по всей памяти в соответствии с комбинацией адресов в массиве INDEX. Операция сбора, наоборот, собирает разбросанные элементы X в упорядоченный массив Y. Названные операции имеются в задачах сортировки, быстрого преобразования Фурье, при обработке графов, представленных в форме списка, и во многих других задачах.

Быстродействие векторного процессора на таких операциях снижается до уровня быстродействия скалярного процессора.

Выше конвейерные ЭВМ были описаны по частям: сначала скалярная часть, затем векторная. Теперь рассмотрим полную структуру векторной ЭВМ на примере машины CRAY [2] (рис. 2.16).

CRAY-1, созданная в 1976 г., была одной из первых ЭВМ, в которой в полной мере проявились все характерные особенности *векторно-конвейерных машин*. Машина CRAY-1 включает стандартные для любой ЭВМ секции: управления памятью и ввода-вывода, регистровую секцию и группу функциональных устройств. Однако состав оборудования каждой секции существенно отличается от аналогичных устройств последовательных ЭВМ.

Память используется как для выборки команд и векторных данных (конвейерный режим), так и для выборки скалярных данных (поточный режим). Для организации поточного режима применяются буферные регистры адреса (*B*0...*B*63) и данных (*T*0...*T*63). Память обладает большой пропускной способностью, однако для разных узлов не всегда используется полная пропускная способность. Максимальная скорость нужна для заполнения буферов команд.

Структура УУ описана в § 2.1. Здесь имеются механизмы предварительного просмотра команд, а также резервировия регистров и функциональных устройств.

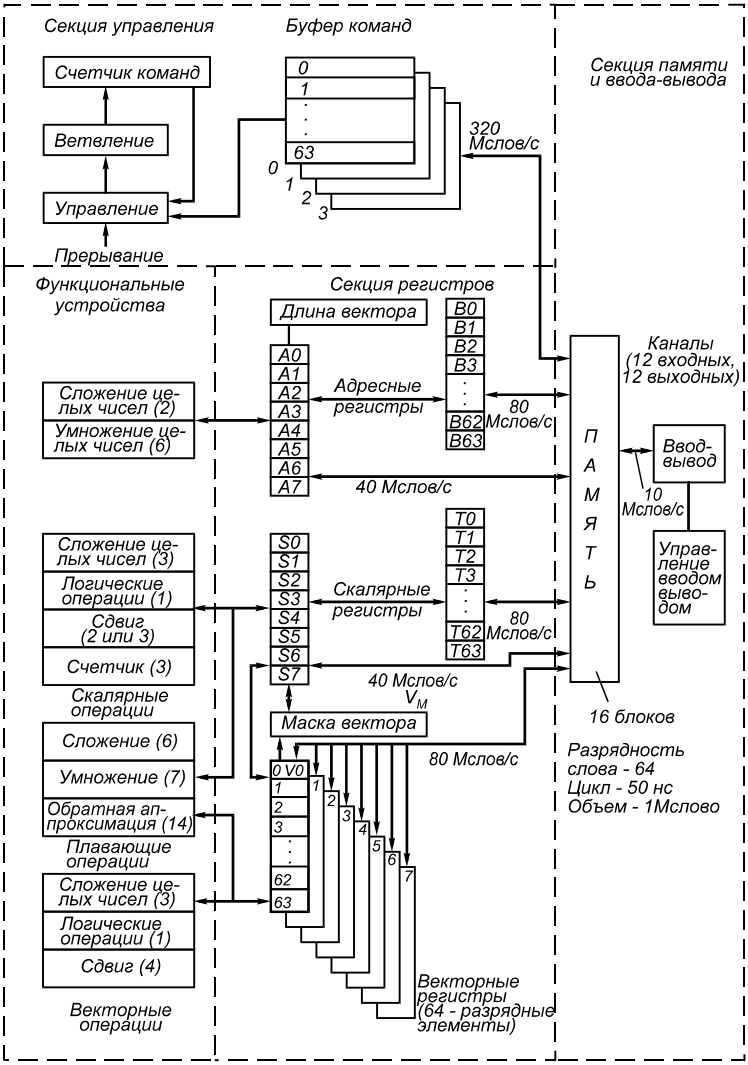


Рис. 2.16. Структура Векторной ЭВМ.
В скобках указана длина конвейеров

Для выполнения адресных операций используются группа адресных регистров *A*0...*A*7 и специальные АЛУ для операций цело-

численной арифметики. Эти АЛУ, как и все другие функциональные устройства, имеют конвейерную структуру.

Скалярные операции выполняются с помощью группы регистров *S*0...*S*7, а векторные — с помощью векторных регистров *V*0...*V*7.

Наибольший интерес представляет состав функциональных устройств. В выполнении скалярных операций участвует семь функциональных устройств: четыре для работы с целочисленной арифметикой и три для работы с числами с плавающей запятой. Устройство “Счетчик” используется для подсчета числа единиц в операнде или числа нулей, предшествующих первой единице операнда.

Поскольку деление плохо поддается конвейеризации, в CRAY-1 оно выполняется в устройствах обратной аппроксимации и умножения посредством итерационной процедуры. Такой подход позволяет использовать конвейерную обработку на операции умножения и зацепление операций.

Векторные операции в CRAY-1 можно разделить на четыре типа. Векторная инструкция первого типа получает операнды из одного или двух регистров *V* и отправляет результат в другой регистр *V*. Последовательные пары операндов передаются из *Vj* и *Vk* в конвейерное АЛУ в каждом такте, и соответствующий результат на выходе АЛУ появляется на *m* тактов позже, где *m* — длина конвейера. Результат отправляется в регистр результата *Vi.* Векторная инструкция второго типа получает по одному операнду из регистров *S* и *V*. Инструкции двух других типов передают данные между памятью и регистрами *V*.

При выдаче векторной инструкции требуемое АЛУ и регистры операндов резервируются на число тактов, определяемое длиной вектора. Последующая векторная инструкция, требующая тех же ресурсов, что и выполняемая, не может выполняться до тех пор, пока ресурсы не будут освобождены. Однако выполнение последующих инструкций, не пересекающихся по ресурсам с неоконченной инструкцией, разрешается.

Две команды машины CRAY-1 требуют специального объяснения. Установка маски 64-разрядного векторного регистра маски (*VM*) соответствует 64 элементам векторного регистра. Если элемент удовлетворяет условию, то соответствующий разряд *VM* устанавливается в 1, в противном случае — в 0. Таким образом ко-

манда, *VM V*5, *Z* устанавливает разряд *VM* в 1, когда элементы *V*5 равны нулю; *VM V*7, *P* устанавливает разряд *VM* в 1, если элементы *V*7 положительны.

По команде слияния векторов содержимое двух векторных регистров *Vi* и *Vk* сливается в один результирующий вектор *Vi* в соответствии с маской регистра *VM*. Если *l*-й разряд *VM* — единица, то *l*-й элемент *Vj* становится *l*-м элементом регистра результатов, в противном случае *l*-й элемент *Vk* становится *l*-м элементом регистра результатов. Значение регистра длины вектора определяет число сливаемых элементов. Таким образом, команда *Vi Vj*!*Vk*&*VM* сливает *Vj* и *Vk* в *Vi* в соответствии с комбинацией в *VM*; *V*7*S*2!*V*6&*VM* сливает *S*2 и *V*6 в *V7* в соответствии с комбинацией в *VM.*

Цель команд маски и слияния — обеспечить условные вычисления с векторными командами. В CRAY-1 соотношение объема памяти хорошо сбалансировано с производительностью системы. По эмпирическому правилу Амдала 1 бит памяти должен приходиться на 1 оп/с. В CRAY-1 при памяти 1 Мслов (64 Мбит) производительность равна 80 Моп/с.

В данной системе впервые применено зацепление операций. За счет этого могут работать два, а иногда три функциональных устройства, доводя производительность системы до 160 и даже 240 миллионов результатов с плавающей запятой в секунду.

§ 2.3. Коммутация в процессорных матрицах

Рассмотрим различные способы выполнения арифметической операции для последовательной, конвейерной и матричной архитектур (рис. 2.17). В качестве примера возьмем задачу сложения двух векторов чисел с плавающей запятой $Z_i = X_i + Y_i$, i =1, 2, ..., n.

Выполнение этой операции для последовательных и конвейерных ЭВМ было объяснено раньше. Как видно из рис. 2.17, в ПМ ускорение достигается не за счет совмещения операций, как в конвейерном ЭВМ, а за счет параллельного во времени исполнения одинаковых этапов команды для разных единиц данных. При этом в общем случае чем больше процессоров в ПМ, тем меньше время выполнения нужной векторной операции.

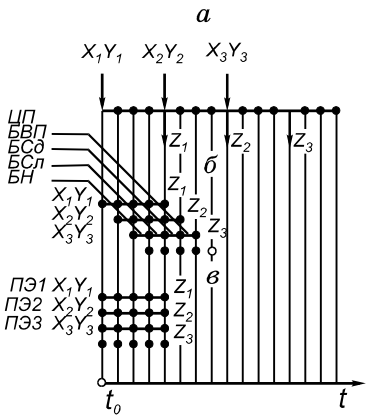


Рис. 2.17. Сравнение способов выполнения векторной операции сложения чисел с плавающей запятой Z(∗)=X(∗)+Y(∗) в последовательной (*а*), конвейерной (*б*) и матричной (*в*) ЭВМ:

а — четыре такта на результат; *б* — один такт на результат; *в* — N процессоров, четыре такта на N результатов; БВП, БСд, БСл, БН — блоки вычитания порядков, сдвига сложения и нормализации соответственно

Таким образом, под матрицей процессоров, или *процессорной матрицей*, будем понимать совокупность идентичных процессорных элементов, объединенных коммутационной сетью. Эта совокупность называется процессорным полем и управляется центральным устройством управления (ЦУУ), которое хранит и выполняет единственную программу, т. е. ПМ функционирует по принципу одиночный поток команд — множественный поток данных (ОКМД).

Процессорный элемент содержит АЛУ, локальную память и собственное УУ. Все ПЭ объединены коммутатором.

Очевидна важность коммутатора для процессорных матриц как с точки зрения затрат на оборудование (до 50% объема оборудования ПП), так и с точки зрения удлинения времени решения задачи за счет потерь времени на коммутацию. Системы коммутации подробно описаны в [2, 8].

Типы перестановок. *Сеть коммутации* представляет собой множество связей между двумя наборами узлов, которые называются входами и выходами. Всего между N входами и M выходами существует N^M различных связей, включая парные связи (“один — одному”), связи “один — многим”, смешанные связи. Сеть, осуществляющая N^M соединений, называется *обобщенной соединительной сетью* (рис. 2.18). Если сеть осуществляет только парные связи, она называется *соединительной*, в ней возможно $N!$ соединений

(рис. 2.18, б, в). В соединительных сетях любой допустимый набор связей (перестановка) выполняется за один такт и без конфликтов.

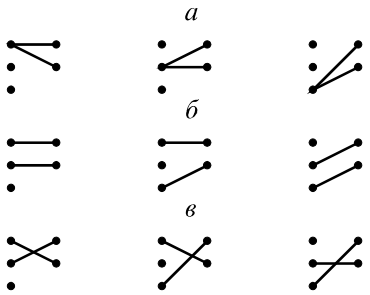


Рис. 2.18. Варианты соединений в обобщенной соединительной сети для N = 3, M = 2:
а — связи “один — многим”; *б*, *в* — парные связи

Для ПМ соединительные сети имеют большее значение, чем обобщенные соединительные сети. Обобщенная соединительная сеть, реализующая N^M соединений, изображается в виде двудольного графа (рис. 2.19, а). Она представляет собой решетку, на пересечениях которой осуществляются необходимые замыкания (рис. 2.19, б). Такая решетка называется координатным переключателем и имеет существенный недостаток: объем оборудования в ней пропорционален $N \times M$, поэтому координатные переключатели используются для ПП размером не более 16...32 процессора.

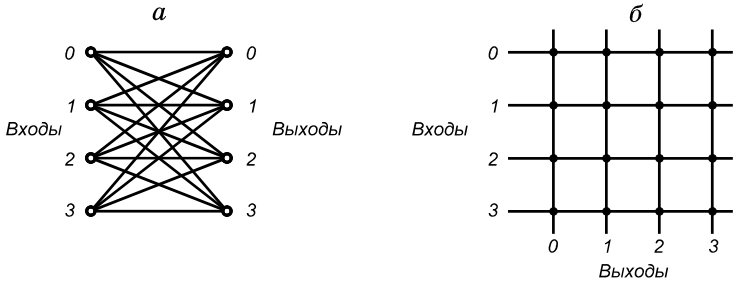


Рис. 2.19. Два представления координатного переключателя размером 4 × 4:
а — в виде двудольного графа; *б* — в виде решетки связей

Коммутатором, противоположным по своим свойствам полному координатному соединителю, является общая шина UNIBUS — общая магистраль. Этот коммутатор (рис. 2.20) способен за один такт выполнить соединение только между одним входом и одним

выходом в отличие от координатного соединителя, который за один такт выполняет все соединения.

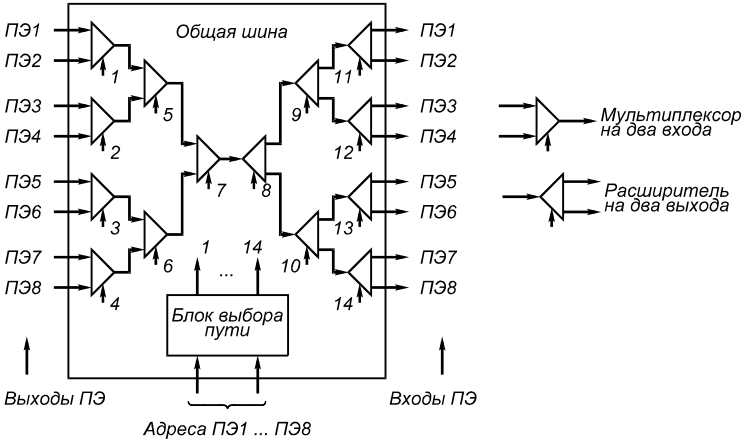


Рис. 2.20. Коммутатор типа “общая шина”

Следует четко разделять два понятия: связь и коммутатор. Связи различного вида возникают между процессорами в ходе выполнения вычислительного алгоритма, а коммутатор (коммутирующая сеть, система коммутации) есть среда, в которой реализуются эти связи. Не всякий коммутатор в состоянии реализовать требуемые виды связей.

Рассмотрим три варианта изображения некоторой многоэлементной связи:

а) с помощью рисунка (рис. 2.21). Это наиболее наглядный вид изображения связей, однако он не может быть использован в программе;

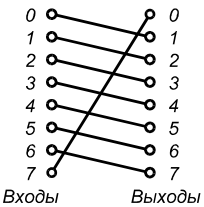


Рис. 2.21. Изображение связей в виде схемы соединений

б) с помощью таблицы (пакета) связей:

Входы	0	1	2	3	4	5	6	7
Выходы	1	2	3	4	5	6	7	0

Это наиболее общий вид изображения связи. Здесь позиция целого числа в таблице означает номер входа, а число в этой позиции — номер выхода, на который замкнута данная связь;

в) с помощью закона, позволяющего на основании номеров (адресов) входов вычислять номера (адреса) выходов.

В зависимости от характера различают регулярные и распределенные случайным образом связи. Для *регулярных связей* таблица связей не нужна, так как все связи вычисляются по определенному, достаточно простому закону. Такие связи называются *перестановками.* Они, как правило, используются для обработки регулярных структур данных при решении систем уравнений, выполнении матричных операций, вычислении быстрого преобразования Фурье и т. д.

Случайно распределенные связи возникают при обработке нечисловой информации (графов, таблиц, текстов) и могут задаваться только в виде таблицы случайных связей.

Рассмотрим некоторые виды перестановок, часто упоминаемые в литературе по параллельным ЭВМ.

1. Перестановки с замещением (рис. 2.22). Закон получения номера выхода следующий:

$$\{b_n, ..., b_k, ..., b_1\}_j = \{ a_n, ..., \overline{a}_k , ..., a_1\}_i,$$

где b_k, a_k — k-е двоичные разряды номера i-го входа и j-го выхода связи; $\overline{a}_k$ — двоичная инверсия k-го разряда.

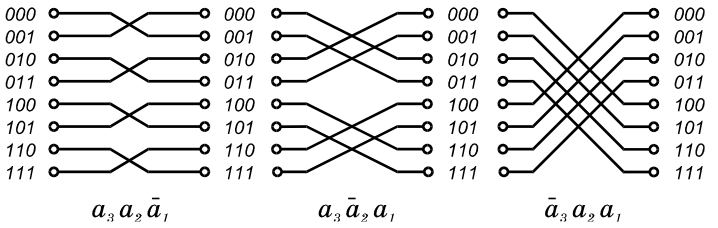


Рис. 2.22. Перестановки с замещением

2. Перестановки с полной тасовкой (рис. 2.23). Они соответствуют циклическому сдвигу влево всех разрядов или группы разря-

дов номера нужного входа. Полученный двоичный результат дает номер соответствующего выхода.

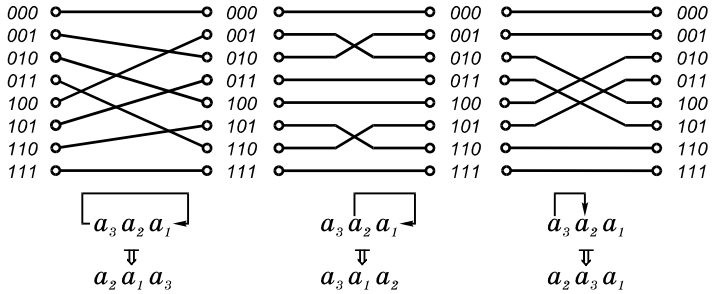


Рис. 2.23. Перестановки с полной тасовкой

3. Перестановки со смещением (рис. 2.24). Они выполняются по формуле

$$i_B = (i_A + r)_N\,,$$

где r — величина сдвига; N — число входов. Вычисления производятся по модулю N. Для случая $i_B = (i_A + 1)_4$ (см. рис. 2.24) в разрядном представлении номеров входов и выходов рассматриваются только два младших разряда.

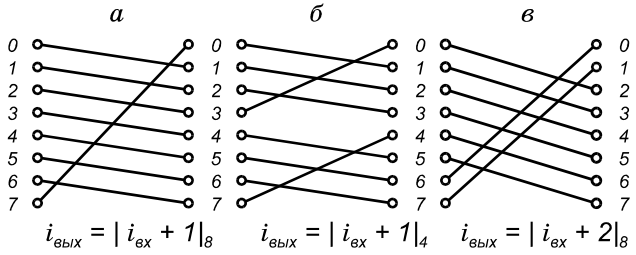


Рис.2.24. Престановки со смещением
величина сдвигов и число входов различны

Если в некотором коммутаторе предусмотрено (конструктивно) использование m перестановок, т. е.

$$КМ = \{П_1, П_2, ..., П_m\},$$

где $П_k$ — перестановка типа k, то для задания кода любой перестановки в КМ для использования необходимо управляющее слово с разрядностью $\log_2 m$.

Как уже говорилось, произвольные парные связи описываются таблицей из N чисел по $\log_2 N$ разрядов на одно число, т. е. требуется всего $N \log_2 N$ бит.

Координатный переключатель в состоянии реализовать за один такт все виды связей как регулярных, так и случайных. Однако объем оборудования такого коммутатора растет слишком быстро.

Поэтому надо искать другие типы коммутаторов, имеющих несколько меньше возможностей по числу устанавливаемых связей, но требующих значительно меньшего объема оборудования. Критерии при поиске таких коммутаторов могут быть разными: минимальное время реализации пакета связей t; минимальный объем оборудования коммутатора H; минимальная величина произведения $t \cdot H$ и др.

Не проводя поиска коммутаторов по указанным критериям, перечислим основные типы коммутаторов и оценим их наиболее важные характеристики.

По структуре (рисунку соединений) коммутаторы можно разделить на две группы: среды и многокаскадные коммутаторы.

Среда — коммутатор, каждый узел которого связан только со своими ближайшими соседями.

Однокаскадные коммутаторы, которые спроектированы для выполнения значительного числа перестановок, имеют большой объем оборудования. Поэтому часто строятся более простые однокаскадные коммутаторы, а перестановки реализуются путем итераций во времени, т.е. за несколько тактов.

В коммутаторе с односторонним сдвигом (рис. 2.25, а) каждый ПЭ связан с двумя соседними. Этот коммутатор строго соответствует перестановке, показанной на рис. 2.24, а. Однако, чтобы выполнить перестановку, как на рис. 2.24, в, требуется уже два такта работы такого коммутатора. В общем случае коммутатор (см. рис. 2.25, а) в состоянии выполнить за N тактов любую перестановку, где N — число ПЭ. При этом каждую единицу передаваемой информации можно снабдить номером процессора прием-

ника. По мере сдвигов, когда информация достигнет соответствующего ПЭ, она должна “осесть” в этом ПЭ.

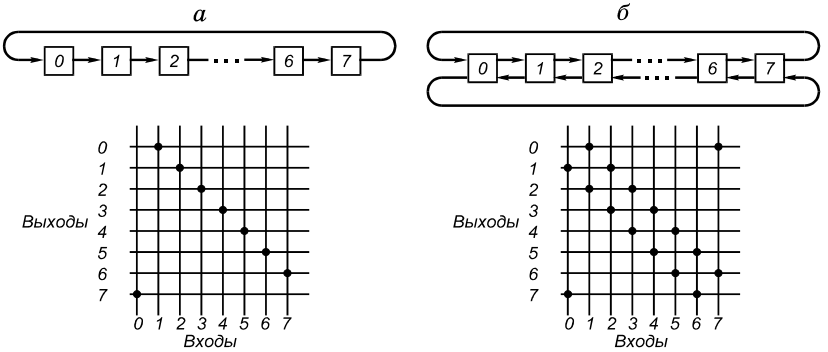


Рис.2.25. Структура кольцевых коммутаторов

Коммутатор с односторонним сдвигом (см. рис. 2.25, а) представлен в сетке координатного переключателя. Точки на пересечениях означают, что все эти соединения могут осуществляться одновременно. Такое представление коммутаторов удобно для теоретических исследований. По данному представлению видно, что коммутатор с односторонним сдвигом — часть полного координатного переключателя и поэтому не является соединительной сетью.

Любая совокупность перестановок в коммутаторе с двусторонним сдвигом (рис. 2.25, б) будет выполнена за более короткое время, чем в коммутаторе на рис. 2.25, а. В частности, чтобы передать единицу информации по связи ПЭ0 → ПЭ7, в первом коммутаторе требуется семь тактов, во втором — один такт.

В двумерном коммутаторе каждый ПЭ соединен двусторонними связями с четырьмя соседними (справа, слева, снизу, сверху) (рис. 2.26, а). Внешние связи могут в зависимости от решаемой задачи программно замыкаться по-разному: вытягиваться в цепочку (рис. 2.26, б), замыкаться в виде вертикально (рис. 2.26, в) и горизонтально (рис. 2.26, г) расположенных цилиндров, в виде тороида (рис. 2.26, д). Коммутационный элемент (КЭ) здесь обычно соединен с оборудованием ПЭ, и объем оборудования всего коммутатора растет пропорционально числу процессоров *N*.

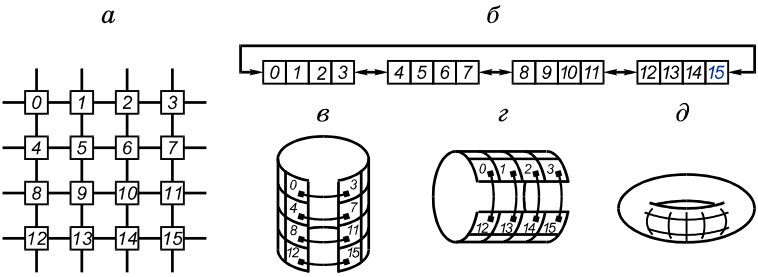


Рис. 2.26. Коммутатор с матричной схемой соединения

Коммутатор с матричной схемой соединений выполняет некоторый набор перестановок быстрее, чем рассмотренные выше коммутаторы с кольцевыми схемами.

Быстродействие любого коммутатора во многом определяется максимальным расстоянием *D,* под которым понимается число промежуточных узлов или тактов передачи информации между самыми удаленными процессорами.

Под кубическим коммутатором понимается пространственная трехмерная структура, где каждый процессор соединен с шестью соседними: четырьмя в своей плоскости и двумя в прилегающих плоскостях.

В общем случае все среды можно назвать *n*-мерными кубами, тогда кольцевой коммутатор имеет размерность $n = 1$, а матрица — $n = 2$ и т. д. Большое распространение в параллельной вычислительной технике получили среды с размерностями 2 и 3.

Чем больше *n* при неизменном числе процессоров, тем больше радиус связей и тем меньше *D.* Под радиусом связи понимается число процессоров, напрямую связанных с данным ПЭ.

Величина *n* может быть и больше трех. Покажем общий прием построения *n*-кубов произвольной размерности (рис. 2.27).

Для примера рассмотрим возможные связи элемента 9 матрицы, изображенной на рис. 2.26, а. Этот элемент в своей строке связан с элементами 8 и 10, адреса которых отличаются

на -1 и +1. Если бы данная строка составляла независимый коммутатор с размерностью 4 (n = 1, N = 4), то связи любого ПЭi такого коммутатора описывались бы рангом 1. Если элемент 9 находится в матрице, то его связи с элементом 5 верхней строки и элементом 13 нижней строки отличаются на $+N$ и $-N$, где N = 4 — размер строки. Следовательно, связи элемента ПЭi матричного коммутатора описываются рангом 2 (см. рис. 2.27).

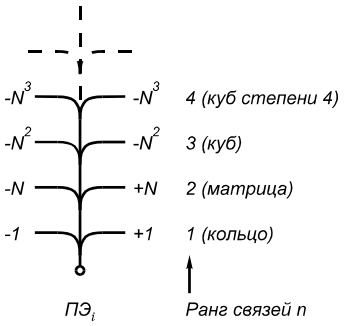


Рис. 2.27. Способ построения n-мерных кубов

Если бы элемент 9 находился в кубе, то номера смежных элементов соседних плоскостей отличались бы на $\pm N^2$, в кубе степени 4 — на $\pm N^3$ и т. д., что и показано на рис. 2.27. Такая методика позволяет построить куб любой степени и вычислить величину D.

Многокаскадные коммутаторы помогают создать более дешевые варианты соединительных и обобщенных соединительных сетей. Координатный переключатель $N\times N$ можно свести к двум ($N/2\times N/2$) переключателям с замещением (рис. 2.28, а) [2]. В свою очередь переключатели $N/2\times N/2$ могут быть разложены на более мелкие подобным же образом. Пример сети, приведенной до уровня стандартных КЭ размерностью 2×2,

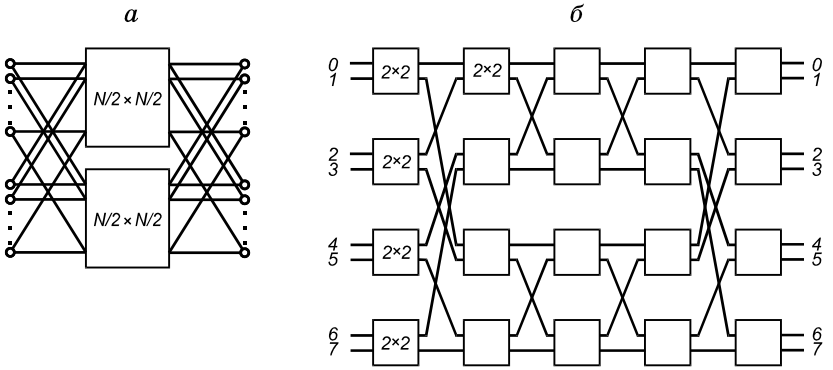


Рис. 2.28. Способ построения сети Бенеша

дан на рис. 2.28, б. По определению, это соединительная сеть (сеть Бенеша), обеспечивающая $N!$ возможных перестановок, каждая из которых выполняется за один такт работы сети. В такой сети отсутствуют конфликты для любых парных соединений.

Стандартный КЭ размерностью 2×2 и возможные в нем соединения показаны на рис. 2.29. Иногда для построения больших сетей применяются элементы 4×4 или 8×8. Основной недостаток сети Бенеша — большое количество элементов, необходимых для ее построения.

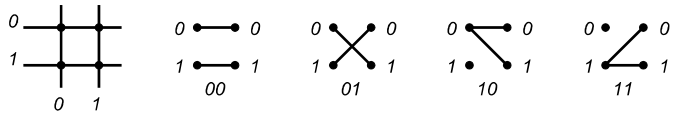


Рис. 2.29. Коммутационный элемент размерностью 2×2
и допустимые перестановки.
Внизу приведены коды нумерации перестановок

Вследствие этого на практике используются коммутаторы со значительно меньшим объемом оборудования, не являющиеся соединительными сетями в полном смысле, однако обеспечивающие широкий класс перестановок. В таких коммутаторах возможны конфликты, которые разрешаются последовательно во времени. Примером конфликта может быть ситуация, когда для двух входов требуется одна и та же выходная линия.

Широко используются неполные соединительные сети: омега-сеть, сеть Бенеша, *R*-сеть и др. Омега-сеть относится к типу сетей тасовки с замещением.

Коммутационные сети могут работать в двух режимах: коммутации каналов и коммутации сообщений. В первом случае сначала с помощью адресной системы устанавливается прямой тракт между парой соединяемых точек, а затем по этому тракту передается информация. Такой вид связи используется, например, в телефонных сетях. Во втором случае каждая передаваемая единица информации снабжается адресом требуемого выхода и перемещается от узла n_i к узлу m_{i+1}, где i — номер каскада; n и m — номера коммутационных узлов в каскадах. После того как информация принята узлом m_{i+1}, линия, соединявшая n_i и m_{i+1}, освобождается

для других соединений. Узел m_{i+1} по адресной части сообщения определяет дальнейший маршрут сообщения.

В матрицах процессоров используется как коммутация каналов, так и коммутация сообщений. Оценим основные характеристики коммутаторов, описанных выше. Оценка будет производиться для случая коммутации сообщений и перестановок двух видов: регулярных и случайных. Существует множество вариантов регулярных перестановок, и довольно трудно аналитически или методом имитационного моделирования определить число тактов, необходимое для реализации некоторой тестовой группы перестановок. Поэтому в качестве теста принята единственная, но очень широко используемая в процессорных матрицах операция — сдвиг на некоторое число разрядов. Минимальная дистанция сдвига равна единице, а максимальная совпадает с величиной максимального межпроцессорного расстояния *D*, в связи с чем за среднюю величину сдвига принято *D*/2 (см. табл. 2.2). Конечно, это только качественная оценка скорости выполнения регулярных операций коммутации.

Таблица 2.2.

Основные характеристики коммутаторов различных типов

Тип коммутатора	Время реализации пакета, такты		Число КЭ
	Регулярные перестановки	Случайный пакет	
Общая шина	$N/2$	N	$\frac{N}{2}\log_2 N$
Кольцо (ранг 1)	$N/4$	$N/2$	N
Матрица (ранг 2)	$\sqrt{N}/2$	$\sqrt{N}$	N
Куб (ранг 3)	$3\sqrt[3]{N}/4$	$\sqrt[3]{N}$	N
Омега-сеть	1	$\log_2 N$	$\frac{N}{2}\log_2 N$
Полный координатный переключатель	1	1	N^2

Для оценки времени реализации пакетов случайных связей обычно проводят имитационное моделирование работы коммутатора при подаче на вход наборов, распределенных по равномерному закону. При этом конфликты разрешаются последовательно в соответствии с некоторым приоритетом. Результаты такого моделирования и число КЭ в коммутаторе каждого типа приведены в табл. 2.2.

Сделаем следующие выводы:

1) коммутаторы типа полного координатного переключателя не могут использоваться для больших N из-за значительного объема оборудования, а общая шина — из-за большого времени реализации сообщения;

2) порядок быстродействия сред на регулярных и нерегулярных пакетах связей одинаков и равен $\sqrt[n]{N}$, где n — ранг среды, а для каскадных коммутаторов подчиняется логарифмическому закону;

3) омега-сеть превосходит в большинстве случаев среды на регулярных пакетах. Для качественного сравнения сред и многокаскадных коммутаторов построен график (рис. 2.30), из которого следует, что выбор того или иного типа коммутатора должен производиться строго в зависимости от типа ЭВМ, класса решаемых задач (виды перестановок) и размера ПП.

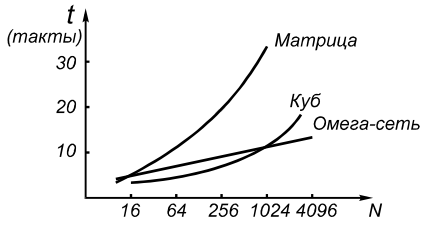


Рис. 2.30. Время реализации случайного пакета связей для различных комутаторов

Рассмотрим некоторые вопросы управления в коммутационных сетях. Для примера разберем управление в омега-сети для двух случаев: реализации регулярных перестановок и пакетов случайных связей.

Коммутатор для реализации регулярных перестановок (рис. 2.31, а) состоит из коммутационного поля и блока дешифрации и управления (ДУ). Выходы блока ДУ с номерами 0...11 подключены к соответствующим адресным входам коммутационных элементов (чтобы не загромождать рисунок, эти линии не показаны).

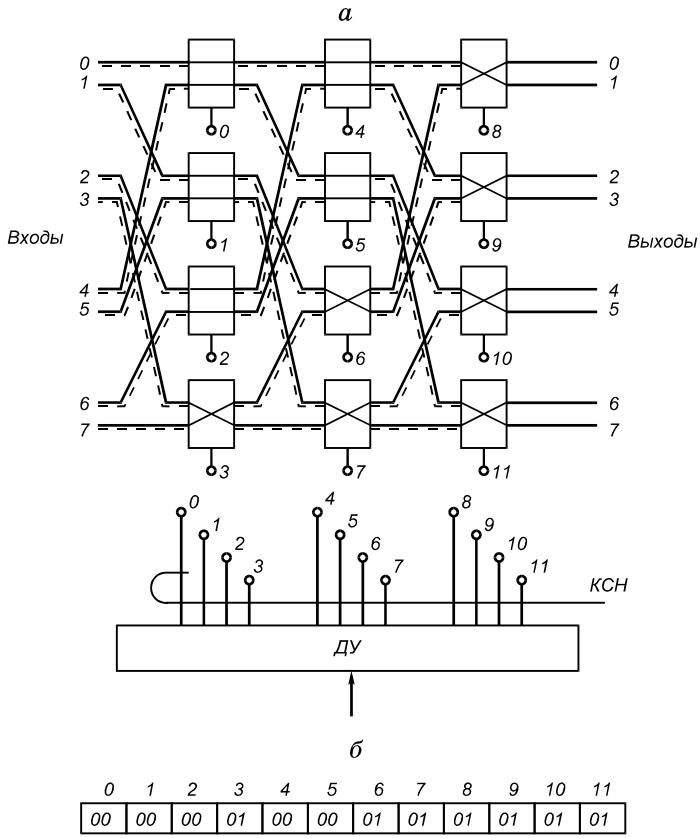


Рис. 2.31. Структура коммутатора для выполнения регулярных перестановок:
а — схема коммутатора; *б* — кодовое слово настройки, использованной в примере перестановки

При выполнении команды коммутации код коммутации подается на вход блока ДУ. Размерность кодового номера перестановки невелика, зависит от числа реализуемых перестановок m и равна $\log_2 m$. Блок ДУ на основе кодового номера перестановки вырабатывает кодовое слово настройки (КСН), размер которого равен $(N / p)C_p^p \log_p N$, где N — число процессоров; p — размерность КЭ; N/p — число КЭ в одном каскаде; C_p^p — число перестановок в

КЭ размерностью $p \times p$; $\log_p N$ — число каскадов. Например, КСН для коммутатора на 128 ПЭ при $p = 2$ содержит 1792 разряда, что сильно затрудняет создание коммутаторов с большой размерностью. Кодовые слова настройки могут храниться в памяти блока ДУ либо вычисляться при выполнении команды коммутации. В первом случае требуется большой объем памяти, во втором время вычисления будет слишком велико. Все КЭ (см. рис. 2.31, а) замкнуты в соответствии с КСН (см. рис. 2.31, б), которое обеспечивает смещение данных на один ПЭ (см. рис. 2.24).

После настройки коммутационного поля с помощью блока ДУ производится передача данных за один такт. Такой вид функционирования соответствует коммутации каналов.

Если при реализации некоторой перестановки блок ДУ обнаруживает, что в некоторых КЭ возможны конфликты, блок ДУ вырабатывает два или более КСН, которые реализуются в коммутационном поле последовательно.

Как правило, вид перестановок заранее не известен и генерируется в процессе решения задачи. Так происходит, например, при определении кратчайшего пути в графе, если каждая вершина графа расположена в отдельном ПЭ и представлена в виде списка вершин. Тогда выборка очередного слоя из списка связей (по одной из каждого процессора) приводит к образованию на входе коммутатора случайно распределенного пакета связей.

В этом случае каждая единица информации, передаваемая любым процессором, состоит из двух частей: А, И. Здесь А — адрес (номер) ПЭ, куда надо передать информацию И. Адрес может передаваться по отдельным шинам (штриховые линии на рис. 2.31, а) или по одним и тем же шинам с разделением во времени: сначала адрес, затем информация. В каждом ПЭ принятый адрес анализируется, чтобы определить на какой выход КЭ должна быть дальше передана информация. Таким образом, в КЭ дешифратор должен дополнительно выполнять функцию анализа принятого адреса. Кроме того, для случайных пакетов конфликты являются обычной ситуацией, поэтому коммутационный элемент должен иметь память для хранения принятой информации, так как неизвестно, будет принятая информация передана в настоящем такте или послана с задержкой. В таком коммутаторе некоторые связи из случайного пакета могут быть реализованы раньше, чем другие,

хотя процесс коммутации на входе был начат для всех связей одновременно.

Описанный способ функционирования соответствует коммутации сообщений. Здесь управление является децентрализованным и распределено по коммутационным элементам.

§ 2.4. Процессорные матрицы

Понятие “процессорная матрица” подчеркивает тот факт, что параллельные операции выполняются группой одинаковых ПЭ, объединенных коммутационной сетью и управляемых единым ЦУУ, реализующим единственную программу. Следовательно, ПМ являются представителями класса ОКМД-ЭВМ. Процессорная матрица может быть присоединена в качестве сопроцессора к БМ. В данном случае БМ обеспечивает ввод-вывод данных, управление графиком работы ПМ, трансляцию, редактирование программ и некоторые другие вспомогательные операции. В отдельных случаях все эти функции выполняет ЦУУ. Процессорные матрицы описаны в [2, 7].

Каждый ПЭ может иметь или не иметь собственную (локальную) память. Известно, что более половины обращений за информацией приходится на долю локальной памяти, поэтому в таких системах к быстродействию коммутатора предъявляются относительно низкие требования. Схема ПМ с локальной памятью приведена на рис. 2.32. Центральное устройство управления является полноценным процессором для выполнения единственной программы, однако в этой программе используются команды двух типов: скалярные и векторные. Скалярные команды начинаются и заканчиваются в ЦУУ, векторные команды начинаются в ЦУУ, а продолжаются в ПП.

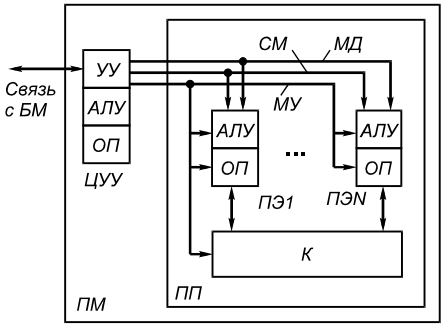


Рис. 2.32 Схема ПМ

Процессорное поле содержит ряд ПЭ, каждый из которых является частью полного процессора, т. е. содержит АЛУ и память, но функции УУ в ПЭ весьма ограничены. В большинстве случаев управление заключается в том, что отдельный ПЭ может выключаться в связи с требованиями решаемой задачи на время выполнения одной или нескольких команд. Число ПЭ колеблется от нескольких десятков (в этом случае процессоры делаются мощными) до нескольких сотен и даже тысяч штук (в этом случае из-за ограничений объема оборудования можно использовать только маломощные в вычислительном отношении процессоры).

Центральное устройство управления связано с ПП тремя магистралями:

1) по магистрали данных МД из ЦУУ в ПП передаются общие адреса и данные, а из отдельных ПЭ в ЦУУ проводится выборка необходимых единиц информации;

2) по системной магистрали СМ, которая имеет одну шину на каждый ПЭ, в ПП передается (или из ПП считывается) вектор активности процессорных элементов;

3) по магистрали управления МУ из ЦУУ всем ПЭ передаются одни и те же микрокоманды для одновременного исполнения.

Как в ЦУУ, так и в ПЭ имеются регистры РОН, поэтому форматы и типы скалярных и векторных команд такие же, как и в последовательных ЭВМ: регистр — регистр (RR), регистр — память (RX) и др.

Векторные команды можно разделить на пять основных групп: арифметико-логические команды; команды пересылки между ЦУУ и ПП; команды коммутации; команды групповых условных переходов; команды активации процессорных элементов.

Рассмотрим функционирование ПП при выполнении команд этих групп.

Пусть имеется следующий фрагмент программы:

```
1  S  ЧТ      5, A1
2  V  ПС      5, 3
3  V  ЧТ      2, A2
4  V  УМН     2, 3
```

Здесь S и V — скалярная и векторная команды соответственно.

По команде 1 в ЦУУ из локальной ОП (адрес А1) считывается константа Z1 и записывается в регистр 5. По команде 2 константа *Z*1 из регистра 5 ЦУУ передается в регистр 3 всех ПЭ по МД. По команде 3 все ПЭ считывают из локальной памяти (адрес А 2) константу Z 2 и помещают в регистр 2. Наконец, по команде 4 все ПЭ умножают содержимое регистров 2 и 3 и результат помещают в регистр 2, т.е. выполняется операция умножения на константу, которую на фортраноподобном векторном языке можно записать так:

Z (∗) = Z1 ∗ Z2 (∗)

Рассмотрим подробнее выполнение векторной команды 3 на микропрограммном уровне (табл. 2.3).

Таблица 2.3.

Выполнение векторной команды обращения
к локальной памяти по тактам

Наименование микрооперации	Место выполнения микро-операции	Используемые	
		магистра-ли	блоки ПЭ*i*
Считывание команды из ОП	ЦУУ	-	-
Дешифрация полей команды в УУ ЦУУ	ЦУУ	-	-
Модификация адреса памяти	ЦУУ	-	-
Установка ОП*i* на прием адреса	ПП	МУ	ОП*i*
Прием адреса А1 из ЦУУ в ОП*i*	ПП	МУ, МД	ОП*i*
Запуск ОП*i* на чтение	ПП	МУ	ОП*i*
Установка АЛУ*i* на прием в регистр 2	ПП	МУ	АЛУ*i*
Операция в АЛУ*i*	ПП	МУ	АЛУ*i*
Возврат к чтению команды из ОП	ЦУУ	-	-

Из таблицы следует, что все микрокоманды генерируются в УУ ЦУУ, однако для команды векторного типа некоторые микрооперации выполняются не в ЦУУ, а в ПП (микрокоманды 4...8). При этом управление производится выборочно различными блоками ПЭ*i* и (как будет далее показано) коммутатором. В ПП все

ПЭ*i* выполняют одну и ту же микрооперацию. Рассмотрение этой микропрограммы позволяет оценить разрядность МУ и МД.

Для большинства ЭВМ магистраль МД имеет 32...64 разряда, а магистраль МУ — 30...80 разрядов; разрядность магистрали СМ равна числу ПЭ.

Как и для конвейерных ЭВМ, в ЦУУ для повышения быстродействия обычно применяется совмещение операций. Например, при выполнении микрокоманд 4...8 для команды *Kj* производится выполнение микрокоманд 1...3 для следующей команды *K* (*j* + 1).

Совмещение может быть применено и в процессорных элементах.

В ПМ *индивидуальное управление* каждым ПЭ зависит от выключения этих ПВ на период выполнения одной (нескольких) задачи или команды в зависимости от исходных данных, внутреннего состояния ПЭ и некоторых внешних условий. Таким включением и выключением занимается двухуровневая система активации.

Первый уровень активации — реконфигурация — позволяет отключать ненужные или неисправные ПЭ на период выполнения нескольких задач и дает возможность проводить вычисления при наличии неисправных ПЭ. Триггер конфигурации имеется в каждом ПЭ. После выполнения тестовых программ процессорного поля в триггеры конфигурации исправных ПЭ из ЦУУ по СМ заносятся единицы, а в триггеры неисправных ПЭ — нули.

Процессорные элементы с нулевым значением триггера конфигурации не участвуют в вычислениях: в них не выполняются команды, поступающие из ЦУУ, становится невозможным считывание и запись в ОП таких ПЭ, состояние их триггеров активации не влияет на команды векторного условного перехода.

Кроме триггера конфигурации в каждом ПЭ есть несколько (до 16) триггеров активации. Триггер основной активности А0 управляет активностью ПЭ: если А0 находится в состоянии 1, то данный ПЭ активен, если в состоянии 0 — пассивен. Вспомогательные триггеры А1...А7 запоминают активность ПЭ перед различными внутренними циклами, если эта активность меняется; после выхода из цикла нужно восстановить исходное значение активности. Триггеры А8...А15 отражают результат вычислений, проводимых в ПЭ: А8 — равенство результата нулю; А9 — знак результата; А10 — перенос; А11 — арифметическое переполне-

ние; А12 — исчезновение порядка; А13 — переполнение порядка; А14 — некорректное деление; А15 — служебный триггер.

Активация бывает внешней и внутренней. В первом случае состояние триггеров активности каждого ПЭ устанавливается со стороны ЦУУ по СМ с помощью команд вида V ВША Ri, Aj. Здесь код ВША означает, что выполняется команда внешней активации, по которой содержимое 16 разрядов регистра Ri ЦУУ поразрядно записывается во всех ПЭ в регистры активации с номером j. Если j = 0, то это означает, что производится загрузка триггеров основной активности, которая может привести к выключению некоторых ПЭ, когда в соответствующих разрядах находятся нули.

Команда V ЧТА Ri, Aj выполняет обратную операцию: считывание состояния триггеров активации в разряды регистра Ri ЦУУ.

Коммутатор в ПП (см. рис. 2.32) может управляться командами двух типов:

V КМ1 Ri, Rj, Rk, F
V КМ2 Ri, Пk, F (2.3)

Команда коммутации КМ1 передает содержимое регистров Ri всех ПЭ в регистры Rk других ПЭ в соответствии с вектором адресов (номеров ПЭ-приемников), каждый элемент которого расположен в регистре Rj данного ПЭ. При выполнении команды содержимое Ri и Rj передается в коммутатор, прокладывающий по адресу из Rj нужный маршрут и затем передающий информацию из Ri.

Детальный алгоритм выполнения команды коммутации зависит от типа используемого коммутатора.

Функция F в команде КМ1 указывает на разновидность команды: запись или считывание данных. В случае считывания требуется обеспечить еще и обратную передачу информации.

Здесь описана только передача информации между регистрами. Если требуется произвести обмен данными, хранящимися в локальной памяти, то предварительно необходимые данные должны быть с помощью команд обращения в память приняты в нужные регистры.

Как упоминалось в § 2.3, команды коммутации с адресным вектором используются там, где заранее не известен вид переста-

новки, т. е. при обработке сложных структур данных (графов, деревьев, таблиц и т. д.).

Однако не всякий коммутатор пригоден для эффективной реализации команд типа КМ1. Такой коммутатор должен допускать работу в асинхронном режиме, внутренними средствами разрешать конфликты, если они возникают. Время передачи пакета случайных связей должно находиться в допустимых пределах. Назовем подобный коммутатор универсальным (УК). Для небольших размеров ПП ($N = 16...32$) в качестве УК может использоваться полный координатный переключатель, а для $N > 32$ должны использоваться, как было показано в § 2.3, многокаскадные коммутаторы.

В УК изменение конфигурации путем разделения ПП на независимые части или по причине неисправностей процессоров не вызывает трудностей, так как для продолжения решения задачи необходимо только переустановить адресные таблицы. Это обеспечивает повышенную живучесть ПП. В машинах с реконфигурацией число ПЭ задается как параметр. Это означает, что в библиотеках стандартных программ число процессоров не определено и конкретизируется в процессе трансляции пользователем или ОС. В синхронных же сдвиговых коммутаторах типа двумерной среды в ILLIAC-IV выход из строя одного узла разрушает всю систему сдвигов.

При выполнении команд типа КМ1 возможна ситуация, когда несколько ПЭ обращаются к одному, т.е. производится сбор информации (коллекция). В коллекционных командах коммутатор должен самостоятельно обеспечить последовательную запись или выдачу требуемых данных.

Команда типа КМ2 означает, что на коммутатор из ЦУУ через МД передается код перестановки Пk, в соответствии с которым настраивается коммутатор и производится бесконфликтная передача информации из регистров Ri или считывание через коммутатор в регистры Ri. Команда КМ2 может выполняться как на универсальных, так и на синхронных коммутаторах для всех видов регулярных перестановок (см. § 2.3).

Рассмотрим вопросы, связанные с эффективностью вычислений и методами программирования для ПМ, на примерах программирования некоторых операций.

Эффективность вычислений в первую очередь зависит от затрат времени на коммутацию и выборку информации. Время выполнения команды коммутации сравнимо со временем выполнения простых команд (логические команды, целочисленное сложение и т.д.), поэтому в большинстве случаев не коммутация, а *размещение данных* влияет на время выполнения программы.

Рассмотрим два варианта размещения квадратной матрицы в ПП (рис. 2.33). Предположим, что для решения некоторой задачи, например, умножения матриц, требуется параллельная выборка как строк, так и столбцов. Стандартное, принятое для большинства языков и ЭВМ размещение “по столбцам” (см. рис. 2.33, а) позволяет выбирать в АЛУ из ОП параллельно только строки, а элементы столбца расположены в одной ОП*i* и могут выбираться только последовательно. В результате при таком размещении параллелизм процессорного поля не будет использован в полной мере, поэтому для подобных задач характерно специальное размещение информации. При “скошенном” размещении матрицы (см. рис. 2. 33, б) за один такт могут быть выбраны как строка, так и столбец. Но при выборе столбца должна использоваться “фигурная” выборка, когда каждый ПЭi обращается в ОПi по собственному адресу, вычисляемому в зависимости от номера ПЭ.

Размещение информации влияет и на методы программирования. Рассмотрим операцию сложения двух векторов (рис. 2.34) для $L = N$ и $L > N$, где L — длина вектора; N — число процессоров. Программы будут выглядеть различно:

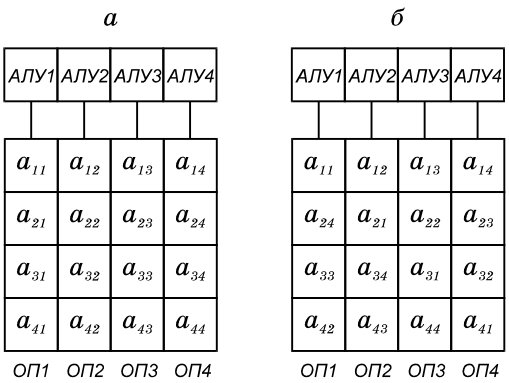


Рис. 2.33. Размещение матрицы в памяти процессорного поля

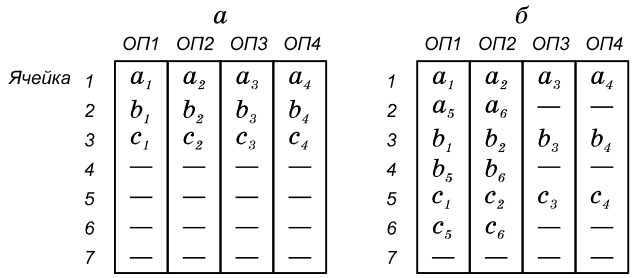


Рис. 2.34. Многослойное размещение коротких (а) и длинных (б) векторов

а) V ЧТ 1, $Я_1$ (считывание среза ячеек $Я_1$ памяти в регистры 1)

V СЛ 1, $Я_2$ (сложение содержимого регистров 1 со срезом ячеек $Я_2$ памяти)

V ЗП 1, $Я_2$ (запись содержимого регистров 1 в срез ячеек $Я_3$ памяти)

б) V ЧТ 1, $Я_1$

V СЛ 1, $Я_3$

V ЗП 1, $Я_5$

V ВША 5, А0 (из регистра 5 ЦУУ передан код активации 1100, что запрещает обращение к ОП3, ОП4)

V ЧТ 1, $Я_2$

V СЛ 1, $Я_4$

V ЗП 1, $Я_6$

Таким образом, несоответствие размеров L и N меняет не только длину, но и состав операторов программы. На программах на параллельных ЯВУ это не отражается: обе программы будут представлены в виде одного и того же оператора

C (∗) = A (∗) + B (∗)

Поскольку наиболее эффективные программы могут быть написаны только на ассемблере, то в языках подобного уровня следует предусматривать средства, позволяющие вне зависимости от соотношения L и N получать одинаковые программы. Так как это не всегда возможно, то из-за необходимости учитывать распо-

ложение данных программирование на ассемблере для ПМ становится намного сложнее, чем для последовательных ЭВМ.

Некоторые операции в ПМ программируются проще и выполняются быстрее, чем в параллельных ЭВМ других типов. Например, операция сборки

V = V0 (INDEX (i)), i = 1, 2, ..., n

состоит в том, что из вектора V0 выбираются числа в порядке, указанном в целочисленном массиве INDEX(i) и помещаются в вектор V1. В конвейерной ЭВМ (см. § 2.2) данная операция выполняется в скалярной части (и поэтому медленно). В ПМ операция сборки отображается в одну команду коммутации согласно (2.3):

КМ1 Ri, Rj, Rk, F

Здесь в регистре Ri хранится вектор V0, в регистре Rj — массив INDEX(i), в регистр Rk будет записан вектор V1. Поскольку все операции в команде КМ1 выполняются параллельно, то время выполнения этой команды в простейшем случае ($L = N$) будет соответствовать одному циклу работы коммутатора.

Для многих параллельных ЭВМ принципиальное значение имеет команда SUM V, т. е. операция суммирования элементов некоторого вектора V. В процессорных матрицах с УК она выполняется в течение нескольких микротактов, которые строятся на основе команды КМ1. В каждом микротакте реализуется следующая операция (работают все ПЭi):

V СЛ Ri, Rj

Здесь операция СЛ выполняется в каждом ПЭ*l* и производит сложение двух чисел, одно из которых расположено в регистре Ri данного ПЭk, а другое — в регистре Ri, но в ПЭ*l*. В регистре *R*j процессора ПЭk указан номер процессора ПЭl. Если в регистре Rj процессора ПЭk записан нуль, то этот процессор не производит суммирование, а выборка операнда из него через коммутатор не запрещается.

Операция SUM V для $N = 8$ (рис. 2.35) выполняется за три такта ($\log_2 8 = 3$) и для каждого такта по вертикали указано содержимое адресного регистра Rj для любого ПЭ.

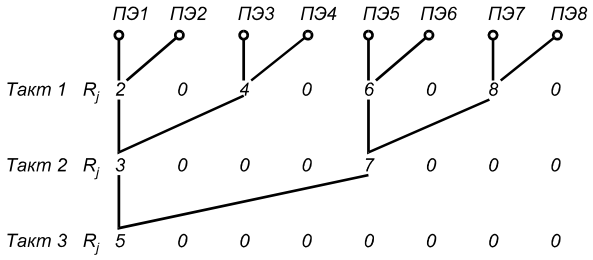


Рис. 2.35. Выполнение операции SUM V в ПМ

Для ПМ довольно сложным является выполнение циклов с оператором IF внутри цикла. В примере

```
    DO 5 I = 1, L
    IF (X (I) - R) 1, 2, 3
1   Y (I) = f1 (X (I))
    GO TO 5
2   Y(I) = f2 (X (I))                    (2.4)
    GO TO 5
3   Y(I) = f3 (X (I))
5   CONTINUE
    STOP
```

как X, так и Y являются векторами, но Y вычисляется по трем различным формулам в зависимости от значений X(I) и R.

Следовательно, если даже длина векторов X и Y равна числу процессоров, т. е. $L = N$, то и тогда все три части вектора Y нельзя вычислять одновременно в ПП, так как в нем имеется только одно ЦУУ, а для одновременного вычисления всех элементов Y в (2.4) требуется три ЦУУ. Наиболее простой выход из положения заключается в том, что участки $f_1(\mathrm{X(I)})$, $f_2(\mathrm{X(I)})$, $f_3(\mathrm{X(I)})$ вычисляются последовательно в единственном ПП.

Этот метод будет эффективным только в том случае, если функция внутри оператора проста и разбивает всю длину вектора L на сплошные, примыкающие друг к другу участки, длина которых $l_1, l_2, l_3 >> N$.

Наиболее известным представителем процессорных матриц является ЭВМ ILLIAC-IV [2].

Контрольные вопросы.

1. Определите сущность конвейера. Что такое конвейер команд и арифметический конвейер?
2. Перечислите и поясните методы ускорения обращения к памяти.
3. Какое влияние на структуру и характеристики ЭВМ оказало введение в состав машины регистров общего назначения?
4. Что такое КЭШ, какова его организация?
5. Охарактеризуйте структуру скалярного конвейерного процессора.
6. Объясните, как выполняется программа в скалярном конвейерном процессоре.
7. Охарактеризуйте структуру векторного конвейерного процессора. Какова роль векторных регистров?
8. Назовите факторы, влияющие на быстродействие векторного конвейерного процессора.
9. Опишите структуру и особенности функционирования ЭВМ типа CRAY.
10. Назовите типы соединительных сетей и способы описания связей.
11. Охарактеризуйте основные виды регулярных связей.
12. В чем сущность коммутаторов типа сред, что такое многомерные кубы?
13. Что такое каскадные коммутаторы? Каковы их свойства?
14. Каковы принципы управления процессом коммутации? Приведите основные характеристики коммутаторов.
15. Опишите структуру процессорной матрицы.
16. Как выполняются векторные команды узлами процессорной матрицы?
17. Охарактеризуйте систему управления процессорного элемента. Что такое активация?
18. Как выполняются команды коммутации в процессорной матрице?
19. Какое влияние на быстродействие процессорной матрицы и ее программу оказывает способ размещения данных в оперативной памяти.

СТРУКТУРЫ ЭВМ С МНОЖЕСТВЕННЫМ ПОТОКОМ КОМАНД

В соответствии с классификацией Флинна (см. § 1.2) к *многопроцессорным ЭВМ* относятся ЭВМ с множественным потоком команд и множественным потоком данных (МКМД-ЭВМ). Основы

управления вычислительных процессом в таких ЭВМ изложены в [4, 12, 13], элементы теории управления — в [14].

В основе МКМД-ЭВМ лежит традиционная последовательная организация программы, расширенная добавлением специальных средств для указания независимых, параллельно исполняемых фрагментов. Такими средствами могут быть *операторы* FORK и JOIN, скобки parbegin ... parend, оператор DO FOR ALL и др. Параллельно-последовательная программа достаточно привычна пользователю и позволяет относительно просто собирать параллельную программу из обычных последовательных программ. МКМД-ЭВМ имеет две разновидности: ЭВМ с общей и индивидуальной памятью. Структура этих ЭВМ представлена на рис. 3.1.

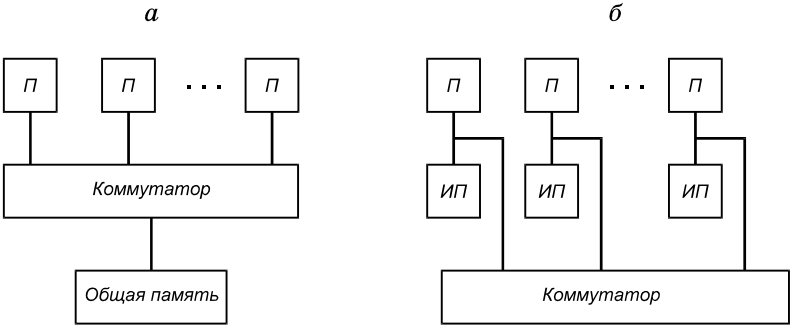


Рис. 3.1. Структура многопроцессорной ЭВМ с общей (*а*) и индивидуальной (*б*) памятью. Здесь:
П — процессор, ИП — индивидуальная память

В качестве коммутатора на рис. 3.1 может использоваться любой коммутатор из описанных в § 2.3. Общая память для повышения пропускной способности обычно содержит более одного блока.

Главное различие между МКМД-ЭВМ с общей и индивидуальной (локальной, распределенной) памятью состоит в характере адресной системы. В машинах с *общей памятью* адресное пространство всех процессоров является единым, следовательно, если в программах нескольких процессоров встречается одна и та же переменная *X*, то эти процессоры будут обращаться в одну и ту же физическую ячейку общей памяти. Это вызывает как положительные, так и отрицательные последствия:

1. Наличие общей памяти не требует физического перемещения данных между взаимодействующими программами, которые параллельно выполняются в разных процессорах. Это упрощает программирование и исключает затраты времени на межпроцессорный обмен.

2. Несколько процессоров могут одновременно обращаться к общим данным и это может привести к получению неверных результатов. Чтобы исключить такие ситуации, необходимо ввести систему синхронизации параллельных процессов, что усложняет механизмы операционной системы.

3. Поскольку при выполнении каждой команды каждым процессором необходимо обращаться в общую память, то требования к пропускной способности коммутатора этой памяти чрезвычайно высоки, что и ограничивает число процессоров в системах с общей памятью величиной 10...20.

В системах с индивидуальной памятью каждый процессор имеет независимое адресное пространство и наличие одной и той же переменной X в программах разных процессоров приводит к обращению в физически разные ячейки индивидуальной памяти этих процессоров. Это приводит к необходимости физического перемещения данных между взаимодействующими программами в разных процессорах, однако, поскольку основная часть обращений производится каждым процессором в собственную память, то требования к коммутатору ослабляются и число процессоров в системах с распределенной памятью и коммутатором типа гиперкуб может достигать нескольких десятков и даже сотен.

В настоящем параграфе будет рассмотрена система управления вычислительным процессом в МКМД-ЭВМ с общей памятью, а в § 3.2 — управление вычислениями в МКМД-ЭВМ с индивидуальной памятью в каждом процессоре.

Типы процессов. Для описания параллелизма независимых ветвей в программах для МКМД-ЭВМ, написанных на ЯВУ, обычно используют операторы FORK, JOIN или аналогичные им по функциям.

Пример программы, использующей эти операторы, представлен в § 1.1.

Введем понятия *процесса*, *ресурса* и *синхронизации*, которые понадобятся далее. А.Шоу [12] определяет процесс следующим

образом: “процесс есть работа, производимая последовательным процессором при выполнении программы с ее данными”. Это означает, что процесс является результатом взаимодействия элементов пары <процессор, программа>. Одна программа может порождать несколько процессов, каждый из которых работает над своим набором данных. Процесс потребляет ресурсы и поэтому является единицей планирования в многопроцессорной системе.

Ресурс является средством, необходимым для развертывания процесса. Ресурсом может быть как аппаратура (процессоры, оперативная память, каналы ввода-вывода, периферийные устройства), так и общие программы и переменные. В мультипрограммных и мультипроцессорных ЭВМ наиболее употребительными общими переменными являются системные таблицы, описывающие состояние системы и процессов, например, таблицы очередей к процессорам, каналам, ВнУ; таблицы распределения памяти, состояния семафоров и др.

Синхронизация есть средство обеспечения временной упорядоченности исполнения взаимодействующих процессов.

Существует два основных вида зависимости процессов: зависимость по данным и зависимость по ресурсам.

В общем случае зависимость по данным имеет место между множеством процессов. Пусть два множества процессов связаны таким образом, что каждый процесс из множества $\{L\}$ получает данные от всех процессов множества $\{M\}$. Пример информационного графа такой зависимости представлен на рис. 3.2. Тогда условие запуска множества процессов для этого примера можно записать так:

if(JOIN1 $\wedge$ JOIN2 $\wedge$ JOIN3) **then** FORK L1, L2

то есть запуск процессов осуществляется по логической схеме “И” после выполнения всех операторов JOIN.

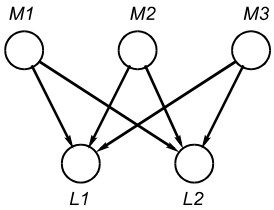


Рис. 3.2. Взаимодействие синхронных процессов

Будем называть такие процессы *синхронными*. Отличительной особенностью синхронных процессов является наличие связного ИГ и вызванный этим обстоятельством запуск по схеме “И”. Иногда такие процессы именуют сильносвязанными.

Другой разновидностью процессов являются *асинхроннные.* Такие процессы не имеют связного ИГ, поэтому иногда называются слабосвязанными (через ресурс). Взаимодействие между ними осуществляется в ситуации, когда несколько таких процессов на основе конкуренции пытаются в одно и то же время захватить единственный ресурс. Это могут быть задачи независимых пользователей, которые одновременно обращаются к каналам ВнУ, требуют объемов физической памяти. Асинхронными являются все системные процессы, работающие с общими таблицами очередей к ресурсам, состояний процессов.

Синхронные процессы, например, *M*1, *M*2, *M*3 на рис. 3.2 могут выполняться двояко: либо им во время запуска отводятся собственные ресурсы (место в памяти, процессор и т. д.) и тогда они независимо выполняются на своих “площадях”; либо эти процессы запускаются как асинхронные на одном и том же наборе ресурсов, конкурируя за их захват.

Поскольку асинхронные процессы пытаются одновременно захватить единственный ресурс, то задача синхронизации в этом случае состоит в том, чтобы обеспечить последовательное во времени использование ресурса процессами, то есть выполнить задачу взаимоисключения процессов. Это соответствует запуску процессов по логической схеме “Исключающее ИЛИ”.

Рассмотрим некоторые методы и средства синхронизации в МКМД-ЭВМ. Сначала коснемся вопроса о средствах, реализующих методы синхронизации. Все процессы между точками синхронизации выполняют обычные команды однопроцессорной ЭВМ. Следовательно, средством выполнения процессов между точками синхронизации является аппаратура процессора. Операции же синхронизации (FORK, JOIN, взаимное исключение) из-за большого объема выполняемой ими работы не могут быть реали-

зованы только в виде аппаратных команд и выполняются также как подпрограммы операционной системы, в совокупности составляющие раздел ОС “Управление процессами и ресурсами”.

Взаимодействие асинхронных процессов на основе семафоров. Рассмотрим следующий пример [12]. Пусть имеется система с общей памятью и двумя процессорами, каждый с одним регистром.

В первом процессоре выполняется процесс $L1$, во втором — $L2$:

$L\ 1$: ... X: = X + 1; ...
$L\ 2$: ... X: = X + 1; ...

Процессы выполняются асинхронно, используя общую переменную X. При выполнении процессов возможно различное их взаиморасположение во времени, например, возможны следующие две ситуации:

$L1$: $R1 := X$; $R1 := R1 + 1$; $X := R1$; ... (3.1)
$L2$: ..... $R2 :=$ X ; $R2 := R2 + 1$; $X := R2$;

$L1$: $R1 := X$; $R1 := R1 + 1$; $X := R1$; ... (3.2)
$L2$: ..... $R2 := X$; $R2 := R2 + 1$; $X := R2$;

Пусть в начальный момент $X = V$. Нетрудно видеть, что в (3.1) второй процессор производит чтение X до завершения всех операций в первом процессоре, поэтому $X = V + 1$. В случае (3.2) второй процессор читает X после завершения всех операций первым процессором, поэтому $X = V + 2$. Таким образом, результат зависит от взаиморасположения процессов во времени, что для асинхронных процессов определяется случайным образом. Значит, результат зависит от случая, что недопустимо. Ясно, что должно учитываться каждое приращение X.

Выход заключается в разрешении входить в *критическую секцию* (КС) только одному из нескольких асинхронных процессов. Под критической секцией понимается участок процесса, в котором процесс нуждается в ресурсе. При использовании критической секции возникают различные проблемы, среди которых можно выделить проблемы состязания (гонок) и тупиков. Условие состя-

зания возникает, когда процессы настолько связаны между собой, что порядок их выполнения влияет на результат. Условие тупиков появляется, если два взаимосвязанных процесса блокируют друг друга при обращении к критической секции. Рассмотрим сначала решение проблемы состязаний.

Если бы КС были короткими, например, длительностью в одну команду, тогда можно было решить задачу взаимного исключения аппаратным способом. Для этого надо было бы на время выполнения такой команды запретить обращение в общую память командам от других процессоров. Однако, КС могут иметь произвольную длительность, поэтому необходимо общее решение данной проблемы.

Это решение предложил Дейкстра [12] в виде семафоров. *Семафором* называется переменная S, связанная, например, с некоторым ресурсом и принимающая два состояния: 0 (запрещено обращение) и 1 (разрешено обращение). Над S определены две операции: V и P. Операция V изменяет значение S семафора на значение $S + 1$. Действие операции P таково: 1) если $S \neq 0$, то P уменьшает значение на единицу; 2) если $S = 0$, то P не изменяет значения S и не завершается до тех пор, пока некоторый другой процесс не изменит значение S с помощью операции V. Операции V и P считаются неделимыми, т. е. не могут исполняться одновременно.

Приведем пример синхронизации двух процессов:

begin semaphore *S*; *S*:=1;
process 1: **begin** *L*1:*P*(*S*);
Критический участок 1;
V(*S*);
Остаток цикла, **go to** *L*1
end (3.3)
process 2: **begin** *L*2:*P*(*S*);
Критический участок 2;
V(*S*);
Остаток цикла, **go to** *L*2
end
end

В (3.3) операторы *parbegin* и *parend* обрамляют блоки *process* 1 и *process* 2, которые могут выполняться параллельно. Процесс может захватить ресурс только тогда, когда *S*:=1. После захвата процесс закрывает семафор операции *P*(*S*) и открывает его вновь после прохождения критической секции *V*(*S*).

Таким образом, семафор *S* обеспечивает неделимость процессов *Li* и, значит, их последовательное выполнение. Это и есть решение задачи взаимного исключения для процессов *Li.*

Семафор обычно реализуется в виде ячейки памяти. Процессы *Li,* чтобы захватить ресурс, вынуждены теперь на конкурентной основе обращаться к ячейке памяти *S*, которая таким образом сама стала ресурсом. Следовательно, обращение к *S* также является критической секцией каждого *Li.* Однако, эта КС значительно короче и ее можно реализовать в виде пары команд, за которыми закрепилось название *TS* (Test and Set — проверить и установить) и *SI* (Store Immediate — записать немедленно). Функции команд *TS* и *SI* представлены на рис. 3.3. На этом рисунке для семафора используется наименование *mutex* (MUTual EXcluding — взаимное исключение). С целью упрощения выражений в дальнейшем тексте в отличие от семафоров Дийкетры закрытым состоянием *mutex*

является 1 (true), открытым — 0 (false). Команды *TS* и *SI* технически реализуются неделимыми.

Установка *mutex* в 1 производится сразу же после чтения *mutex* в регистр *TS*. Это делается для того, чтобы сократить задержку на обращение в память другим процессам во время выполнения команды *TS* процессом *Li.* Если дальнейший анализ регистра *TS* покажет, что в момент чтения *mutex* находился в состоянии 1, то это означает, что распределяемый ресурс занят другим процессом и *Li* будет повторно запрашивать *mutex*. Если же окажется, что *mutex* находился в состоянии 0, то процесс *Li* может входить в КС, а предварительная установка *mutex* в 1 запретит другим процессам использовать ресурс до момента его освобождения. Это делает команда *SI,* которая обращается в ячейку *mutex,* минуя все очереди на обращение к памяти.

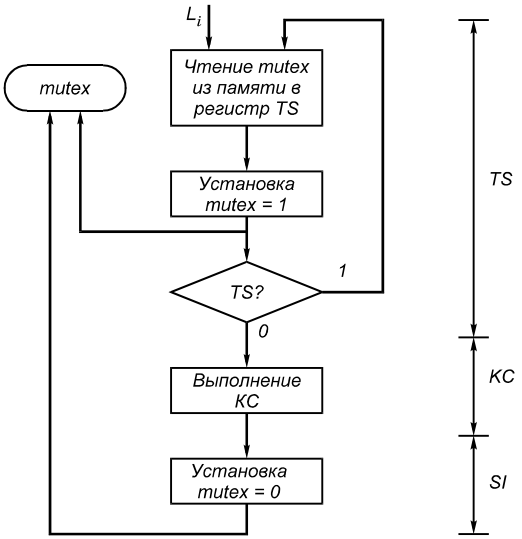


Рис. 3.3. Выполнение команд TS и SI

Взаимодействие двух процессов с целью использования некоторого ресурса можно представить в виде отрезка программы:

Lj: **if** *TS*(*mutex*) **then go to** *Lj*;
Li: **if** *TS*(*mutex*) **then go to** *Li*; (3.4)

KCi; *SI*;

Процессы *Li* и *Lj* выполняются независимо друг от друга и асинхронно. В (3.4) предполагается, что *Li* первым вошел в свою КС. Процесс *Lj* будет выполнять опрос *mutex,* пока *Li* будет находиться в КС. Это называется "*занятым ожиданием*", во время которого *Lj* непроизводительно затрачивает время процессора и циклы памяти.

Более подробно эта ситуация представлена на рис. 3.4, а для тех же двух процессов, хотя рис. 3.4, как и программа (3.4) могут быть расширены для произвольного числа процессов. На рисунке предполагается, что процессы, выполняясь на разных процессорах, одновременно подошли к своим КС.

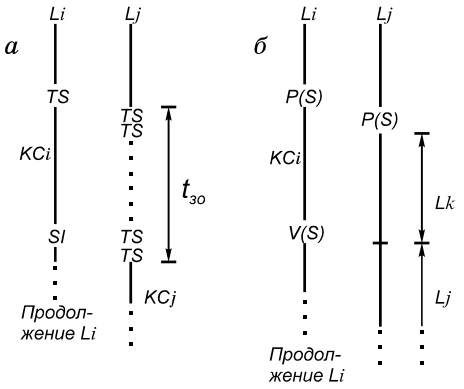


Рис. 3.4. Взаимодействие асинхронных процессов при входе в КС

Поскольку время занятого ожидания $t_{зо}$ может быть произвольно большим, разработаны способы его исключения или уменьшения.

Чтобы устранить занятое ожидание процесса *Lj* или других ожидающих освобождения ресурса *R* процессов, необходимо разработать механизм постановки этих процессов в очередь к ресурсу *R*. Постановка этих процессов в очередь позволит освободить соответствующие процессоры для выполнения других работ до момента освобождения занятого ресурса. С этой целью структура семафора усложняется (рис.3.5).

mutex	S	Адрес начала очереди процесса

Рис. 3.5. Структура ячейки семафора

В поле *S* хранится семафор *S* ресурса *R*. Над *S* выполняются операции *P*(*S*) и *V*(*S*). Семафор *S* может принимать отрицательные

значения. Абсолютная величина такого значения определяет длину очереди к *R*. Семафор *mutex* обеспечивает неделимость операции постановки процессов в очередь к *R*.

Над семафором *S* возможны следующие операции занятия *P*(*S*) и освобождения *V*(*S*) ресурса *R*:

P(*S*): *L*: **if** *TS* (*mutex*) **then go to** *L*; (3.5)
S: = *S* - 1
if *S*<= - 1 **then begin** поставить *L* в очередь к *R*; *SI*;
end;
else *SI*.

V(*S*): *L*: **if** *TS* (*mutex*) **then go to** *L*; (3.6)
S: = *S* + 1
if *S* < 0 **then begin** продвинуть очередь к *R*; *SI*; **end**;
else *SI*.

На рис. 3.4, б показано поведение двух процессов при одновременном обращении к общему ресурсу *R*. Здесь процесс *Li* захватывает ресурс благодаря тому, что он первым успешно выполняет *TS,* а процесс *Lj* становится в очередь к *R,* при этом выполнявший его процессор переключается на другой процесс *Lk*, устраняя тем самым простой процессора. Таким образом, очередь поглотила “занятое ожидание”.

Особенностью операций (3.5) и (3.6) является то, что они включены во все процессы, использующие ресурс *R,* при этом операции *P*(*S*) и *V*(*S*) выполняются над семафором *S* ресурса, а операции *TS* (*mutex*) и *Si* (*mutex*) — над семафором, защищающим операции над *S*. По существу, операции *P*(*S*) и *V*(*S*) сами стали объектом, который обладает всеми признаками ресурса, поскольку эти операции разделяются процессами {*Li*} на основе конкуренции и имеют свой семафор *mutex*, обеспечивающий их неделимость.

На входе операций *P*(*S*) и *V*(*S*) существует занятое ожидание. Если бы эти операции имели большую длительность, имело бы смысл для уменьшения занятого ожидания завести очереди к ним, но обычно обработка общих переменных типа очередей и другого рода таблиц выполняется быстро. Во всяком случае время работы растет не более, чем линейно с ростом объемов таблиц, списков. Поэтому процессу *L* нецелесообразно, обнаружив, что другой процесс захватил их в свое распоряжение, покидать процессор и ста-

новиться в очередь к *mutex*. Время переключения процессора на другой процесс может оказаться больше времени исполнения $P(S)$ или $V(S)$.

С целью упрощения программирования процессов и уменьшения количества ошибок используются синхропримитивы более высокого уровня, в частности мониторы и почтовые ящики [13]. Монитор представляет собой совокупность управляющих переменных, связанных с некоторыми общими ресурсами, и процедур запросов и освобождения этих ресурсов. Монитор "отторгает" от процессов все их критические секции, связанные с данными ресурсами и превращает их в свои процедуры, которые вызываются указанием имен процедуры и монитора. Они доступны всем процессам в том смысле, что любой из процессов может попытаться вызвать любую процедуру, но только один из процессов может войти в процедуру, остальные должны дожидаться того момента, когда будет завершен предыдущий вызов. Процедуры монитора могут содержать только переменные, локализованные в теле монитора.

Почтовый ящик служит примером синхронизации с помощью сообщений. Если процессу А необходимо взаимодействовать с процессом В, он просит систему образовать почтовый ящик для передачи сообщения. После этого процесс А помещает сообщение в почтовый ящик, откуда оно может быть взято процессом в любое время. С программной точки зрения почтовый ящик — это структура данных, для которой фиксированы правила работы с ней. Она включает заголовок, содержащий описание почтового ящика, и несколько гнезд, в которые помещаются сообщения. Число и размеры гнезд задаются при образовании почтового ящика. Правила работы с ящиком различаются в зависимости от его сложности.

Управление процессами и ресурсами. Пусть имеется система типа МКМД с общей памятью. В системе имеется несколько процессоров и несколько классов ресурсов, причем в каждом классе имеется несколько единиц ресурса. К процессорам имеется общая очередь процессов, которой управляет планировщик (диспетчер).

Процессы в этой очереди могут находиться в одном из трех состояний:

- выполняется на некотором процессоре;

• готов к исполнению, но свободный процессор отсутствует;

• блокирован, то есть находится в очереди к одному из ресурсов, а значит, не может выполняться на процессоре.

Рассмотрим, как производится управление процессами и ресурсами в такой системе.

Основной единицей планирования в системе является *задача.* Задача — это независимая единица вычислений, которая обладает собственными ресурсами и представляет пользователя. Описание задачи, содержащее состав требуемых ресурсов, время работы и другие условия выполнения задачи составляются пользователем и входят в состав задачи. Задача состоит из одного или нескольких процессов, находящихся в разной степени подчинения. Каждый процесс характеризуется своим описанием, которое называется *дескриптором.* Дескриптор создается специальной процедурой Create перед выполнением процесса на основе описания задачи и информации, имеющейся в программе пользователя. Дескрипторы хранятся в системной области памяти и используются во время работы процесса. Уничтожение процесса вызывает также и уничтожение дескриптора. Дескриптор содержит следующую информацию:

1. Имя процесса *Li.*

2. Описание ресурсов, занимаемых процессом *Li:* номер процессора, разделы и адреса памяти, информация о других ресурсах.

3. Состояние процесса (выполняется, готов, блокирован).

4. Имя процесса, породившего данный процесс; имена подчиненных процессов, порожденных данным процессом. Процесс *Li* считается выполненным, когда будут выполненными все подчиненные процессы. Для них заводится счетчик, из которого вычитается 1 при выполнении каждого подчиненного процесса. Следовательно, блок 4 дескриптора, называемый *критическим блоком* (не совпадает с критической секцией), выполняет функции операторов FORK и JOIN.

5. Прочие данные: приоритет *Li,* приоритеты подчиненных процессов, предельно допустимое время использования процессора и др.

Дескриптор класса ресурсов создается операционной системой и обычно содержит следующие компоненты:

1. Имя класса ресурса.

2. Состав ресурса, например: число процессоров в системе; число сегментов, страниц основной памяти; число ВнУ одного типа.

3. Таблицу занятости единиц ресурса, которая указывает имена процессов, занимающих каждый ресурс.

4. Описание очереди ждущих процессов, указывает число элементов очереди, адреса начального и конечного элементов очереди.

5. Адрес или имя распределителя, ответственного за порядок занятия единиц ресурса процессами. Распределитель процессора обычно называется планировщиком или диспетчером.

Как уже ранее говорилось, общие переменные также являются разделяемым ресурсом, однако, их описание является ограниченным и состоит из определения семафора типа *mutex*.

Дескрипторы ресурсов хранятся в системной области памяти. Информация, содержащаяся в дескрипторах ресурсов, изменяется при выполнении задачи соответствующими процедурами.

Операции над процессами и ресурсами будем называть *примитивами*. Примитив следует рассматривать как команду, которая выполняется в пределах вызывающего процесса и не является отдельным процессом. Примитивы являются неделимыми, они защищаются семафором типа *mutex*, следовательно в начале и конце такой операции используются команды *TS* и *SI*. Во время выполнения примитива прерывания процессора запрещены, чтобы обеспечить выполнение примитива до освобождения процессора.

При дальнейшем описании примитивов эти защитные операции явно не введены, однако их наличие подразумевается.

Примитивы реализуются программно, аппаратно или смешанным образом. Вызовы примитивов (или команды) размещаются в тексте программы пользователем или включаются в текст программы компьютером на основе информации, размещенной пользователем в исходном тексте задачи.

Основное назначение примитивов — связать задачу с ресурсами системы.

Для управления процессами используются два основных примитива:

1. Create — создать процесс.

2. Destroy — уничтожить один или несколько процессов.

Новый процесс создается задачей или некоторым процессом этой задачи. Создание подчиненного процесса заключается в построении части дескриптора нового процесса. Информация для дескриптора передается от порождающего процесса к порождаемому при выполнении первым примитива Create. Начальные ресурсы памяти M и другие ресурсы R, необходимые для начальной стадии функционирования создаваемого процесса L, предоставляются из состава ресурсов, принадлежащего порождающему процессу, то есть эти ресурсы являются общими. В дальнейшем порожденный процесс может затребовать собственные ресурсы с помощью операции Request.

Приоритет порождаемого процесса не может превышать приоритет порождающего процесса.

Procedure Create (L, E_0, M_0, R_0, P_0)

begin

присвоить процессу имя Li

описать начальные ресурсы R_0 и требуемую память M_0;

задать начальное состояние процесса E_0=“готов”; поместить процесс в очередь готовых;

указать имя процесса-отца и процессов-потомков;

задать приоритет процесса P_0;

end.

Уничтожение процесса Li состоит в возврате всех занятых ресурсов R и областей памяти. При этом изменяются таблицы занятости соответствующих дескрипторов ресурсов. Если процесс находился в состоянии выполнения, он останавливается и вызывается планировщик для реорганизации очереди процессов к процессорам. Наконец, удаляется дескриптор самого уничтожаемого процесса Li. Аналогичные действия выполняются для всех подчиненных процессов, порожденных процессом Li, если они есть, то есть удаляется все дерево процессов. Операция Destroy (Li), где Li имя корневого процесса, выполняется в процессе, породившем Li.

Ее можно описать следующей процедурой:

Procedure Destroy (*Li*);

begin
sched: = **false**;
if *E* (*Li*) = “выполняется” **then**
begin *stop* (*Li*); *sched* = **true**; **end**;
Удалить *Li* из очереди на блокирующий ресурс;
Освободить личные ресусы M и R, включить их в
дескрипторы как свободные;
Удалить из памяти дескриптор процесса *Li*;
if *sched* = **true then** вызвать планировщик;
end.

Для управления ресурсами используются следующие основные примитивы:

1. Request — затребовать ресурс.
2. Release — освободить ресурс.

Операцию Request можно представить в виде процедуры Request (*R*), где *R* — имя класса требуемых ресурсов.

Procedure Request (*R*);

begin
Поместить процесс *Li* в очередь к *R*;
Получить результат распределения очереди *R*(*Q*);
if *Li* $\subset$ *Q* **then** Установить: “*Li* блокирован”
else поместить *Li* в очередь готовых;
Выполнить планирование готовых процессов;
end.

Согласно процедуре запрос процесса *Li* сначала становится в очередь на ресурс (с предварительным запретом прерываний и выполнением операции *TS*), затем эта очередь обрабатывается распределителем данного ресурса, который вырабатывает список *Q* и количество *k* удовлетворенных запросов. Следовательно, за одно обращение может удовлетворяться более одного запроса. Затем в цикле каждый элемент списка *Li* включается в очередь готовых процессов. Для запрашивающего процесса *M* это не делается, поскольку он уже находится в этой очереди. Если *M* оказался не

включенным в список *Li*, он блокируется. Во всех ситуациях включается планировщик для реорганизации очереди.

Procedure Release (*R*);

begin
Включить освобождаемый ресурс в опись свободных дескриптора ресурса;
Получить результат перераспределения очереди *R*(*K, Q*);
Поместить готовые процессы в очередь готовых;
Выполнить планирование готовых;
end.

Чтобы рассмотреть пример функционирования МКМД-системы в общем, введем еще две операции, которые по своим функциям аналогичны действию операторов FORK и JOIN. Сохраним эти названия за соответствующими процедурами. Процедура FORK позволяет запустить одновременно *k* процессов, которые помещены в списке *Q*:

Procedure Fork (*k*, *Q*);

begin
Создать критический блок в дескрипторе порождающего процесса;
for *i* = 1 **to** *k* **do**
Create (*Li*, E_0, M_0, R_0, P_0);
Установить статус порождающего процесса *E*="блокирован";
Произвести планирование очереди процессоров;
end.

В процедуре JOIN *Li* — имя процесса-отца, *l* — имя порожденного процесса, а *k* — число подчиненных процесссов. При каждом вызове процедуры выполняются следующие действия:

Procedure JOIN (*Li*, *l*, *k*);

begin

$k = k - 1$;
if $k = 0$ **then begin**
Установить $E(Li)$: = “готов”;
Включить Li в очередь готовых; **end**;
Выполнить планирование готовых;
end.

На рис. 3.6 приведена диаграмма состояний группы процессов $i1$, $i2$, $i3$, порожденных с помощью примитива FORK основным процессом $i0$, в качестве которого может, например, выступать и задача. Для процессов на рис. 3.6 отмечены 9 точек изменения состояний, а в таблице 3.1 указано состояние процессов и общей очереди к процессорам во всех этих 9 точках. Для рис. 3.6 предполагается, что процессы $i1$ и $i2$ получают ресурсы без блокирования, а процесс $i3$ некоторое время блокирован из-за очереди на ресурсы. Основной процесс $i0$ после выполнения оператора FORK не выполняется до момента, когда сработает критический блок ($k = 0$), но и после этого из-за отсутствия свободного процессора он также блокирован. Одним из первых его действий после

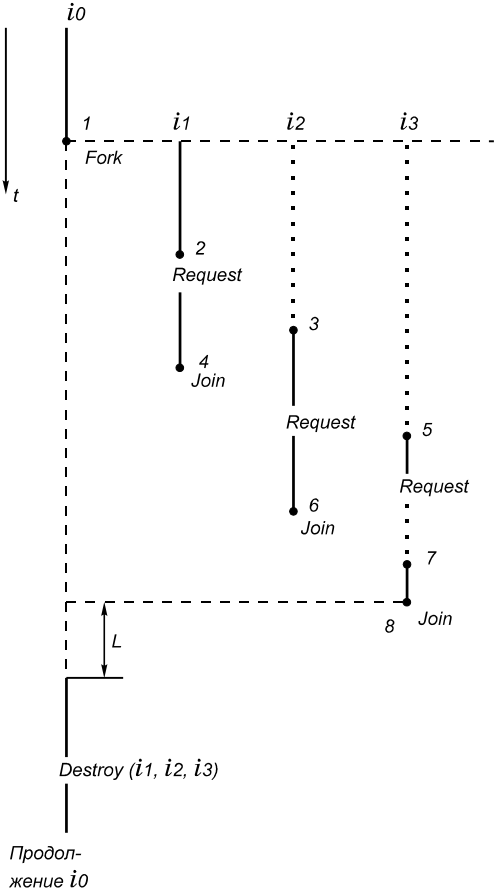


Рис. 3.6. Диаграмма выполнения операций FORK и JOIN

получения процессора является уничтожение процессов $i1$, $i2$, $i3$ и возврат ресурсов.

Таблица 3.1.

Состояния процессов и процессоров
при выполнении примера на рис. 3.6

NN состояния	$P1$	$P2$	Очередь к процессорам
1	N	$i0$	$i1$, $i2$, $i3$
2	N	$i1$	$i2$, $i3$
3	$i2$	$i1$	$i3$, K, L, M
4	$i2$	$i3$	N, L
5	$i2$	N	L
6	L	N	M
7	L	$i3$	M
8	L	M	$i0$
9	$i0$	M	-

В большинстве случаев основной функцией порождающего процесса является управление подчиненными процессами.

Отличие указанных выше примитивов Create, Destroy, Request, Release, Fork, Join от операций над семафорами $P(S)$, $V(S)$ состоит в том, что в примитивах, привязанных к операционной системе, стандартизован большой объем функций, связанных с управлением процессами и ресурсами, в то время как операции $P(S)$ и $V(S)$ являются небольшой частью этих функций, а остальную часть функций должен достраивать пользователь, что приводит к запутанности управления и частым ошибкам.

Примером таких ошибок являются *тупики* (deadlock). Рассмотрим в качестве примера ситуацию, когда два процесса $L1$ и $L2$ обращаются за общим ресурсом, которым является, например, память. Пусть максимальный объем памяти 100 страниц. Тогда представленная ниже ситуация является тупиковой:

Процесс	Требуемое число страниц	Уже имеется страниц	Дополнительное требова-

			ние
*L*1	80	41	39
		+	
*L*2	60	21	39

Свободная память 100 - 62 = 38

Таким образом, свободная память 38 страниц меньше запроса от любого из процессов, поэтому ни один из процессов далее не может выполняться. В настоящее время большое внимание уделяется разработке методов формального определения тупиков [14].

Различают *централизованное* и *децентрализованное управление* мультипроцессорной системой. В первом случае вводится специальный управляющий процессор, в котором хранятся все системные таблицы и который выполняет все примитивы и планирование. Другие процессоры обращаются к управляющему с запросами на выполнение системных функций через систему прерываний.

Централизованное управление реализуется достаточно просто, однако при большом потоке запросов от исполнительных процессоров управляющий процессор может не успевать обрабатывать запросы и исполнительные процессоры будут простаивать.

Этого недостатка лишена более сложная система с децентрализованным управлением. При таком управлении каждый исполнительный процессор содержит ядро операционной системы и в состоянии сам выполнять системные примитивы и производить планирование.

Децентрализованное управление обладает еще одним достоинством: при выходе из строя какого-либо из процессоров производится автоматическое перераспределение работ и система продолжает функционирование.

В зависимости от назначения системы возможны различные *алгоритмы планирования*. Различают планирование циклическое, приоритетное, адаптивное. При *циклическом* планировании процесс, использовавший выделенный ему квант времени, помещается в конец очереди на выполнение. Возможно организовать несколько очередей, в каждой из которых величина кванта времени меняется.

При *приоритетном* планировании каждому процессу присваивается приоритет, который определяет его положение в очереди на выполнение. Процесс с высшим приоритетом ставится в начало очереди, с низшим — в ее конец. Приоритет может быть статическим и динамическим.

При *адаптивном* планировании предполагается контроль над реальным использованием памяти. Каждому процессу устанавливаются значения объема памяти и виртуального времени процессора, которое обратно пропорционально максимальному объему памяти, необходимому процессу. Процессу выделяется необходимый временной интервал только при наличии достаточного объема свободной памяти. Планирование ориентировано на систему процессов с минимальными запросами, т. е. отдается предпочтение процессам небольшого размера с малым временем выполнения.

Управление процессами и ресурсами для многопроцессорных ЭВМ в значительной степени позаимствовало методы управления из ОС мультипрограммных ЭВМ. Дополнительным свойством МКМД ЭВМ является управление синхронными процессами типа FORK-JOIN.

§ 3.2. Многопроцессорные ЭВМ с индивидуальной памятью

МКМД-ЭВМ с *индивидуальной памятью* получили большое распространение ввиду относительной простоты их архитектуры. В таких ЭВМ межпроцессорный обмен осуществляется с помощью передачи сообщений в отличие от описанных в предыдущем параграфе ЭВМ с общей памятью, где такой обмен отсутствует. Наиболее последовательно идея ЭВМ с индивидуальной памятью отражена в транспьютерах и параллельных системах, построенных на их основе [15, 16].

Транспьютер (transputer = transfer (передатчик) + computer (вычислитель)) является элементом построения многопроцессорных систем, выполненном на одном кристалле СБИС (рис. 3.7, а). Он включает средства для выполнения вычислений (центральный процессор, АЛУ для операций с плавающей запятой, внутрикристальную память объемом 2...4 кбайта) и 4 канала для связи с другими транспьютерами и внешними устройствами. Встроенный

интерфейс позволяет подключать внешнюю память объемом до 4 Гбайт.

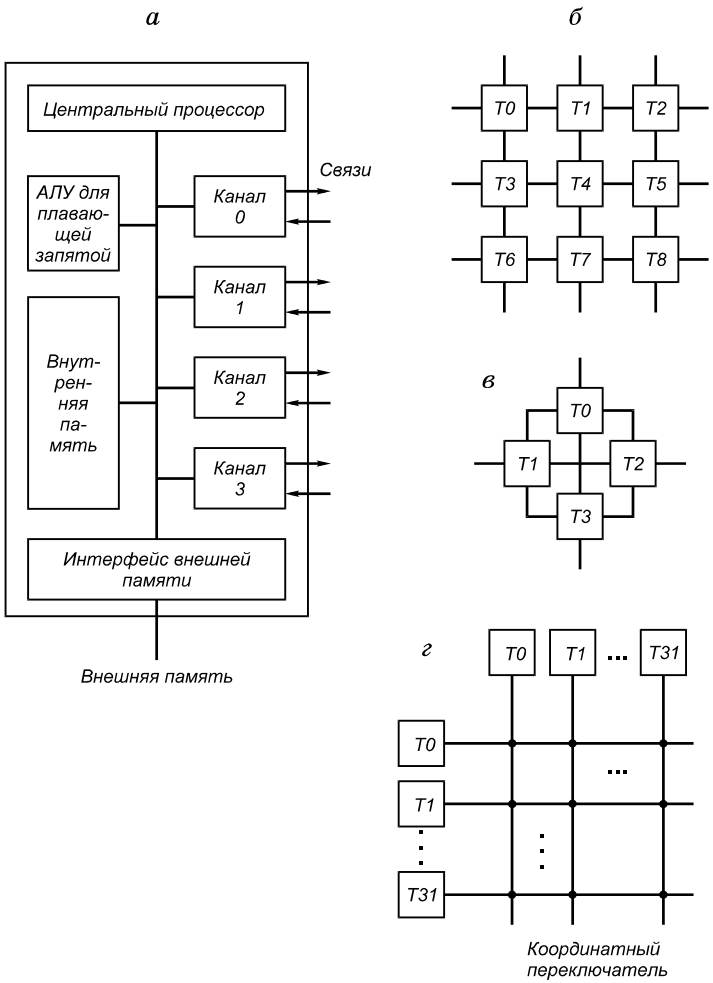


Рис. 3.7.Структура транспьютера и систем на его основе

Для образования транспьютерных систем требуемого размера каналы различных транспьютеров могут соединяться непосредственно (рис. 3.7, б, в) или через коммутаторы типа координатный переключатель на 32 входа и выхода, который обеспечивает одно-

временно 16 пар связей (рис. 3.7, г). Такие переключатели могут настраиваться программно или вручную и входят в комплект транспьютерных СБИС.

Размер транспьютерных систем не ограничен, а структура системы может быть сетевой, иерархической или смешанной. Для описания организации и функционирования транспьютера и многопроцессорных систем на его основе необходимо сначала рассмотреть элементы языка Оккам.

Введение в Оккам. Организация транспьютеров основана на языке *Оккам* (Occam), разработанном под руководством Хоара (C.A.R.Hoar) в 1984 году. Основой языка являются: средства описания параллелизма выполняемых процессов; средства описания межпроцессорного обмена данными; средства описания размещения процессов по единицам оборудования. Рассмотрим основные конструкции этого языка.

Согласно Хоару [17] Оккам-программа — это процесс. Сложные процессы строятся из процессов-примитивов. Рассмотрим некоторые процессы-композиции в таком порядке: последовательные процессы, параллельные процессы, каналы в Оккаме, циклы, подпрограммы, размещение процессов, примеры вычислительных программ.

Последовательные процессы описываются с помощью конструктора *SEQ*. Чтобы записать последовательный процесс, необходимо записать имя конструктора последовательного процесса *SEQ*, а затем с отступом в две позиции записать его подпроцессы в порядке сверху вниз, например:

```
SEQ
  a: = 1
  b: = a + 7
  c: = b * 12
  d: = ((b + c) * 100)/4
```

Процесс *SEQ* считается законченным, когда последовательно будут выполнены все его подпроцессы.

Последовательный условный конструктор состоит из ключевого слова IF и списка процессов-компонент, у каждого из которых есть свое условие исполнения. Число компонент не ограничено.

Процессы-компоненты записываются под своим условием и представляют собой либо процессы-примитивы, либо процессы-композиции. При выполнении IF условия просматриваются сверху вниз до тех пор, пока не будет найдено первое условие, имеющее значение “истина”. Затем выполняется процесс, относящийся к этому условию, например:

```
IF
   x < 0
     y: = -1
   x = 0
     y: = 0
   x > 0
     y: = 1
```

Следует отметить, что в конструкторе IF выполняется только один процесс, соответствующий первому истинному условию.

Параллельные процессы формируются с помощью конструктора PAR, например:

```
PAR
   a: = b - 5
   d: = b + 1
```

Здесь порядок выполнения подпроцессов является произвольным. Параллельный процесс считается законченным, когда выполнены все его подпроцессы.

При выполнении параллельных процессов возможны ошибки программирования следующего вида:

```
PAR
   a: = b - 5                                    (3.7)
   b: = a + 2
```

Здесь одновременное выполнение подпроцессов недопустимо, так как они зависят друг от друга (прямая и обратная типы зависимостей, см. § 1.1). Компиляторы должны автоматически выявлять ошибки типа (3.7).

Операции ввода-вывода рассмотрим на следующем примере:

```
SEQ
```

```
c1 ? x
c2 ! x * x
```

Подпроцессы в этом конструкторе *SEQ* обозначают следующее: сначала через *канал* *c*1 в транспьютер вводится значение, которое будет размещено в ячейке x. Знак “?” обозначает операцию ввода. Второй подпроцесс вычисляет выражение $x*x$ и результат выводит из транспьютера через канал *c*2. Знак “!” обозначает операцию вывода.

При программировании ввода-вывода также возможны “тупики”, например:

```
L1              L2
SEQ             SEQ
   c1 ! a          c2 ! x          (3.7)
   c2 ? b          c1 ? y
```

В этом примере процесс *L*1 не может обрабатываться, пока процесс *L*2 не прочтет из канала *c*1 значение *a*; процесс *L*2 не может этого сделать, пока не выполнит операцию записи значения *y* в канал *c*2. Но прием значения из *c*2 в переменную *b* в процессе *L*1 может произойти только после операции записи *a* в канал *c*1 и т. д.

В языке Оккам имеется два вида циклов: неограниченный цикл последовательного выполнения процессов WHILE и ограниченный цикл или размножитель процессов FORK.

Цикл WHILE заканчивает работу при выполнении некоторого условия, поэтому заранее число итераций этого цикла неизвестно.

Рассмотрим примеры:

```
WHILE TRUE
    SEQ
       c1 ? y                    (3.8)
       c2 ! x *x
```

Поскольку значение условия не меняется (значение логической переменной постоянно равно TRUE), то цикл будет бесконечно вырабатывать на выходе значение *x*.

Другой цикл:

```
SEQ
   c1 ? x
   WHILE x > = 0
```

```
SEQ
  c2 ! x*x
  c1 ? x
```

возводит в квадрат только положительные числа, а при получении отрицательного числа процесс завершается.

Цикл WHILE используется и в конструкторе PAR:

```
CHAN OF INT cc:
PAR
  WHILE TRUE
    INT x:
    SEQ
      c1 ? x                                    (3.9)
      cc ! x * x
  WHILE TRUE
    INT y:
    SEQ
      cc ? y
      c2 ! y * y
```

Здесь вычисляется четвертая степень входных целых значений. Для этого два процесса необходимо связать каналом *cc*. Поскольку канал *cc* находится внутри данной параллельной конструкции, поэтому он описывается перед началом конструкции предложением CHAN OF, а INT — описание переменной целого типа.

В этой конструкции два процесса образуют конвейер, обе ступени которого работают параллельно. Оба процесса начинают одновременно и взаимодействуют через канал *cc*. Ввод и вывод происходят синхронно. Ввод не завершится, пока не произойдет вывод по тому же каналу, и наоборот. Если один из процессов достиг состояния обмена, а другой еще нет, то первый приостанавливается и ждет, когда второй процесс также будет готов к обмену.

Когда взаимодействие произойдет, оба процесса продолжат работу независимо. Очередная синхронизация произойдет при передаче следующего промежуточного результата.

Повторитель FOR используется в конструкциях SEQ, PAR, IF, например:

```
CHAN OF INT c:
SEQ i = 1 FOR n
    c ! i
```

Этот процесс последовательно выводит в канал с значения от 1 до *n*.

Повторитель в конструкции PAR создает массив параллельных процессов:

```
[n + 1] CHAN OF INT pipe:
PAR i = 0 FOR n
    WHILE TRUE
        INT x:
        SEQ
            pipe [i] ? x
            pipe [i + 1] ! x
```

Здесь образуется буфер на *n* значений, работающий по принципу “первым пришел — первым ушел”.

В Оккаме используются процедуры, содержащие формальные параметры, которые при вызове процедуры заменяются на фактические.

Если процесс (3.8) записать в виде процедуры:

```
PROC squar (CHAN OF INT in, out)
WHILE TRUE
    INT x:
    SEQ
        in ? x
        out ! x * x
```

то параллельный процесс (3.9) записывается в простой форме:

```
CHAN OF INT cc:
PAR
    squar (c1, cc)                                    (3.10)
    squar (cc, c2)
```

Параллельные процессы распределяются между связанными в сеть процессорами конструкцией PLACED PAR. Каждый процес-

сор с его локальной памятью исполняет один из процессов-компонент.

Вернемся к примеру (3.10). Пусть имеется двухпроцессорная система, на которой необходимо выполнить параллельные процессы. У каждого из процессоров (обозначим их номерами 1 и 2) имеются локальная память и по два порта связи — link 1.in (входной) и link 2.out (выходной). Физическая связь между процессорами осуществляется соединением попарно их портов, соответствующих одному каналу. Тогда распределение частей задачи (3.10) по процессорам можно описать следующим образом:

```
CHAN OF INT cc:
PLACED PAR
  PROCESSOR 1
    link 1.in IS c1:
    link 2.out IS cc:                                    (3.11.)
      squar (c1,cc)
  PROCESSOR 2
    link 1.in IS cc:
    link 2.out IS c2:
      squar (cc,c2)
```

На рис. 3.8 показана схема соответствующих этой программе соединений. Естественно, перед выполнением программы (3.11) транспьютеры уже должны быть заранее физически соединены согласно рис. 3.8.

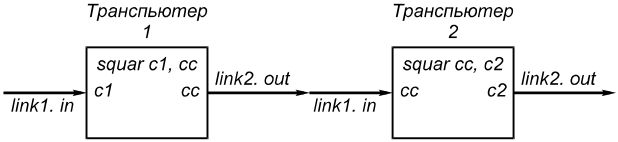


Рис. 3.8 Схема размещения процессов по транспьютерам

Размещение с помощью оператора PLACED PAR может выполняться только для именованных процессов, как это сделано в (3.11) для процедур. Следовательно, конструкция типа:

```
PAR
    a: = b - 5
    d: = b + 1
```

не может быть размещена по двум транспьютерам и может выполняться только в одном транспьютере квазипараллельно (в режиме разделения времени).

Таким образом, вычисления в транспьютерной системе требуют написания двух программ: программы вычислений и программы, закрепляющей ресурсы за процессорами. Более подробно эти вопросы рассмотрены в § 5.4.

Для оценки возможностей транспьютерных систем и языка Оккам рассмотрим пример — выполнение быстрого преобразования Фурье (БПФ), которое является одним из основных алгоритмов цифровой обработки сигналов.

Основной операцией БПФ является операция “бабочка”. Она имеет следующий вид:

$$a' = a + b,$$
$$b' = (a - b) * W^{k}, W = e^{i(2\pi/N)},$$

где все значения комплексные. Эту операцию можно записать в виде одного именованного Оккам-процесса, который по двум входным каналам — ain и bin — получает значения *a* и *b*, а еще по двум — *aout* и *bout* — выдает для последующей обработки *a*' и *b*'.

Программа имеет такой вид:

```
PROC butterfly (CHAN OF REAL32 ain, bin, aout, bout,
VAL REAL32 wreal, wimag)
  REAL32 areal, aimag, breal, bimag:
  WHILE TRUE
    SEQ
    PAR -- одновременное считывание a и b
      SEQ
        ain ? areal
        ain ? aimag
      SEQ
        bin ? breal                                    (3.12)
        bin ? bimag
    PAR -- одновременный вывод a' и b'
      SEQ
        aout ? areal + breal
        aout ? aimag + bimag
      SEQ
        bout ? (wreal * (areal - breal))
         - (wimag * (aimag - bimag))
        bout ? (wreal * (aimag - bimag))
         - (wimag * (areal - breal))
```

Пусть требуется выполнять 8-точечное БПФ. Параллельная программа алгоритма следующая:

```
VAL [12] INT ktable IS [0,1,2,3,0,0,2,2,0,0,0,0]:
  CHAN OF REAL32 c [32]:
    PAR stage = 0 FOR 3
      PAR node = 0 FOR 4                               (3.13)
        butterfly (c [(stage * 8) + node],
                   c [(stage * 8) + node + 4],
                   c [((stage + 1) * 8)+ node * 2)],
                   c [((stage + 1) * 8) + (node * 2)+ 1],
                   twidle.real [ktable [(stage * 4) + node]],
                   twidle.imag [ktable [(stage * 4) + node]])
```

Здесь в массивах twidle.real и twidle.imag хранятся заранее посчитанные константы W^k twidle[k] = $e^{i(2\pi k/N)}$

Во входные каналы c[0]...c[7] подаются входные данные, из выходных каналов c[24]...c[31] считываются результаты.

Изображая процесс butterfly в виде квадрата, а каналы в виде отрезков с указанием направления передачи данных, получаем сеть, изображенную на рис. 3.9.

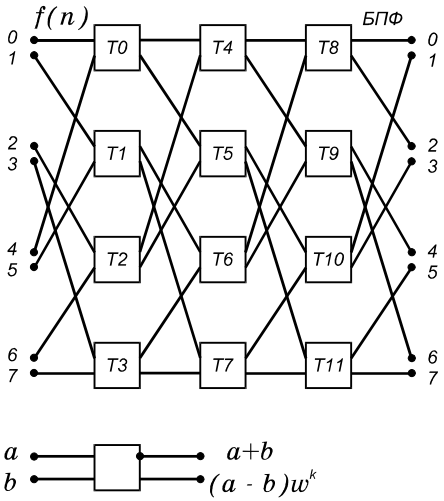


Рис. 3.9. Мультитранспьютерная сеть для БПФ

Если 12 транспьютеров связать между собой в соответствии с этим рисунком и загрузить в каждый программу (3.12) и коэффициенты W в соответствии с программой (3.13), то получим вычислительную систему, которая может выполнять 8-точечное БПФ по программе (3.13).

В этой вычислительной системе транспьютеры образуют конвейер из трех ступеней. Все транспьютеры работают параллельно, позволяя выполнять множественное БПФ в потоковом режиме.

Организация транспьютеров. Состав оборудования транспьютера представлен на рис. 3.7. Рассмотрим особенности его структуры.

Каждый канал транспьютера физически состоит из двух однонаправленных каналов, один для работы в прямом, другой — для работы в обратном направлении, обозначаемые как link.in и link.out. Таким образом, один канал транспьютера соответствует двум каналам языка Оккам. Поскольку каждый канал транспьютера имеет автономное управление, то все каналы могут работать независимо друг от друга и от процессоров транспьютера.

АЛУ транспьютера, а значит, и система команд строятся по стековому принципу. На рис. 3.10 представлена регистровая структура центрального процессора.

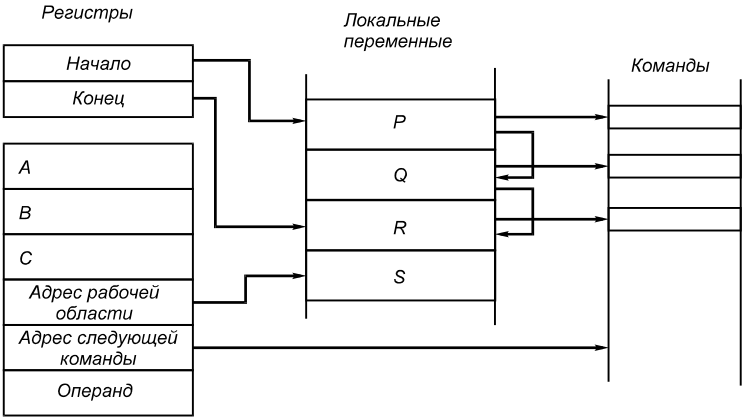


Рис. 3.10. Обработка параллельных процессов

В ЦП используются 6 регистров по 32 разряда каждый:

- указатель рабочей области для локальных переменных программы;
- указатель следующей команды;
- регистр операндов, в котором формируются операнды и команды;
- *A*, *B* и *C* — регистры, образующие вычислительный стек.

В вычислительном стеке выполняются не только арифметические и логические операции, но и команды планирования параллельных процессов и коммуникаций, в него записываются параметры при вызове процедур и др.

Наличие вычислительного стека устраняет необходимость задания в командах явного указания регистра. Например, команда *ADD* складывает значения из регистров *A* и *B*, помещает результат в *A* и копирует *C* в *B*. Поэтому большинство команд транспьютера (70-80%) являются однобайтовыми.

В транспьютере, кроме вычислительного стека ЦП для целочисленной арифметики, имеется стек для работы над данными с плавающей запятой с регистрами *AF*, *BF*, *CF*.

Список команд транспьютера включает 110 команд. Они делятся на две группы: с прямой адресацией (один байт) и с косвенной адресацией (два или более байтов).

Транспьютер может одновременно обрабатывать любое число параллельных процессов. Он имеет специальный планировщик, который производит распределение времени между ними. В любой момент времени параллельные процессы делятся на два класса: активные процессы (выполняются или готовы к выполнению) и неактивные процессы (ожидают ввода-вывода или определенного времени).

Активные процессы, ожидающие выполнения, помещаются в планировочный список. Планировочный список является связным списком рабочих областей этих активных процессов в памяти и задается значениями двух регистров, один из которых указывает на первый процесс в списке, а другой — на последний. Состояние процесса, готового к выполнению, сохраняется в его рабочей области. Состояние определяется двумя словами — текущим значением указателя инструкций и указателем на рабочую область следующего процесса в планировочном списке. В ситуации, изображенной на рис. 3.10, имеется четыре активных процесса, причем процесс *S* выполняется, а процессы *P*, *Q* и *R* ожидают выполнения в планировочном списке.

Команда транспьютера *start process* создает новый активный процесс, добавляя его в конец планировочного списка. Перед выполнением этой команды в *A*-регистр вычислительного стека должен быть загружен указатель инструкций этого процесса, а в *B*-регистр — указатель его рабочей области. Команда *start process* позволяет новому параллельному процессу выполняться вместе с другими процессами, которые транспьютер обрабатывает в данное время.

Команда *end process* завершает текущий процесс, убирая его из планировочного списка. В Оккаме конструкция *PAR* может закончиться только тогда, когда завершатся все ее компоненты параллельного процесса. Каждая команда *start process* увеличивает их число, а *end process* уменьшает. В транспьютере предусмотрен специальный механизм учета числа незавершившихся компонент данной параллельной конструкции (необходимо учитывать как активные, так и неактивные процессы).

При обработке параллельных процессов на самом деле присутствует не один, а два планировочных списка — список высокого приоритета и список низкого приоритета. Процессы с низким приоритетом могут выполняться только тогда, когда список высокого приоритета пуст, что должно быть его нормальным состоянием. Процессы с высоким приоритетом обычно вводятся для обеспечения отклика системы на них в реальном времени.

Коммуникации между параллельными процессами осуществляются через каналы. Для организации этого объема используются команды транспьютера "*input message*" и "*output message*". Эти команды используют адрес канала, чтобы установить, является он внутренним (в том же транспьютере) или внешним. Внутренний канал реализуется одним словом памяти, а обмен осуществляется путем пересылок между рабочими областями этих процессов в памяти транспьютера.

Несмотря на принципиальные различия в реализации внутренних и внешних каналов команды обмена одинаковы и различаются только адресами. Это позволяет осуществить компиляцию процедур безотносительно к способу реализации каналов, а следовательно, и к конфигурации вычислительной системы.

Рассмотрим пересылку данных по внешнему каналу. Команда пересылки направляет автономному канальному интерфейсу задание на передачу данных и приостанавливает выполнение процесса.

После окончания передачи данных канальный интерфейс помещает этот процесс в планировочный список. Во время обмена по внешнему каналу оба обменивающихся процесса становятся неактивными. Это позволяет транспьютеру продолжать обработку других процессов во время пересылки через более медленные внешние каналы.

Каждый канальный интерфейс использует три своих регистра, в которые загружаются указатель на рабочую область процесса, адрес пересылаемых данных и количество пересылаемых байтов.

В следующем примере процессы *P* и *Q*, которые выполняются на различных транспьютерах, обмениваются данными по внешнему каналу *C*, реализованному в виде линии связи, соединяющей эти два транспьютера (рис. 3.11).

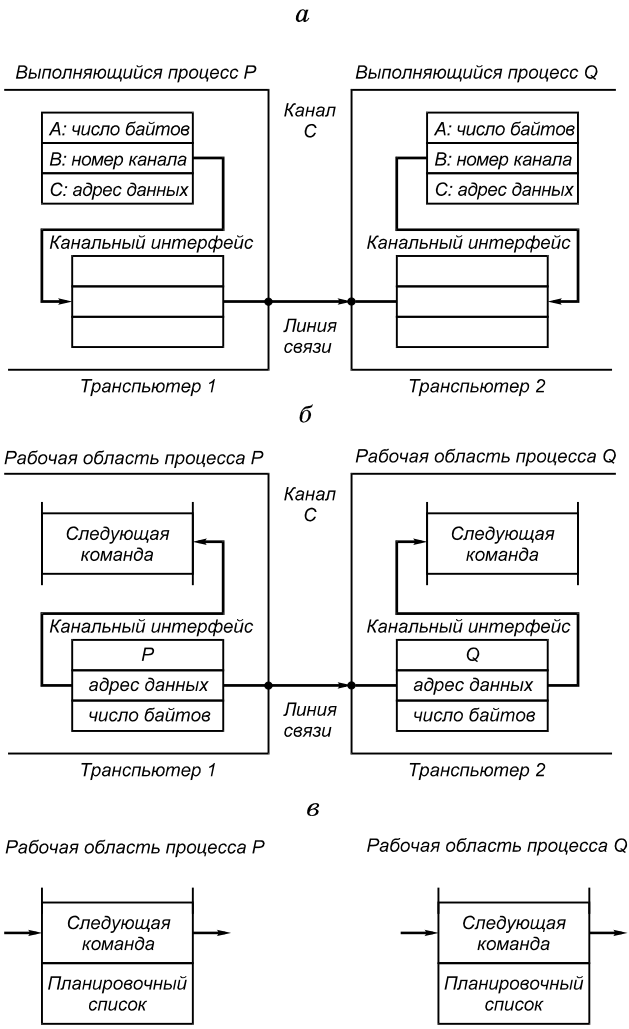


Рис. 3.11. Пересылка данных по внешнему каналу

Пусть *P* передает данные, а Q принимает (рис. 3.11, а). Когда процесс *P* выполняет команду *output message*, регистры канального интерфейса транспьютера, на котором выполняется *P*, инициали-

зируются, а процесс *P* прерывается и становится неактивным. Аналогичные действия происходят в другом транспьютере при выполнении процессом *Q* команды *input message* (рис. 3.11, б).

Когда оба канальных интерфейса инициализированы, происходит копирование данных по межтранспьютерной линии связи. После этого процессы *P* и *Q* становятся активными и возвращаются в свои планировочные списки (рис. 3.11, в).

Для пересылки данных используется простой протокол, не зависящий от разрядности транспьютера, что позволяет соединять транспьютеры разных типов.

Сообщения передаются в виде отдельных пакетов, каждый из которых содержит один байт данных, поэтому наличие буфера в один байт в принимающем транспьютере является достаточным для исключения потерь при пересылках.

После отправления пакета данных транспьютер ожидает получения пакета подтверждения от принимающего транспьютера. Пакет подтверждения показывает, что процесс-получатель готов принять этот байт и что канал принимающего транспьютера может начать прием следующего байта. Форматы пакетов данных и пакетов подтверждения показаны на рис. 3.12.

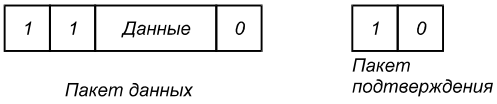


Рис. 3.12. Форматы пакетов данных и пакетов подтверждения

Протокол передачи пакета данных позволяет отсылать пакет подтверждения, как только транспьютер идентифицировал начало пакета данных. Подтверждение может быть получено передающим транспьютером еще до завершения передачи всего пакета данных, и поэтому пересылка данных может быть непрерывной, то есть без пауз между пакетами.

В последних модификациях транспьютеров для упрощения программирования и увеличения пропускной способности физических каналов связи используется *процессор виртуальных каналов* VCP (Virtual Chanal Processor). VCP позволяет использовать на этапе программирования неограниченное число виртуальных каналов.

§ 3.3. ЭВМ с управлением от потока данных

Как отмечалось в § 1.2, кроме МКМД-ЭВМ типа IF с управлением от потока команд существуют МКМД-ЭВМ типа DF с управлением от потока данных.

Рассмотрим принципы построения DF-ЭВМ на примере машины Денниса [1]. В таких машинах отсутствуют понятия “память данных”, “переменная”, “счетчик команд”. Вместо этого команды (операторы) управляются данными. Считается, что команда готова к выполнению, если данные присутствуют на каждом из ее входных портов и отсутствуют в выходном порту. Программа представляет собой направленный граф, образованный соединенными между собой командами: выходной порт одной команды соединен с входным портом другой команды. Следовательно, порядок выполнения команд определяется не счетчиком команд, а движением потока данных в командах.

Принцип обслуживания потоков данных рассмотрим на уровне языка программирования, а структуру и функционирование DF-ЭВМ — на уровне исполнения конкретной программы на машинном языке.

Хотя для потоковых ЭВМ разрабатываются языки программирования разного типа, для ясности изложения принципов функционирования далее будет использован двумерный графический язык. В состав языка входят следующие понятия: “исполнительный элемент”, “информация”, “связи”.

Основные типы исполнительных элементов приведены на рис. 3.13. Информация представляется в виде *токенов* (зачерненные точки), которые передаются по линиям связи и поглощаются исполнительными устройствами. Имеется два вида информации: числовая и управляющая.

Линии связи — это аналог понятий “переменная”, “память”. Здесь и далее на рисунках используются два типа линий связи: данных (сплошные линии) и управления (штриховые линии).

Операция “размножения” выполняется, если есть токен на входе и выходные линии пусты. Блок выполнения операций работает при наличии двух токенов на входе и свободном выходе.

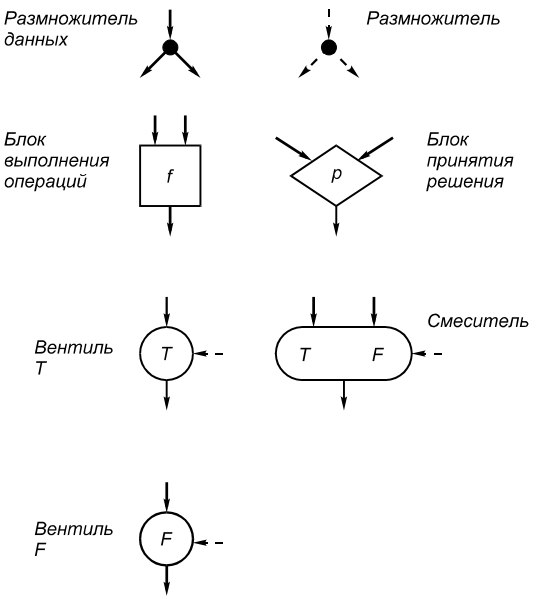


Рис. 3.13.Исполнительные элементы двумерного языка

Блок принятия решения выполняет операцию отношения типа $a > b$, $a = b$; результат — T (истина) или ложь F (ложь). Вентили T и F пропускают токен данных на выход в том случае, если они находятся в состоянии T или F. В зависимости от управляющего сигнала на выход передается входная информация T или F. Рассмотрим пример выполнения программы, записанной на языке PL-1:

```
Z: PROCEDURE (X, Y);
   DO      I = 1 TO X;
        IF Y > 1 THEN Y=Y*Y;
                 ELSE Y=Y+Y+2;
   END;
END;
```

Программа приведена на рис. 3.14. Там же указано начальное состояние программы при значениях входных переменных X = 2 и Y = 0. Слева на рисунке — тело цикла, а справа — управление циклом DO. Отдельные блоки имеют на входах константы. Это означает, что как только блок выполняет предписанную ему работу, константа вновь автоматически восстанавливается на входе.

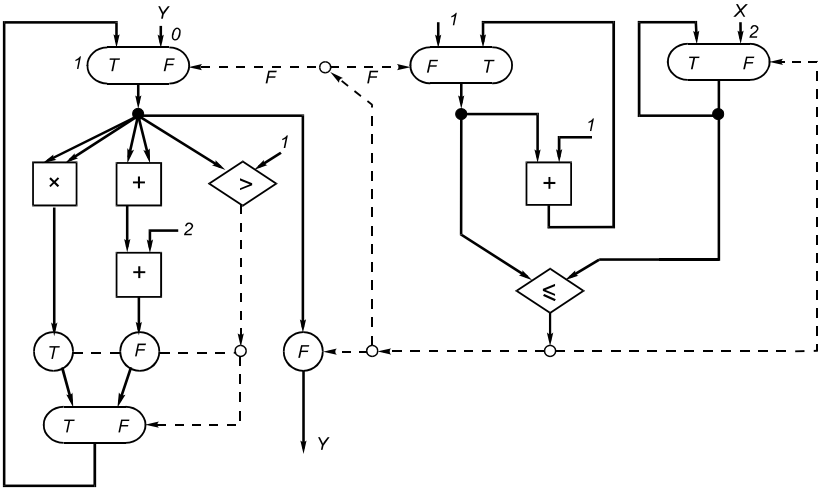


Рис. 3.14. Графическое представление программы
на языке потоков данных

Фазы выполнения программы изображены на рис. 3.15. Дадим некоторые пояснения. В соответствии с начальным состоянием три блока готовы к работе. Один смеситель (вверху слева) находится в состоянии готовности, поскольку к нему приложен управляющий токен FALSE, а на его входной линии *F* имеется токен данных. Два других смесителя также готовы к работе. Таким образом, одновременно выполняются три операции (см. рис. 3.15, а).

Первый смеситель передает величину 0 через размножитель на шесть входов других блоков. Через третий смеситель (вверху справа) и размножитель величина 2 поступает на вход *T* этого же смесителя и на блок принятия решения. Работа второго смесителя (вверху по центру) завершается передачей величины 1 на исполнительный блок и блок принятия решения. Теперь пять блоков готовы к работе в параллельном режиме. Блок принятия решения по критерию “первый операнд отношения больше второго” формирует управляющий токен FALSE, который посылается в три блока: вентили *T*, *F* и нижний смеситель. В результате блок принятия решения по критерию “первый операнд отношения меньше или равен второму” формирует управляющий токен TPUE, направляе-

мый четырем адресатам. Блок сложения пошлет величину 2 обратно на вход второго смесителя.

Момент, когда к работе готовы три вентиля, соответствует изображению на рис. 3.15, б. Поскольку вентиль *F* имеет управляющий токен TRUE, величина 0 будет просто "поглощена" этим вентилем (предотвращается вывод из программы ошибочного результата). То же самое произойдет и в вентиле *T*, но второй вентиль *F* формирует величину 2, которая переведет в состояние готовности расположенный под этим вентилем смеситель, пересылающий величину 2 на другой смеситель. При графическом изображении программы удается отразить тот факт, что оба арифметических выражения, определенные в операторе IF, могут вычисляться параллельно с вычислением отношения оператора IF ($Y >$ 1). Описываемые действия завершаются работой вентилей *T* и *F*, осуществляющих выбор нужного выражения с целью его передачи для нового шага итерации.

Одно из следующих состояний программы отражено на рис. 3.15, в. Величина 2 будет возвращена на вход программы в результате нового повторения циклического процесса в левой части графа, т. е. действие программы аналогично предыдущему. На выходные линии никакие данные помещены не будут, но при этом блок принятия решения по критерию "первый операнд отношения больше второго" формирует на выходе значение TRUE, что дает возможность входным данным вентиля *T* пройти на его выход и далее, в верхнюю часть графа.

Очередное состояние программы иллюстрирует рис. 3.15, г.

Блок принятия решения по критерию "первый операнд отношения меньше или равен второму" формирует управляющий токен FALSE, рассылаемый четырем адресатам программы.

На рис. 3.15, д изображено следующее состояние программы: на выходе вентиля *F*, расположенного в середине графа,

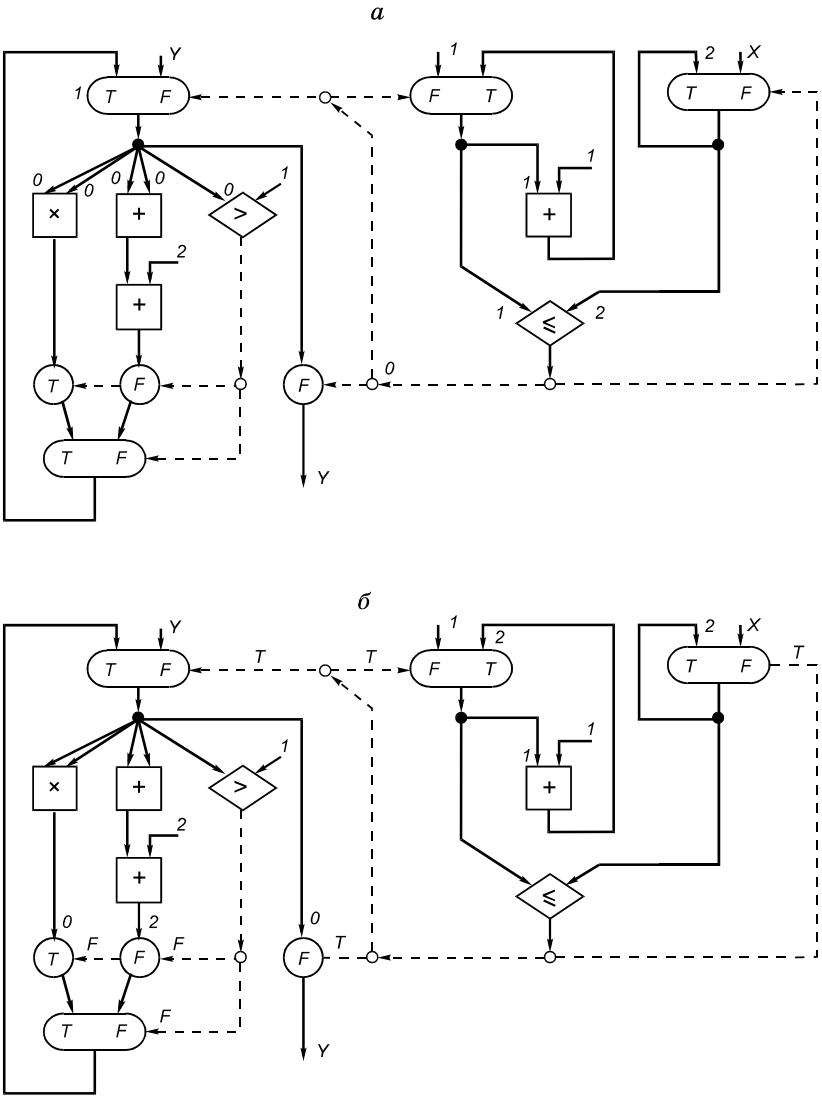


Рис. 3.15. Процесс выполнения потоковой программы

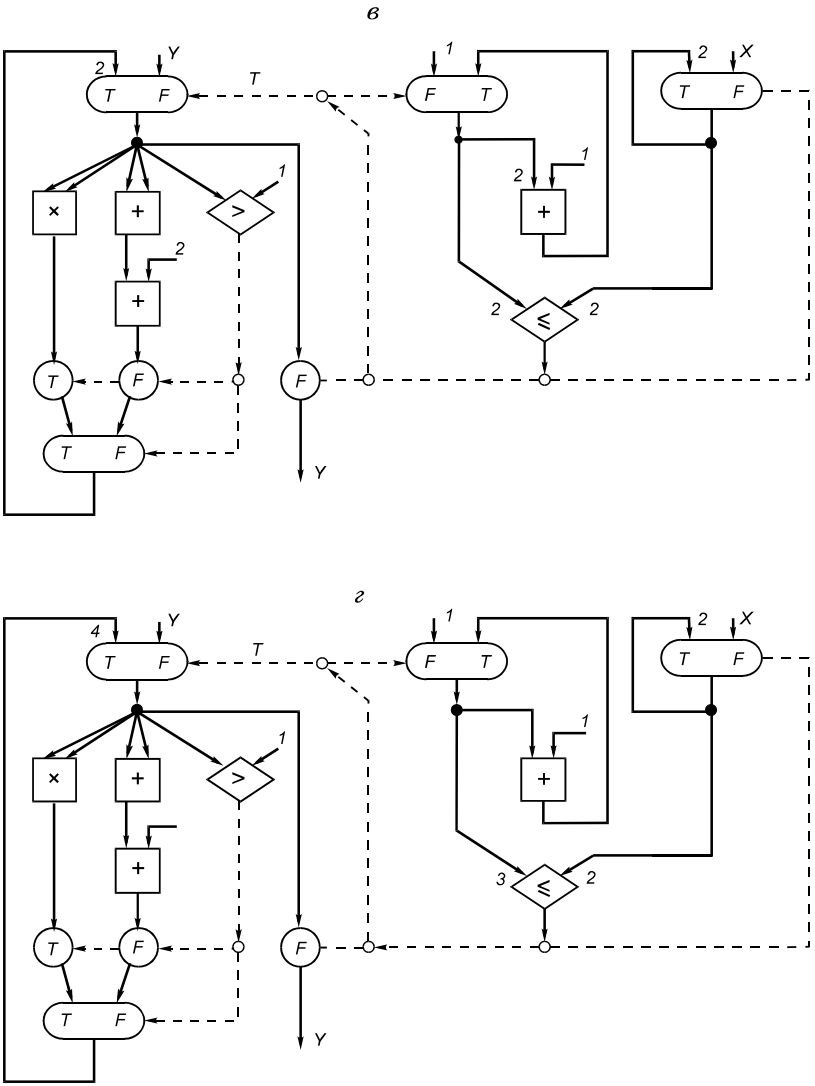


Рис. 3.15. Процесс выполнения потоковой программы (продолжение)

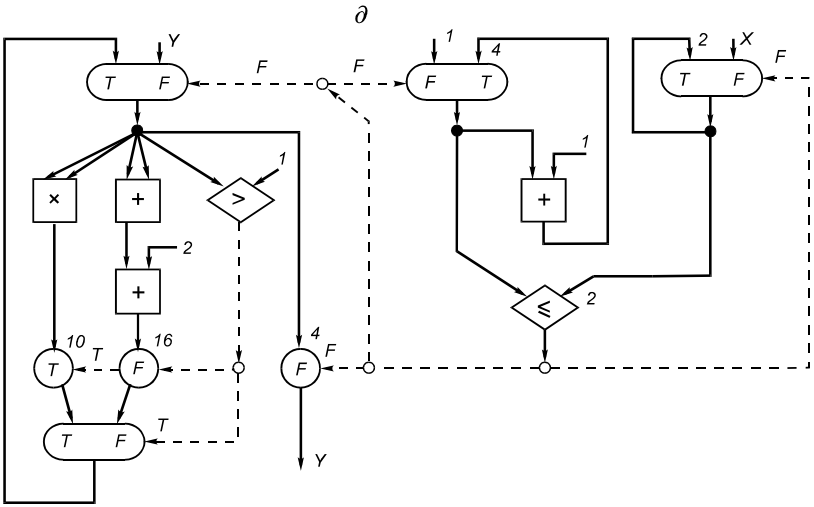


Рис. 3.15. Процесс выполнения потоковой программы (продолжение)

имеются управляющий токен FALSE и токен данных, значение которого равно 4 (поэтому на выходной линии программы появится величина 4). Хотя результат получен, программа все еще продолжает работать, поэтому важно проследить ее работу до момента остановки. Вентиль *F*, расположенный в левой нижней части, уничтожит оба своих входных сигнала. Вентиль *F* пропустит величину 16, которая пройдет через смеситель в верхнюю часть графа.

Здесь дальнейшее движение этой величины будет заблокировано, поскольку на вход соответствующего смесителя подается управляющий токен FALSE. Таким же образом окажутся заблокированными величины 2 и 4, поступающие соответственно на правый и средний смесители.

Описанный способ представления программы позволяет достичь максимального параллелизма. В частности, в этой простой программе он составил по тактам: 5 (см. рис. 3.15, а), 3 (см. рис. 3.15, б - г), 2 (см. рис. 3.15, д).

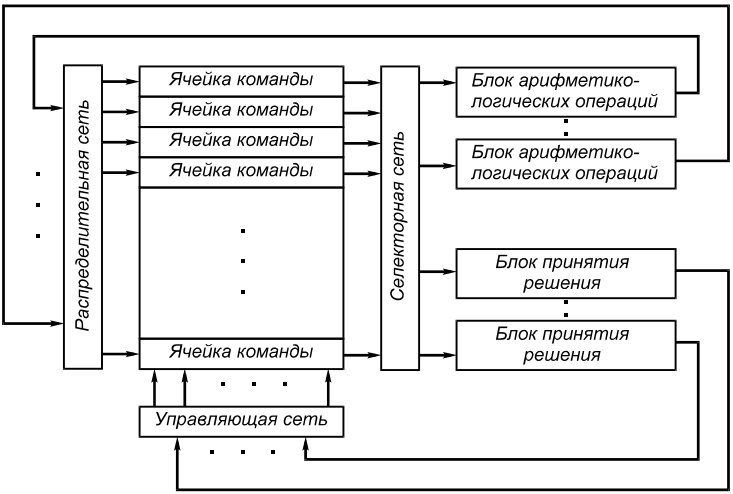


Рис. 3.16. Структура машины Дениса

В качестве примера машины, реализующей принципы DF-ЭВМ, опишем машину Денниса (рис. 3.16). Она состоит из трех секций. К первой относится память с ячейками команд. Эти ячейки являются пассивной памятью, но в них содержатся и некоторые логические схемы. Блоки арифметико-логических операций вырабатывают данные, а блоки принятия решений — управляющую информацию. Машина Денниса может содержать произвольное число блоков обоих типов, числом которых и определяется ее реальный параллелизм. Эти блоки составляют вторую секцию. Третья секция состоит из трех переключающих сетей. Селекторная сеть извлекает готовую к выполнению команду и направляет ее в соответствующий блок операций или блок принятия решения. Распределительная сеть принимает пакет результата (данные и адрес назначения) и направляет его в указанную ячейку команды. Точно так же управляющая сеть передает управляющий пакет (результат принятия решения и адрес назначения) в указанную ячейку команды.

Вопрос представления в DF-ЭВМ структур данных (массивы, графы) достаточно сложен. Один из вариантов его решения предусматривает введение дополнительной памяти для хранения структур данных, а также отдельных блоков для их обработки. Запуск

на обработку структуры данных выполняется командой потокового типа, а обработка больших массивов производится методами, близкими к традиционным.

Команда такой машины соответствует одному из элементов, представленных на рис. 3.15 и содержит код операции; блок состояний, описывающий наличие операндов; сами операнды; адреса, по которым нужно разослать результат.

Этот язык имеет достаточно низкий уровень и не пригоден для написания больших программ, поэтому разрабатывается ряд языков с традиционной формой представления операторов. Символьный ЯВУ можно получить и на основе языка Денниса, для этого необходимо исполнительные элементы (см. рис. 3.13) представить в виде операторов ЯВУ. Блок выполнения операций при этом отображается в обычный оператор присваивания; блок принятия решения совместно со смесителем преобразуются в подобие оператора: IF (условие) THEN X: = X1 ELSE X: = X2. Несколько сложнее с размножителями данных, однако эти и другие элементы графического языка все же могут быть заменены обычными или специально введенными операторами символьного типа.

На таком символьном ЯВУ программа (см. рис. 3.14) будет изображаться в виде списка операторов, причем порядок следования операторов в списке может быть произвольным и ни в коей мере не определяет порядок вычислений.

Чтобы повысить уровень программирования в символьных ЯВУ для потоковых машин, некоторые устойчивые совокупности операторов, используемые, в частности, для организации циклов, заменяются одним оператором, которым и пользуется программист.

Порядок вычислений в потоковой символьной программе определяется связями между операторами и готовностью данных на линиях этих связей. Чтобы правильно преобразовать графическую программу (см. рис. 3.14) в символьную, необходимо всем связям на рис. 3.14 присвоить уникальные, неповторяющиеся имена и эти имена перенести в операторы. Тогда символьная программа по связям будет строго соответствовать графической. Присвоение уникального имени называется в потоковых машинах *принципом однократного присваивания* и используется практически во всех графических и символьных языках.

§ 3.4. Надежность и отказоустойчивость параллельных ЭВМ

Параллельные ЭВМ, в частности, многопроцессорные, наряду с высоким быстродействием естественным образом обеспечивают высокую надежность и отказоустойчивость вычислений. Это достигается за счет многоэлементности (многопроцессорности) и способности к реконфигурации этих ЭВМ. В этом параграфе надежность и отказоустойчивость рассматриваются преимущественно в отношении оборудования, а не программного обеспечения.

Приведем некоторые сведения из теории надежности, необходимые для дальнейшего рассмотрения [18].

Надежность определяется как свойство технического изделия выполнять заданные функции в течение требуемого времени.

Основным понятием теории надежности является **отказ**, под которым понимают случайное событие, нарушающее работоспособность изделия. Применительно к вычислительной технике различают два вида отказов: устойчивые (собственно отказы) и самоустраняющиеся (сбой, перемежающиеся отказы). Сбой возникает вследствие одновременного неблагоприятного изменения нескольких параметров и существует кратковременно. Перемежающиеся отказы могут возникать, например, при плохом контакте в соединителе. Сбои встречаются в ЭВМ наиболее часто.

Основными параметрами надежности для невосстанавливаемых изделий являются ***интенсивность отказов*** λ, вероятность безотказной работы $P(t)$, среднее время безотказной работы T:

$$\lambda = \frac{m}{N \cdot t}$$

$$P(t) = e^{-\lambda t}$$

$$T = \int_0^{\infty} P(t)dt = \frac{1}{\lambda},$$

Здесь m - число изделий, отказавших за время t, а N — число исправных элементов на начало промежутка времени. Следова-

тельно, λ определяет долю (а не количество) изделий, отказавших в единицу времени, в качестве которой обычно принимают один час. Интенсивность принимается постоянной на этапе нормальной эксплуатации изделий.

Для микросхем обычно $\lambda=10^{-6}...10^{-8}$ 1/час, поэтому среднее время безотказной работы большое: $T=10^{6}...10^{8}$ часов. Но в ЭВМ обычно входят тысячи микросхем. Поскольку

$$\lambda_{\text{ЭВМ}} = \sum_{i=1}^{N} \lambda_i ,$$

где $\lambda_{\text{ЭВМ}}$, λ_i — интенсивность отказов ЭВМ и микросхемы соответственно, а N — число микросхем в составе ЭВМ, то время безотказной работы ЭВМ может составлять всего десятки или сотни часов.

Для восстанавливаемых систем наиболее важным параметром является наработка на отказ t_{cp}. Если восстановление является полным, то время наработки на отказ t_{cp} восстанавливаемой системы соответствует времени безотказной работы T невосстанавливаемой системы.

По назначению вычислительные системы, к которым относятся и многопроцессорные ЭВМ, можно разделить на две группы: системы для управления в реальном масштабе времени и информационно-вычислительные системы. Для первых отказ может вызвать тяжелые последствия, поэтому одним из основных показателей эффективности таких систем является надежность. Вторые не критичны к отказу, основным показателем их эффективности является производительность.

Рассмотрим показатели надежности ЭВМ, применяемых для систем управления в реальном времени. Как говорилось выше, наработка на отказ ЭВМ невысока. Основной способ повысить ее — ***резервирование***. Различают два вида резервирования: общее и поэлементное. При общем резервировании резервируется вся ЭВМ, т. е. в случае выхода из строя она заменяется такой же. При поэлементном резервировании резервируются отдельные части ЭВМ (процессоры, каналы) и в случае отказов они заменяются идентичными. Наиболее употребительным является постоянное (горячее) резервирование, при котором резервное устройство выполняет ту же нагрузку, что и основное, и при отказе последнего

резервное устройство без задержки замещает основное. Число резервных устройств может быть более одного.

Общее резервирование значительно повышает надежность системы, в частности для системы без восстановления вероятность безотказной работы $P_{рез}(t)$ равна:

$$P_{рез}(t) = 1 - \left[\left(1 - P(t)\right)\right]^m,$$

где $P(t)$ — вероятность безотказной работы одной ЭВМ, а m — число ЭВМ в системе. Так, если $P(t)$ = 0,9, а m = 2 (используется дублирование), то $P_{рез}(t)$ = 0,99. Для восстанавливаемых систем и при m = 3...4 резервирование позволяет достичь характеристик, близких к идеальным.

Однако, резервирование неэкономично, так как объем оборудования возрастает в m раз, а производительность остается на уровне одной ЭВМ. Такую систему нельзя назвать и параллельной, поскольку в последней каждая ЭВМ выполняет независимую работу. Значительно эффективнее поэлементное резервирование, которое прямолинейно реализуется структурой многопроцессорной ЭВМ.

Пусть многопроцессорная ЭВМ содержит m одинаковых процессоров, l из которых являются избыточными. Избыточный процессор полноценно заменяет любой из основных в случае отказа последнего, то есть реализуется плавающее резервирование. Каждый из основных процессоров выполняет независимую часть работы.

Пусть процессор имеет интенсивность отказов λ и интенсивность восстановления μ ($\mu = 1/T_\mu$, где T_μ — среднее время восстановления процессора). В этом случае средняя наработка на отказ многопроцессорной системы с l резервными элементами:

$$t_{ср,рез} = \frac{1}{(m-e)\cdot\lambda} \sum_{i=0}^{l} \frac{(m-l)!}{(m-l+i)!} \left(\frac{\mu}{\lambda}\right)^i$$

Так как $1/(m\lambda)$ — наработка на отказ неизбыточной ЭМВ, то при малых l выигрыш надежности избыточной ЭВМ будет:

$$G = \frac{t_{ср,рез}}{t_{ср}} = \sum_{i=0}^{l} \frac{(m-l)!}{(m-l+i)!} \left(\frac{\mu}{\lambda}\right)^i$$

В практически важных случаях $\mu/\lambda >> 1$, тогда $G \approx [\mu/(m\lambda)]^l$. Для ЭВМ обычно $\mu/(m\lambda) \geq 100$, то уже при $l=2$ или 3 наработка на отказ избыточной ЭВМ будет приближаться к границе долговечности.

Система с плавающим резервированием должна обладать способностью к *реконфигурированию*.

Все вышеприведенное относилось к системам управления реального времени, которые можно определить как двухпозиционные системы: работает — не работает. Именно для таких систем основной характеристикой функционирования является надежность.

Информационно-вычислительные системы являются многопозиционными. Функции, выполняемые этими системами, можно разделить на основные и второстепенные. Если отказ одного из элементов делает невозможным выполнение одной из второстепенных функций, это не препятствует дальнейшему функционированию системы. Если же отказ затрагивает исполнение основной функции, то в результате автоматической реконфигурации выполнение этой основной функции передается на оставшиеся работоспособные элементы (возможно, с вытеснением второстепенных функций). В таких системах возможен отказ более чем одного элемента при сохранении работоспособности системы. После восстановления отказавших элементов система будет выполнять функции в полном объеме. Восстановление производится в процессе функционирования системы.

Системы, обладающие вышеуказанными свойствами, называют отказоустойчивыми (толерантными, “живучими”).

Очевидно, для отказоустойчивых систем не подходит понятие отказа в вышеописанном случае, а надежность не является основным показателем эффективности. В таких системах отказом следует считать такой отказ элементов системы (в общем случае многократный), который вызывает поражение одной из основных функций. Что касается эффективности, то для информационно-вычислительных систем одним из основных ее показателей явля-

ется производительность, которую можно определить выражением:

$$V_c = V \cdot \sum_{i=1}^{m} i \cdot p_i ,$$

где V_c — средняя производительность системы, V — производительность одного элемента системы (процессора), m — максимальное число элементов в системе, i — число элементов в конфигурации, p_i — вероятность этой конфигурации.

Системы с восстановлением имеют существенно лучшее распределение вероятностей p_i, чем системы без восстановления.

Частным случаем отказоустойчивых систем являются системы с постепенной деградацией, в которых ремонт и восстановление невозможны или по каким-либо причинам нецелесообразны. Такие системы продолжают функционировать до тех пор, пока число работоспособных элементов не достигнет минимально допустимого уровня.

Центральным качеством надежных и отказоустойчивых систем является автоматическая реконфигурация. Реконфигурация возможна, если в многопроцессорной ЭВМ имеются аппаратные и программные средства *контроля* и диагностики, реконфигурации и повторного запуска системы после отказа (рестарта).

Средства контроля могут быть реализованы аппаратным, программным или смешанным способом. Аппаратные средства контроля предназначены для контроля передачи информации и контроля правильности выполнения арифметико-логических операций.

Контроль путей пересылки информации (память — АЛУ, память — ВнУ и др.) производится на основе избыточного кодирования (коды Хэмминга, Грея и др.). Особенное распространение имеет простой код с проверкой четности, требующий только одного дополнительного разряда на блок двоичных разрядов (обычно восьми). Эти же методы и дублирование блоков используется для контроля операций в АЛУ. Аппаратный контроль выполняется в темпе основных вычислений, что и является основным его достоинством.

Программный контроль не требует дополнительного оборудования, однако он выполняется с задержкой во времени. К программным методам контроля относятся тестовый и программно-логический контроль. Тестовый контроль выполняется периодически, либо при наличии свободного времени в ЭВМ, либо после обнаружения отказа другими средствами. В этом случае тестовый контроль частично выполняет функции диагностики. Примерами программно-логического контроля являются: двойной просчет, решение задачи по основному и упрощенному алгоритму, проверка предельных значений переменных и др.

Аппаратной основой реконфигурации является наличие развитой системы коммутации, позволяющей исключать из структуры многопроцессорной ЭВМ неисправные элементы в случае отказов и устанавливать новые связи между исправными элементами. Разработано большое количество типов коммутаторов. Наиболее перспективными являются коммутаторы типа многомерный куб. Следует отметить, что коммутаторы являются неотъемлемыми элементами как параллельных, так и отказоустойчивых ЭВМ. Более подробно коммутаторы описаны в § 2.3.

К системному и прикладному программному обеспечению отказоустойчивой ЭВМ предъявляются следующие требования:

1. Вся адресация памяти, ВнУ, каналов должна выполняться на логическом уровне. В этом случае отказ элемента приводит только к изменению таблиц связи логических и физических адресов.

2. Операционная система должна носить распределенный характер, то есть в процессорах должны находиться копии ОС или ее частей.

В наибольшей степени этим условиям удовлетворяют параллельные ЭВМ типа МКМД с децентрализованным управлением.

Для обеспечения рестарта в процессе вычислений необходимо создавать контрольные точки, то есть запоминать результаты вычислений. Рестарт осуществляется с ближайшей контрольной точки. Чем чаще устанавливаются контрольные точки, тем меньше времени будет потрачено на повторение вычислений в процессе рестарта, однако создание контрольных точек также требует дополнительного времени.

Контрольные вопросы

1. Чем отличаются МКМД ЭВМ с общей памятью от МКМД ЭВМ с локальной памятью?
2. Определите понятия процесс, ресурс, синхронизация, назовите виды процессов.
3. Что такое критическая секция, семафор, занятое ожидание? Опишите операции над семафором.
4. Что такое дескриминаторы процессов и ресурсов? Каков их состав?
5. Охарактеризуйте основные операции над процессами.
6. Охарактеризуйте основные операции над ресурсами.
7. В чем состоит централизованное и децентрализованное управление в многопроцессорной ЭВМ?
8. Что такое транспьютер, какова его структура?
9. Назовите основные конструкции языка Оккам.
10. Охарактеризуйте средства описания параллелизма в языке Оккам.
11. Опишите структуру центрального процессора транспьютера и особенности его системы команд.
12. Как осуществляется обмен данными в системе с несколькими транспьютерами.
13. Определите основные понятия двухмерного графического языка для системы с управлением от потока данных.
14. Приведите пример программы на двухмерном графическом языке, опишите ее функционирование.
15. Объясните, что такое надежность и отказоустойчивость многопроцессорных ЭВМ.

Глава 4

СТРУКТУРЫ ПРОЦЕССОРОВ НА ОСНОВЕ СКАЛЯРНОГО ПАРАЛЛЕЛИЗМА И ДРУГИЕ ТИПЫ ПАРАЛЛЕЛЬНЫХ ПРОЦЕССОРОВ

§ 4.1. Скалярный параллелизм

Термин *скалярный параллелизм* соответствует параллелизму операций внутри тела цикла или отдельного выражения. Вместо термина “скалярный” часто используется термин “локальный па-

раллелизм", чтобы отличить его от "глобального параллелизма", соответствующего параллелизму итераций цикла или независимых ветвей программы.

Скалярный параллелизм определяется в границах *базового блока* (ББ), под которым понимают отрезок программы, не содержащий условных или безусловных переходов. ББ обычно содержит одно или несколько выражений ЯВУ.

Из закона Амдала:

$$r = \frac{1}{a + \frac{1-a}{n}}$$

следует, что значение ускорения r весьма чувствительно к удельной величине скалярной части вычислений a, поэтому в быстродействующих ЭВМ прикладывают значительные усилия для распараллеливания скалярных участков.

Результаты анализа программ показывают, что в одном такте можно выполнить 1,5...2 операции или команды. Комплексную команду, содержащую больше одной параллельной команды, называют длинной командой (ДК). Для увеличения скалярного параллелизма используется ряд методов, рассмотренных в § 5.5, например, развертка циклов. Если при этом число параллельных операций превышает 2, то такой параллелизм называется *суперскалярным*, а команда — *сверхдлинной* (СДК).

Величина скалярного параллелизма зависит от уровня представления программы. Параллелизм на уровне ЯВУ будем называть *математическим*, на уровне системы команд — *аппаратным*. Существует несколько уровней систем команд, и каждому уровню соответствует своя величина скалярного параллелизма. Рассмотрим основные уровни систем команд.

1. Исторически в больших ЭВМ и ПЭВМ наибольшее распространение получила *система команд типа CISC* (Complex Instruction Set Computer), содержавшая большое количество команд. Основой этой системы команд являлся формат

КОП R1, D2 (B2) (4.1)

где КОП представлял арифметико-логическую операцию в АЛУ, а второй адрес содержал обращение в память с базой или индексом *B*2 и смещением *D*2. Такая команда требовала многотактного выполнения.

2. Сокращенная *система команд типа RISC* (Redused Instruction Set Computer) более перспективна для параллельных процессоров, поскольку число команд в ней меньше и вследствие их простоты большинство команд можно реализовать аппаратно и выполнить за один такт.

Сокращение числа команд в *RISC*-системе достигается за счет разбиения формата (4.1) на два более простых формата:

Чт, Зп, Ri, D2 [B2] (4.2)
КОП Ri, Rj

Первый формат предназначен только для работы с памятью, второй — только для работы с АЛУ.

Если предположить, что имеется k кодов операций для работы с памятью и l кодов — для работы с АЛУ, то система команд типа *CISC* должна содержать $k*l$ команд формата (4.1), а система *RISC* — только $k+l$ команд. Следовательно, отказ от формата (4.1) значительно сокращает список системы команд.

3. Последующее упрощение системы команд достигается при переходе к *muny MISC* (Minimum Instruction Set Computer), в котором команда формата (4.2) разбивается на две команды:

Rk = B2 + D2 (4.3)
Чт, Зп, Ri, [Rk] (4.4)

где команда (4.3) выполняет индексную операцию, а команда (4.4) — упрощенное обращение в память.

При понижении уровня системы команд увеличивается число служебных операций (команд), которые можно выполнять параллельно, то есть растет скалярный параллелизм.

Скалярный параллелизм длительное время использовался для ускорения процессоров в ЭВМ типа ОКОД, затем в конвейерных ЭВМ типа CRAY. Скалярный параллелизм имеет особое значение для микропроцессорной техники по следующим причинам:

1. Скалярный параллелизм есть свойство последовательных программ и может быть выявлен автоматически сравнительно не-

сложными средствами. Таким образом, ЭВМ со скалярным параллелизмом не требует изменения традиционной системы последовательного программирования. Это обеспечивает массовое применение таких ЭВМ с точки зрения программирования.

2. Успехи микроэлектроники позволяют разместить на одном кристалле процессор ЭВМ с несколькими АЛУ, что предполагает массовость производства с точки зрения аппаратуры.

Таким образом, суперскалярные процессоры, характеризующиеся последовательным программированием и параллельным функционированием, являются массовым видом вычислительной техники. По классификации Флинна они относятся к классу ОКОД.

§ 4.2. Структура и функционирование конвейера

Параллелизм ассемблерного уровня реализуется двумя способами:

1. В виде конвейера (конвейера команд, арифметического конвейера или конвейера смешанного типа). Конвейер команд использует технический параллелизм, возникающий вследствие необходимости выполнять команду с помощью блоков различного функционального назначения (управление, память, АЛУ и др.).

Одновременная работа этих блоков позволяет параллельно обрабатывать сразу несколько смежных команд.

Арифметический конвейер использует разбиение арифме-

тической операции на логически законченные микрооперации. Например, команда сложения с плавающей запятой может быть разделена на следующие микрооперации: вычитание порядков, выравнивание мантисс, сложение, нормализация. Арифметический конвейер также позволяет параллельно обрабатывать несколько команд.

2. В виде многопроцессорной системы, где совокупность процессоров обрабатывает в каждом такте один ярус ЯПФ некоторого выражения. Обычно в каждом процессоре используется и конвейерная обработка. Поэтому в суперскалярном процессоре одновременно обрабатывается m ∗ n команд, где m — число этапов в конвейере, а n — число конвейеров; в каждом такте вырабатывается n результатов. Если в идеальном случае для конвейера принять:

$$\Delta t = t / m ,$$

где Δt — время одного такта, а t — время выполнения одной команды, то быстродействие конвейера будет:

$$V = m / \Delta t = \frac{m \cdot n}{t} . \qquad (4.5)$$

Таким образом, увеличение быстродействия суперскалярного процессора достигается как за счет увеличения числа конвейеров, так и за счет удлинения конвейера. При этом по возможности сначала нужно увеличивать число ступеней конвейера, так как для этого требуется меньше оборудования, чем для увеличения числа процессоров при достижении того же прироста быстродействия.

Следовательно, конвейер является основой суперскалярного процессора. В настоящем параграфе будут рассмотрены структура, функционирование, недостатки и способы получения максимального быстродействия одиночных конвейеров [19, 20].

Конвейеры можно разделить на две большие группы:

1. Векторные конвейеры (см. § 2.2), которые выполняют одну операцию над группами данных, называемых векторами. Такие конвейеры, как правило, являются арифметическими, то есть их ступени выполняют части арифметико-логических операций.

2. Скалярные конвейеры, в которых на разных ступенях обработки одновременно находятся команды с разными кодами операций. Скалярные конвейеры могут содержать только конвейер команд, но в процессорах с плавающей запятой часто скалярный

конвейер включает и арифметические ступени. Для выполнения векторных операций в скалярном конвейере необходимо на вход последнего в каждом такте подавать один и тот же код операции.

С точки зрения скалярного параллелизма интерес представляют преимущественно скалярные конвейеры. В качестве примера такого конвейера рассмотрим наиболее распространенный конвейер микропроцессора типа *CISC* (за прототип взят конвейер МП "*Пентиум*" из семейства МП 80 × 86).

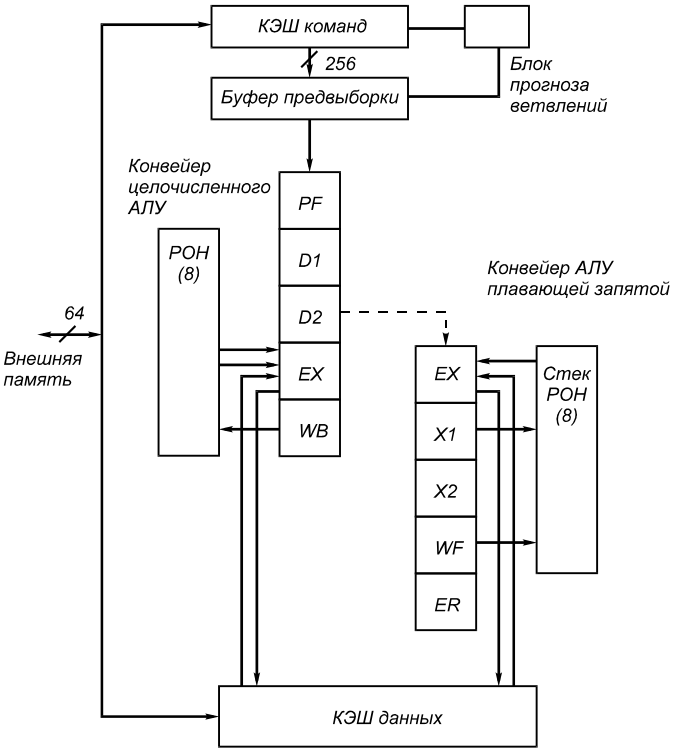


Рис. 4.1. Целочисленный и плавающий конвейер CISC-процессора

Целочисленный конвейер содержит следующие *ступени* (рис. 4.1):

- ступень предвыборки PF (Prefetch), которая осуществляет предварительную упреждающую выборку группы команд в соответствующий буфер;

- ступень декодирования полей команды D1 (Decoder 1);
- ступень декодирования D2 (Decoder 2), на которой производится вычисление абсолютного адреса операнда, если операнд расположен в памяти;
- на ступени исполнения EX (Execution) производится выборка операндов из РОН или памяти и выполнение операции в АЛУ;
- наконец, на ступени записи результата *WR* (Write Back) производится передача полученного результата в блок РОН.

Плавающий конвейер имеет общие с целочисленным конвейером ступени *PF*, *D1*, *D*2, но кроме того, имеет еще 5 дополнительных ступеней (переход показан пунктирной стрелкой):

- на ступени EX производится чтение из памяти или регистров или запись в память;
- на ступени X1 выполняется часть плавающей операции (фаза 1) или запись в плавающий РОН. Возможен обход (bypass) на ступени EX;
- на ступени X2 выполняется продолжение обработки (фаза 2);
- на ступени WF производится округление и обратная запись результата в блок РОН;
- на ступени ER (Error Reporting) выводится сообщение о наличии ошибок.

Наибольшая производительность конвейера достигается, когда каждый такт на вход поступает новая команда, а на выходе получается очередной результат.

Однако получению максимального быстродействия в *CISC*-конвейере препятствуют следующие обстоятельства:

1. Простои конвейера из-за наличия команд, которые требуют многотактного исполнения в АЛУ или на других ступенях конвейера.

2. Простои конвейера из-за зависимости по данным между соседними командами.

3. Простои конвейера из-за очистки и повторной загрузки конвейера в случае ошибки при предварительном выборе направления условного перехода.

4. Ограничения пропускной способности аппаратных средств: РОН, памяти различных видов, шин связи.

Рассмотрим влияние этих факторов на уменьшение быстродействия конвейера и некоторые способы нейтрализации этого влияния.

Многотактные команды. В таблице 4.1 представлено время нахождения некоторых типичных команд в целочисленном конвейере на этапе *EX*, поскольку на остальных этапах на выполнение команды затрачивается только один такт. (Здесь и далее рассматриваются команды семейства 80×86).

Таблица 4.1.

Длительность некоторых команд в целочисленном конвейере

N пп	Формат команды	Число тактов
1	LD R1, D[Ri]	1
2	ST D[Ri], R1	1
3	ADD R1, R2	1
4	ADD R1, [Ri]	2
5	ADD [Ri], R1	3
6	IMUL R1, R2	11

Команда 4 задерживается на этапе *EX* два такта, поскольку один такт тратится на обращение к памяти за операндом, а другой — на выполнение арифметической операции. Команда 5 занимает три такта из-за двукратного обращения к памяти. Команда 6 умножения занимает много тактов из-за большого числа выполняемых ею микроопераций.

Если конвейер выполняет только однотактные команды, то он продвигается каждый такт и достигается максимальная (пиковая) производительность, при этом темп выдачи результатов — один результат за такт. Если в конвейер попадают многотактные команды, то конвейер приостанавливается на время их выполнения, а производительность падает.

В таблице 4.2 приведены времена выполнения некоторых важных команд для конвейера плавающей запятой; в скобках ука-

зано время полного выполнения команды в АЛУ, а перед скобкой — темп получения результатов в потоке команд данного типа.

Таблица 4.2. Длительность некоторых команд в плавающем конвейере

N пп	Формат матрицы	Ступени					Итого
		EX	X1	X2	WF	ER	
1	FLD[M]	1	1	-	-	-	1
2	FST[M]	2	-	-	-	-	2(2)
3	FADD[M]	1	1	1	-	-	1(3)
4	FMUL[M]	1	1	1	-	-	1(3)
5	FDIV[M]						39
6	FSQRT						70

В плавающем конвейере *CISC*-процессора файлы РОН реализованы обычно в виде стека и операции выполнения на верхушке стека, поэтому в командах не указываются номера РОН, а только адрес ячейки памяти.

Из таблицы 4.2 следует, что большинство команд находятся в АЛУ (ступени *EX*, *X*1, *X*2) несколько тактов (время в скобках), однако на каждом этапе задерживаются только 1 такт.

Следовательно, темп выполнения в конвейере — 1 такт на команду (команды *FLD*, *FADP*, *FMUL*). Некоторые команды могут на одном этапе находиться несколько тактов, приостанавливая на это время конвейер.

Поскольку целочисленное (ступень *EX*) и плавающее АЛУ работают независимо, то прием команд на вход общей части конвейера приостанавливается только в следующих случаях:

1. Оба конвейера выполняют многотактные операции.

2. Один из конвейеров выполняет многотактную операцию, но на ступенях *D*1 или *D*2 находятся команды для этого же конвейера.

Зависимость по данным. Рассмотрим пример выполнения в процессоре с плавающей запятой следующего отрезка программы:

```
1  FLD [M1]
2  FMUL [M2]
```

3 FADD [M3] (4.6)
4 FST [M4]

в котором каждая команда размещает свой результат на верхушке стека, а последующая команда использует его как один из операндов. Это означает, что между смежными командами существует строгая зависимость по данным.

На рис. 4.2 представлена временная диаграмма выполнения этого отрезка программы.

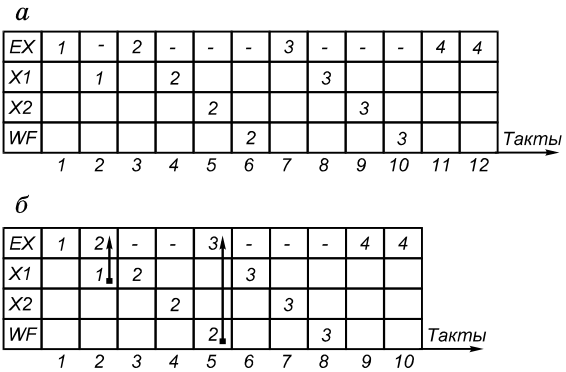


Рис. 4.2. Диаграмма работы конвейера

Из рис. 4.2, а следует, что ступень *EX* простаивает 7 тактов из 12:

1. В такте 1 ступень *EX* не работает, так как результат команды 1 является операндом команды 2, поэтому в такте 2 он записывается в стек и только в такте 3 может быть считан командой 2.

2. Аналогично команда 3 ожидает такты 4, 5, 6, пока результат команды 2 на ступени *WF* не окажется в стеке, и только в такте 7 может считать этот результат из стека.

3. Зависимость между командами 3 и 4 также дает такты простоя 8, 9 и 10 ступени *EX*.

Частично ситуация может быть исправлена благодаря механизму обходов, который использован в плавающем конвейере для уменьшения простоев. Благодаря обходу 1 результат операции *FLD*, который записывается в стек на ступени *X*1, может одновре-

менно быть послан прямо на ступень *EX* для использования следующей инструкцией.

Обход 2 позволяет послать результат арифметической операции со ступени *WF* на приемную ступень *EX* одновременно с записью этого результата в стек.

На рис. 4.2, б представлена диаграмма той же программы, но в такте 2 результат команды 1 одновременно с записью в стек передается на ступень *EX,* благодаря чему может быть начата команда 2. То же происходит и в такте 5. Команда 4 не может быть начата в такте 8, так как команда *FST* начинает работать только после записи результата в РОН предыдущей командой.

Таким образом, обходы оказали небольшую помощь (исключены 2 такта из 12).

Принципиальное решение вопроса об устранении простоя конвейеров из-за зависимости по данным состоит в том, чтобы размещать между зависимыми командами другие команды программы, не зависящие по данным друг от друга, от предшествующих и последующих команд. В этом случае программа (4.6) выглядела бы с независимыми командами (НК) следующим образом (временная диаграмма на рис. 4.3):

```
1  FLD [M1]
2  FMUL [M2]
3  НК
4  НК
5  FADD [M3]                                    (4.7)
6  НК
7  НК
8  НК
9  FST [M4]
```

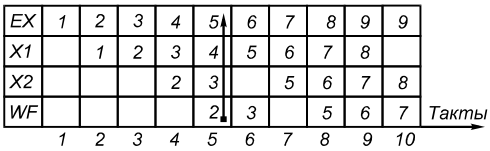


Рис. 4.3. Диаграмма выполнения программы с включением независимых команд

Программа (4.6) согласно рис. 4.2, б выполняется за 10 тактов, следовательно, архитектурная скорость будет равна:

$$z_1 = 4/10 = 0{,}4.$$

Программа (4.7) также выполнится за 10 тактов, но при этом согласно рис. 4.3

$$z_2 = 9/10 = 0{,}9.$$

Программа (4.7) реализует скалярную форму параллелизма в конвейерных процессорах. К сожалению, размещение независимых команд в программах с плавающей запятой в *CISC*-процессорах вызывает трудности из-за стековой организации файла РОН.

Условные переходы. Рассмотрим отрезок программы:

```
        ...
        1    ADD R1, R2
        2    ST [M], R1
        3    JZ LONG
        4    ADD R1, [M1]                    (4.8)
        5    MUL R3, [M2]
        ...
        ...
LONG    11   ADD R1, [M3]
        12   SUB R4, [M4]
        ...
```

Известно, что в конвейере адрес перехода определяется только на ступени *EX*. Но, чтобы не допустить простоев на этапах конвейера, в (4.8) необходимо непосредственно за командой перехода 3 начинать выборку одной из ветвей, начиная с команды 4 либо с команды 11.

Предположим, что принято решение выбирать ветвь с командой 4, и это решение оказалось неверным, тогда потребуется очистка конвейера и его повторное заполнение. На рис. 4.4, а показано, что в такте 5 определяется правильный адрес перехода на ступени *EX* и с такта 6 начинается перезагрузка конвейера. При этом потерянными оказались три такта.

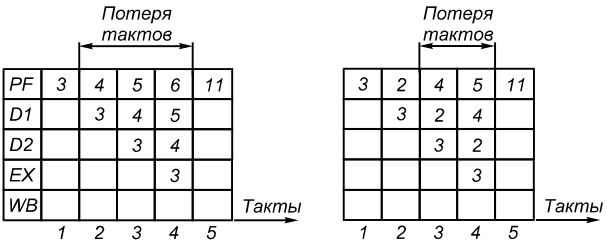


Рис. 4.4. Перезагрузка конвейера при ошибке выбора направления перехода

Потеря тактов будет больше, если команды 11 не окажется в буфере предварительной выборки.

Если учесть, что многие программы содержат от 10 до 20% команд условных и до 10% безусловных переходов, то правильная обработка ветвлений в конвейере оказывается весьма важной.

Методы управления переходами можно разделить на две группы: статические, выполняемые на этапе компиляции, и динамические, выполняемые аппаратурой конвейера в процессе выполнения программы.

К статическим методам относятся:

1. Исключение ветвлений в программе, если это возможно.

Для этого применяют: вычисление ветвлений на этапе компиляции; вынос ветвлений за пределы цикла; развертку итераций.

В приведенном ниже примере развертка позволяет в два раза сократить число ветвлений:

```
      DO1 I = 1, 100                DO 1 I = 1, 100, 2
1     A (I) = A (I) * 1.1           A (I) = A (I) * 1.1
                                1   A (I+1) = A (I+1) * 1.1
```

2. В программе переход назад обычно соответствует циклу и выполняется в 90% случаев, а переход вперед обозначает ветвление и выполняется в 50% случаев. На этапе компиляции определить это несложно, поэтому в машинной программе можно предполагаемое направление перехода пометить специальным битом перехода в команде.

3. Еще один метод называется отложенным ветвлением. В примере (4.8) команда 2 не влияет на условный переход, поэтому ее можно переставить после команды 3 и всегда выполнять после команды перехода независимо от направления перехода.

Программа будет выглядеть следующим образом:

```
        ...
        1     ADD R1, R2
        3     JZ LONG
        2     ST [M], R1
        4     ADD R1, [M1]
        5     MUL R3, [M2]
        ...
        ...
LONG    11    ADD R1, [M3]
        12    SUB R4, [M4]
        ...
```

На рис. 4.4, б показано, что в этом случае потеря тактов уменьшается. Таких отложенных команд может быть и больше одной, но для реализации отложенного ветвления требуется наличие в программе скалярного параллелизма.

В динамике управление переходами осуществляется следующими способами:

1. Самый простой способ состоит в том, чтобы остановить прием команд на вход конвейера до тех пор, пока команда перехода не достигнет ступени *EX* и не будет вычислено реальное направление перехода. Однако этот метод, упрощая управление, не решает проблемы простоев.

2. Наиболее сложный способ заключается в том, чтобы при появлении на входе команды перехода выбирать и условно выполнять обе ветви возможных переходов. Условное выполнение обозначает, что до момента определения адреса перехода запрещается запись в регистры или память.

3. Наконец, третьим способом динамической обработки ветвлений является предсказание. Самым простым способом предсказания является выбор для выполнения той ветви, которая следует непосредственно за командой перехода (короткий переход). Но в 50% случаев этот выбор будет ошибочным.

Более эффективной является система предсказаний на основе истории процесса вычислений.

На рис. 4.1 присутствует устройство прогнозирования ветвлений. Оно содержит таблицу достаточно большого размера, например, на 256 строк. В каждой строке таблицы записан для выполнения части программы:

1. Адрес команды перехода.

2. Адрес дальнего перехода.

3. Бит “истории”, который указывает, по какому направлению произошел переход при последнем использовании команды перехода.

Буфер упреждающей выборки обычно содержит от нескольких до нескольких десятков предварительно выбранных команд, чтобы сгладить задержки, связанные с обращениями в память. Одновременно с подачей команд на вход конвейера устройство управления производит в буфере предварительной выборки просмотр вперед выбранных команд. Если при просмотре обнаружена команда перехода, то по таблице “истории” определяется направление перехода.

Устройство управления функционирует на основе предположения, что при повторном выполнении одной и той же команды перехода переход будет осуществлен по одному и тому же адресу. В соответствии с этим в буфер упреждающей выборки выбирается ветвь, предписанная битом “истории”, если этой ветви в буфере упреждающей выборки еще нет.

Буфер упреждающей выборки содержит две зоны: для текущей и альтернативной ветви и переключение с зоны на зону не вызывает простоев.

Описанный механизм ветвления позволяет выбирать правильные пути ветвления с вероятностью более 80%.

КЭШ. Фактором, также ограничивающим быстродействие конвейера, является недостаточное быстродействие ресурсов, в частности, памяти. Рассмотрим некоторые требования к памяти.

Чем меньше такт конвейера, тем более быстрой должна быть память, обслуживающая этот конвейер. Если учесть, что цикл внешней памяти обычно больше, чем длительность такта конвейера, то становится очевидным, что между конвейером и внешней памятью должны быть расположены вспомогательные средства

памяти различного быстродействия и назначения и усовершенствованная система шин.

Регистры общего назначения являются наиболее быстрым видом памяти, их число колеблется от 8 до 32. РОН используется для хранения информации, как правило, локальной для тела цикла или базового блока. Чтобы избежать конфликтов при обращении к РОН, их делают многопортовыми (многовходовыми). Число портов обычно равняется трем: два для выдачи операндов и один — для приема результата.

Следующий шаг для уменьшения задержки при обращении к памяти состоит в разделении памяти на два независимых блока: память для программ и память данных. Такое разделение носит название гарвардской архитектуры. Для нее характерно использование буфера упреждающей выборки (рис. 4.1), в который за один такт принимается сразу несколько команд из памяти с большой разрядностью ячейки. Этот же буфер позволяет выбирать заранее альтернативные ветви в случае команд условных переходов. Все это сглаживает скорость поставки команд в конвейер.

Наиболее эффективным средством ускорения обращения к памяти является локальная (для АЛУ) память небольшого размера, называемая КЭШ.

Известно, что 90% обращений в память производится в ограниченную область адресов. Эта область называется рабочим множеством, которое медленно перемещается в памяти по мере выполнения программы. Для рабочего множества можно сделать промежуточную память небольшого размера, а значит, в несколько раз более быструю, чем основная память.

Для однокристальных микропроцессоров особенно важно то, что память такого размера можно разместить внутри кристалла, благодаря чему исчезают потери времени на вывод данных из кристалла, вызываемые малым числом выводов, задержками сигнала на контактах и из-за больших расстояний на плате.

Введем понятие строки и отображения [19]. Строка есть базовая единица информации, которая перемещается между основной и КЭШ-памятью. Следовательно, основная память состоит из большого числа строк с последовательными адресами, а в КЭШ-памяти понятие “последовательные адреса” отсутствует. В КЭШ-памяти каждое слово сопровождается адресным тэгом, указываю-

щим, какую строку основной памяти представляет данная строка КЭШ-памяти.

Отображением называется способ размещения и выборки строк основной памяти из КЭШ-памяти.

Можно выделить четыре основных типа отображения:

1. Полностью ассоциативная КЭШ-память. Любая строка основной памяти может находиться в любой строке КЭШ-памяти. Каждая строка КЭШ-памяти содержит свой компаратор (устройство сравнения). Если процессору нужна строка с определенным адресом, он должен искать ее путем сравнения адреса с тэгом всех строк КЭШ-памяти. Если такое сравнение делать последовательно, то КЭШ-память теряет смысл из-за низкого быстродействия; если сравнение делать одновременно со всеми тэгами, то такая память из-за ее сложности не может иметь большой объем.

КЭШ-память ассоциативного типа обычно имеет небольшой объем и используется для вспомогательных целей.

2. Противоположной структурой является память с прямым отображением. Одна из возможных реализаций использует разрядное отображение. Например, если в адресе строки памяти содержится N разрядов, то младшие n из них выбирают, в какую строку КЭШ-памяти она может копироваться. Следовательно, все строки с одинаковыми младшими адресами попадают в одну и ту же строку КЭШ-памяти. Оставшиеся N-n разрядов используются как адресный тэг для сравнения с адресом, выставленным процессором, чтобы убедиться, имеется ли данная строка в КЭШ или отсутствует.

Основное достоинство состоит в том, что накопитель такой памяти имеет структуру обычной прямоадресуемой памяти и нужен всего один компаратор, следовательно, такая КЭШ-память может иметь большую емкость.

Основной недостаток состоит в том, что в КЭШ-памяти может находиться ограниченное число комбинаций строк. Например, в КЭШ-памяти не могут одновременно находиться строки с одинаковыми младшими адресами. Это заметно увеличивает число промахов.

3. Промежуточным между полностью ассоциативным и прямым отображением является секторное отображение, вариант которого описан в § 2.1. В этом случае в КЭШ-памяти располагаются

страницы основной памяти. При обращении в КЭШ-память базовые адреса страниц в КЭШ-памяти устанавливаются с помощью служебной полностью ассоциативной памяти небольшого размера, а поиск слов в странице КЭШ-памяти производится на основе прямого адресного доступа. Секторная память удобна для больших ЭВМ, где программа может целиком располагаться в одной или нескольких страницах КЭШ-памяти. Для суперскалярных процессоров, реализуемых на одном кристалле, такой подход трудно реализовать, поскольку внутрикристальная КЭШ-память реально имеет объем менее одной страницы.

4. Другим промежуточным отображением, которое в основном и используется в однокристальных суперскалярных микропроцессорах является КЭШ-память с *множественно-ассоциативным доступом*. В ней, как и в памяти с прямым отображением, используется выбор строки КЭШ-памяти по младшим разрядам адреса основной памяти, но в этой строке КЭШ-памяти содержится несколько слов основной памяти, выбор между которыми производится на основе ассоциативного поиска.

На рис. 4.5 представлен КЭШ с двукратной ассоциацией. Это означает, что по одному смещению выбирается строка, содержащая информацию по двум разным страницам памяти. Это увеличивает вероятность удачного обращения. Обычно кратность ассоциации колеблется от единицы до четырех. Обычная длина поля тэгов до 24 или 32 разрядов, длина строки данных (одной) — от 4 до 32 байтов; число строк в КЭШе — до 256.

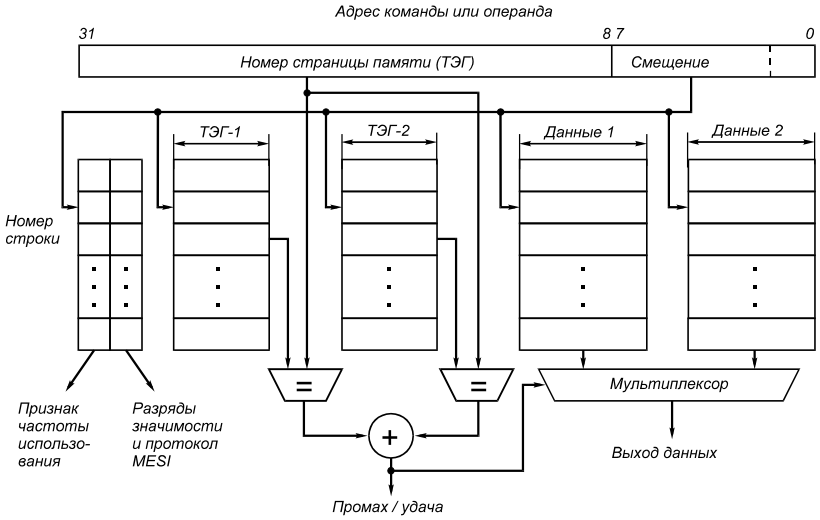


Рис. 4.5. Структура множественного ассоциативного КЭШа

Важной величиной является число портов в КЭШе. Как правило, число адресных портов (входы тэга) составляет 2...3, а порт данных — единственный. Объем внутрикристального КЭШа в МП колеблется в пределах от 4 до 32 килобайт, а внешнего — составляет несколько сотен килобайт.

На рис. 4.6 представлена зависимость числа попаданий от размера КЭШ-памяти с множественно-ассоциативным доступом. Из рисунка следует, что, если строка КЭШ-памяти содержит более четырех строк основной памяти, то это уже увеличивает степень попаданий незначительно.

Переход к RISC. Ранее было показано, что вследствие наличия многотактных команд, зависимости по данным, влияния условных переходов и аппаратных ограничений нельзя достичь предельной для конвейера архитектурной скорости 1 команды за такт. В рамках архитектуры *CISC*-процессора можно осла бить влияние указанных выше причин. В частности, для умень-

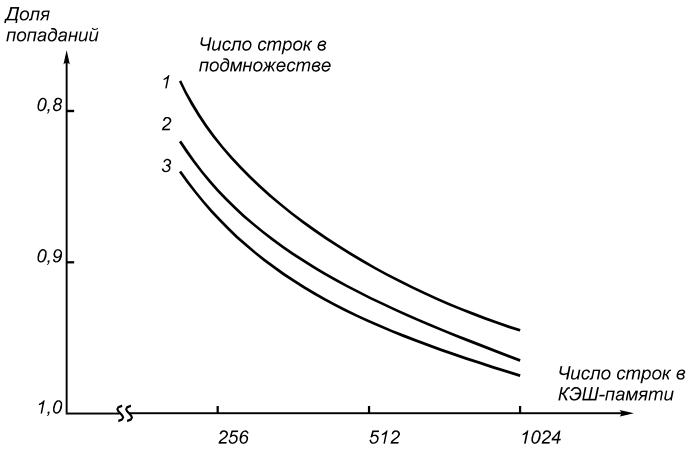


Рис. 4.6. Влияние характеристик КЭШ-памяти
на вероятность попадания

шения влияния зависимости по данным выше были описаны аппаратные обходы и введение независимых команд; для ослабления влияния условных переходов предложены различные методы предсказания; для повышения пропускной способности памяти предложена иерархия запоминающих устройств и шинные системы с высокой пропускной способностью.

Однако проблема многотактных команд может быть решена только при переходе от *CISC* к *RISC* и *MISC* системам команд.

Этот переход позволяет:

1. Сократить систему команд за счет устранения сложных команд.

2. Увеличить количество РОН; использовать для операций АЛУ трехадресные команды, ввести файлы РОН с произвольным доступом вместо стека для плавающей запятой.

3. Использовать команды фиксированного формата.

Рассмотрим эти средства несколько подробнее.

Переход к системе команд типа *RISC* приводит к разрешению системы команд на две группы: одна — для работы с АЛУ, другая — для работы с памятью. Такое разделение приводит к *Load/Store архитектуре*, представленной на рис. 4.7.

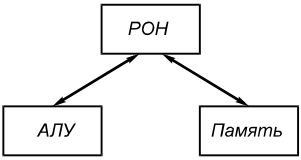


Рис. 4.7. Load-Store архитектура RISC-процессора

В такой структуре АЛУ и память работают параллельно. Чтобы обеспечить максимальное быстродействие системы, необходимо обеспечить упреждающую работу памяти.

Число РОН в *RISC*-процессоре достигает величины 32, 64 и даже более, в то время как в *CISC* оно не превосходит 8 или 16. Увеличение числа РОН уменьшает число обращений к памяти, сокращает длину программы, позволяет устранить некоторые зависимости по данным.

Трехместный (неразрушающий) формат команды выбран по следующим причинам: уменьшается число обращений к памяти (при достаточном числе РОН), устраняются излишние зависимости по данным. Например, в программе:

R3 = R1 + R2
R1 = R4 + R5

существует зависимость обратного типа, а в программе

R3 = R1 + R2
R6 = R4 + R5

использующей большее число регистров, она отсутствует, поэтому команды могут выполняться параллельно.

В *RISC*-процессорах РОН плавающей запятой организованы в виде файла с произвольным доступом, а не в виде стека, как в *CISC*. Это устраняет зависимости между смежными командами, вызванные единственным ресурсом — верхушкой стека, и уменьшает число обращений к памяти.

Препятствием к однотактной реализации команд *CISC* является последовательная структура дешифрации полей микрокоманды и переменная длина команды. Например, в системе команд МП *i*80×86 характер дешифрации полей команды последовательный:

по префиксу или первому байту определяется длина команды, после выборки оставшихся байтов производится дешифрация их полей. В отличие от этого в *RISC* команды имеют фиксированный формат, все поля независимы, поэтому дешифруются одновременно.

Все эти мероприятия позволяют выполнять большинство команд за 1 такт, при этом микропрограммирование перестает быть необходимым.

Таким образом, конвейер на базе системы команд типа *RISC* позволяет получить предельную для одиночного конвейера архитектурную скорость — 1 результат за такт.

Переход к новой системе команд типа *RISC* требует перепрограммирования громадного объема прикладного программного обеспечения, накопленного для широко распространенных систем команд типа *CISC*, используемых в МП *i*80×86, МС 680×80, поэтому естественным является стремление получить более высокую архитектурную скорость для *CISC*-МП на базе приближения свойства *CISC*-процессоров к свойствам *RISC*-процессоров. Для этой цели предпринимаются следующие действия программного и аппаратного характера:

1. Новые компиляторы *CISC*-процессоров строятся таким образом, чтобы для генерации машинных кодов из большого списка команд *CISC*-процессора использовать только небольшой набор простых команд, называемый *RISC*-ядро.

2. Ограничение на количество РОН в *CISC*-процессорах (всего 8 в *i*80×86) преодолевается следующим образом. При проектировании нового кристалла для *CISC*-процессора предусматривается увеличенный файл РОН, например, 32, и эти РОН составляют виртуальный файл. В процессе выполнения команд каждому логическому РОН, описанному в команде, предоставляется свободный РОН из виртуального файла, благодаря чему, как было показано ранее, растет параллелизм исполнения.

3. Для приближения стека по способу функционирования к файлу РОН с произвольным доступом аппаратно реализуются быстрые команды адресного обмена между регистрами стека типа *FXCH*, благодаря которым верхушка стека в каждом такте предстает как новый РОН с требуемым номером.

§ 4.3. Структура суперскалярного процессора

В настоящем параграфе для иллюстрации особенностей построения суперскалярных систем в качестве базовой используется архитектура суперскалярного процессора Пентиум фирмы Intel с системой команд типа *CISC* [20]. Вызвано это следующими причинами:

- *суперскалярная организация процессора* Пентиум достаточно проста для изучения, однако в ней отражены все особенности построения суперскалярных систем;
- система команд процессора Пентиум практически совпадает с системой команд широко известного семейства микропроцессоров *i*80×86, поэтому примеры программ легко воспринимаются;
- наконец, в процессоре Пентиум на программном и аппаратном уровне принято ряд решений, приближающих его архитектуру к архитектуре *RISC*-процессоров, позволяющих, однако, сохранить программную совместимость с большим объемом ранее написанного прикладного матобеспечения.

Появление в структуре процессора более одного конвейера делает этот процессор суперскалярным.

Как правило, в суперскалярных процессорах в первую очередь увеличивают количество целочисленных конвейеров, так как статистика показывает, что в обычных пакетах прикладных программ для ПЭВМ около 80% команд — целочисленные, 15% — команды условных переходов, и только небольшой процент команд является командами с плавающей запятой.

Немедленно после введения второго или большего числа конвейеров возникают принципиально важные для суперскалярной структуры вопросы по организации и технической реализации вычислений:

1. Как организовать запуск параллельных команд во множественных конвейерах. Нетрудно видеть, что здесь решается и вопрос о распараллеливании последовательных программ.

2. Как уменьшить отрицательное влияние на работу конвейеров зависимости по данным для смежных команд и потери быст-

родействия из-за появления в программе условных переходов. Планирование межконвейерного параллелизма должно производиться с учетом указанных эффектов.

3. Как обеспечить повышенную пропускную способность всех видов памяти в суперскалярном процессоре, особенно если число одновременно выполняемых операций составляет 20-30.

4. Каковы в перспективе предельные возможности, размеры суперскалярных процессоров.

Организация параллельного запуска команд предусматривает выполнение трех этапов: распараллеливание последовательной программы; планирование порядка выполнения команд для заданных ресурсов; представление спланированной программы в форме, пригодной для исполнения аппаратурой.

В зависимости от способа реализации этих этапов методы формирования и запуска параллельных команд можно разделить на статические, динамические и смешанные. Считается, что статические способы предпочтительнее, поскольку анализ программы в процессе компиляции обеспечивает более глубокое выделение и планирование параллелизма, кроме того, исключает служебные команды управления параллелизмом из вычислительного процесса [4]. Динамический же запуск позволяет более полно учитывать текущее состояние программы и ресурсов при исполнении программы.

Возможны следующие способы статического формирования параллельной команды:

1. Производится распараллеливание программы обычными методами, затем параллельные команды с помощью оптимизирующего планировщика упаковываются в виде СДК фиксированного формата. В таком СДК каждое поле формата закреплено за определенным конвейером и хранится в одной ячейке параллельной памяти. При чтении СДК из памяти выбираются все поля СДК независимо от их заполнения. Конвейеры запускаются синхронно в каждом такте. Это является возможным, поскольку в *RISC*-конвейерах основные операции выполняются за 1 такт. Основной недостаток схемы с фиксированным форматом СДК — неполное использование памяти при отсутствии команд в некоторых позициях СДК.

2. Более полное использование ячейки памяти фиксированного формата применяется в МП *i*860 (рис. 4.8).

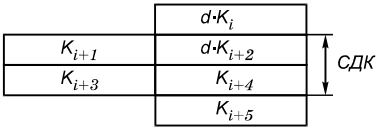


Рис. 4.8.Представление СДК в памяти МП *i*860

В МП *i*860 только два конвейера, поэтому СДК имеет длину 2. Если в некоторой команде, например, K_i указан признак *d* (*dual*-двойной), это означает, что следующие две команды по списку, а именно, K_{i+1} и K_{i+2} образуют параллельную пару. Физически эта пара может быть расположена либо в одной ячейке параллельной памяти, либо в смежных ячейках последовательной памяти.

3. Описанные выше способы годятся для вновь разрабатываемых систем команд, поскольку на этапе разработки можно заложить средства, обеспечивающие объединение команд в СДК фиксированного формата. Однако эти способы нельзя применить к системам команд *CISC*-процессоров, которые нельзя "подкорректировать", поскольку для них уже существует большой объем написанного математического обеспечения.

В этом случае используются элементы смешанного формирования и запуска СДК. Рассмотрим это на примере суперскалярного МП Пентиум с двумя целочисленными конвейерами *U* и *V*, использующего систему команд *i*80×86 и имеющего те же ступени, что и на рис. 4.1.

Пусть имеется отрезок программы на ЯВУ и соответствующий ему ассемблерный вариант:

```
    DO 1 I = 1, 10          M:  mov edx, [eax + 40 + a]
    A (I) = A (I) + 1           mov ecx, [eax + 40 + b]
1   B (I) = B (I) + 1           inc edx
                                inc ecx
                                mov [eax + 40 + a], edx
                                mov [eax + 40 + b], ecx
                                jnz M
```

В этой программе команды расположены последовательно, никаких отметок о параллелизме команд нет, в МП Пентиум также отсутствует СДК фиксированного формата. Следовательно, программа для этого МП оформлена в памяти стандартным для всех МП 80×86 образом. Однако она существенно отличается от программ для МП 386/486, не являющихся суперскалярными. Отличие состоит в том, что на этапе компиляции скалярный параллелизм выявлен и команды переставлены так, что параллельные команды в программе являются смежными.

На ступени *D*1 целочисленных конвейеров аппаратно при последовательной выборке команд из памяти определяется текущий параллелизм. Проверка параллельности двух команд производится с учетом всех видов зависимостей по данным.

В результате потактного распараллеливания в конвейеры *U* и *V* будет загружена и исполнена следующая программа:

	Конвейер U	Конвейер V
M:	mov edx, [eax + 40 + a]	mov ecx, [eax + 40 + b]
	inc edx	inc ecx
	mov [eax + 40 + a], edx	mov [eax + 40 + b], ecx
	add eax, 4	jnz M

которая требует для своего исполнения всего 4 такта вместо 7 в исходном тексте.

Динамическое формирование и запуск СДВ используются в тех случаях, когда имеется большой объем прикладного матобеспечения в виде двоичных кодов и не представляется возможным произвести распараллеливание этих кодов с помощью программных распараллеливателей. Тогда задача распараллеливания возлагается на аппаратуру на этапе исполнения программ.

Рассмотрим следующие варианты динамического распараллеливания:

1. Покомандное формирование и запуск СДК в машинах типа CRAY, осуществляемое под управлением *табло* (scoreboard). Этот метод описан в § 2.1.

2. Пакетное формирование СДК по Флинну.

Суть алгоритма пакетного формирования и запуска СДК состоит в том, что из блока в *n* инструкций, обычно составляющих ББ или часть ББ, выбирается и немедленно исполняется *m* инст-

рукций. Во время их выполнения в исходный блок добавляется *m* новых инструкций и вновь повторяется поиск независимых инструкций. Таким образом, блок из *n* инструкций напоминает “скользящее окно”, перемещаемое по программе по мере ее исполнения, а *m* параллельных команд составляют СДК.

Нетрудно видеть, что анализ на независимость блока из *n* команд требует много времени. С этой целью перед анализом на независимость команды преобразуются в форму, представленную на рис. 4.9. Здесь поля *E* и *D* содержат биты, установленные на номерах входных и выходного регистров, необходимых для выполнения данной команды. Следовательно, сравнение двух команд на зависимость сводится к побитному сравнению полей *E* и *D*, что может быть сделано быстро, и выбор *m* независимых команд может быть выполнен за время выполнения предыдущей СДК.

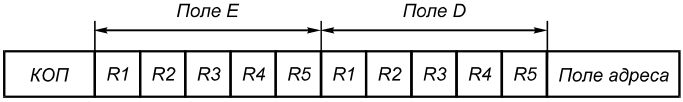


Рис.4.9. Модифицированный формат команды

Очевидно, что описанная схема, в отличие от ЭВМ CRAY, не требует предварительной перестановки команд ББ на этапе компиляции, что позволяет использовать традиционные компиляторы. Это является достоинством пакетной схемы. Однако большая длительность цикла формирования СДК аппаратурой снижает быстродействие процессора.

В качестве законченной структуры суперскалярного микропроцессора рассмотрим структуру МП Пентиум. Пентиум — это суперскалярный МП типа *CISC* семейства *i*80×86, следующий за МП 386 и 486. Он на 100% совместим на уровне двоичных кодов с МП 8086, 386*DX*, 486*DX*, 486*SX* и 486*DX*2. Структура Пентиума соответствует рис. 4.1, однако в нем содержится еще один целочисленный конвейер.

Таким образом, Пентиум содержит два целочисленных конвейера *U* и *V* и плавающий конвейер, входные ступени последнего (*PF*, *D*1, *D*2) являются общими с конвейером *U*. Назначение ступеней, особенности конвейеров описаны ранее в параграфах 4.2 и

4.3. При работе с плавающей запятой конвейеры *U* и *V* объединяются, создавая 64-разрядный обрабатывающий тракт.

Пентиум может запускать одну или две инструкции каждый такт. Чтобы запускать две инструкции одновременно, необходимо, чтобы они не содержали зависимостей по данным, были целочисленными и “простыми”. Простые команды выполняются за один такт. Исключением являются команды с обращением к памяти типа *mem*, *req* и *reg*, *mem*, которые требуют два и три такта соответственно, однако имеется специальная аппаратура, чтобы позволить этим командам выполняться, как простым.

Простыми являются следующие команды:

1. mov reg, reg/mem/imm
2. mov mem, reg/imm
3. alu reg, reg/mem/imm
4. alu mem, reg/imm
5. inc reg/mem
6. dec reg/mem (4.9)
7. push reg/mem
8. pop reg
9. lea reg, mem
10. jmp/call/jcc near
11. nop

Кроме того, команда межрегистрового обмена *FXCH* может образовывать пары с другими плавающими командами. Это сделано, как уже говорилось выше, чтобы ослабить ограничения на параллельную обработку, вызванные стековой организацией файла РОН плавающих регистров.

Быстродействие суперскалярного процессора в значительной степени определяется стилем программирования. Из этого следует, что существующие компиляторы для ЭВМ, построенных на базе МП *i*386, *i*486 не будут эффективными для Пентиума, поскольку они построены в расчете на экономию памяти и РОН, что приводит к появлению искусственных зависимостей по данным между смежными командами и уничтожает параллелизм команд.

Для суперскалярных процессоров необходимы новые компиляторы распараллеливающего типа.

Конвейер МП Пентиум выполняет команды строго в порядке их следования в программе. Такой способ функционирования существенно снижает эффективность конвейера, поскольку конвейер вынужден останавливаться, когда на этапе EX выполняется многотактная команда. Кроме того, для загрузки двух целочисленных конвейеров используется параллелиз только пары смежных команд, а он невелик, поэтому и вероятность совместной работы этих конвейеров невысока.

Повысить эффективность суперскалярной обработки можно, если допустить выполнение команд в порядке готовности их операндов. Такая архитектура использована в МП P6 фирмы Intel, упрощенная структурная схема которого представлена на рис. 4.10. В МП P6 используется та же система команд, что и во всех микропроцессорах i80*x*86.

Рис. 4.10. Структура МП с произвольным порядком выполнения команд

Функции блоков микропроцессора таковы:

1. Блок выборки и декодирования команд ВД обеспечивает чтение команд их КЭШ, их декодирование, преобразование и запись в буфер команд БК. Команды обрабатываются блоком ВД в порядке их следования в программе.

2. Буфер команд БК представляет собой ассоциативную память, в которой команды представлены в трехадресном формате.

Поскольку в системе команд i80x86 мало РОН (всего 8), то из-за этого между командами возникают ложные зависимости по данным. Для устранения этих зависимостей в МП P6 введено 40

дополнительных регистров, которые недоступны программисту. Они используются аппаратурой для временного хранения результатов. Обозначим эти регистры временного хранения через V. Тогда на рис. 4.10 V1, V2 иVP обозначают соответственно номера регистров для хранения первого, второго операндов и результата. Преобразование команд системы i80x86 в трехадресный формат и переименование регистров производится блоком ВД.

Каждая команда в БК сопровождается блоком событий БС, в котором отмечаются следующие состояния команды: готовность каждого операнда, готовность команды к исполнению, готовность результата и др.

3. Центральным блоком МП Р6 является блок планирования и выполнения команд ПВ. Именно он выполняет команды в порядке их готовности. ПВ содержит несколько АЛУ и устройств обращения к памяти. За один такт ПВ способен одновременно запустить на исполнение до пяти команд и передать в БК до пяти результатов.

Выборка готовых команд из БК производится путем ассоциативного опроса блока БС всех команд, а размещение нескольких команд по устройствам ПВ осуществляется в соответствии с определенным алгоритмом планирования. Одним из наиболее простых для аппаратной реализации является алгоритм FIFO (первым пришел — первым ушел).

При получении в ПВ каждого нового результата адрес его размещения используется как ассоциативный признак для полей V1, V2 буфера команд. Если одна или несколько команд откликнулись на этот признак, значит, эти команды завися по данным от выполненной команды. В блоках БС откликнувшихся команд устанавливаются биты и готовности операндов, а, если готовыми оказались оба операнда, то устанавливается и бит готовности для исполнения всей команды. Это позволяет сформировать очередной набор “готовых” команд для следующего такта работы ПВ.

Чтобы блок ПВ мог выполнять за один такт до 3...5 команд необходимо, чтобы в БК находилось до 20...30 команд. По статистике среди такого объема команд в среднем имеется 4...5 команд условных переходов. Следовательно в БК находится некоторая трасса выполнения команд. Выбор таких наиболее вероятных трасс является новой функцией МП с непоследовательным выпол-

нением команд. Эта функция выполняется в блоке ВД на основе расширенного до 512 входов буфера истории переходов.

Поскольку реально вычисленный в ПВ адрес перехода не всегда совпадает с предсказанным в блоке ВД, то вычисление в ПВ выполняется условно, т. е. результат записывается в регистр временного хранения. Только после того, как установлено, что переход выполнен правильно, блок удаления команд УК выводит из БК все выполненные команды, расположенные за командой условного перехода, преобразует их в формат системы i80x86 и производит запись результатов по адресам, указанным в исходной программе.

Блоки ВД, ПВ и УК совместно составляют конвейер из 12 этапов, однако все три части этого конвейера работают практически независимо, взаимодействуя только через БК. Следовательно, такой конвейер можно считать неблокируемым, так как остановка любой его части не прекращает работу других частей.

§ 4.4. Другие типы параллельных процессоров и ЭВМ

Ранее были рассмотрены четыре основных типа параллельных структур: конвейерные процессоры, процессорные матрицы, многопроцессорные ЭВМ с управлением от потока команд и многопроцессорные ЭВМ с управлением от потока данных. Однако число типов реально производимых и разрабатываемых параллельных ЭВМ значительно больше и не укладывается в жесткие рамки приведенной классификации по двум причинам: появляются новые, ранее неизвестные типы ЭВМ; некоторые разновидности машин внутри определенного типа приобретают собственное наименование.

В настоящем параграфе будут кратко рассмотрены следующие важные для теории и практики типы параллельных машин: матричные процессоры, ассоциативные ЭВМ, систолические массивы, многопроцессорные системы с программируемой архитектурой [5].

Матричные процессоры. С понятием *матричный процессор* связывают три различных характеристики: способ организации данных, способ организации устройств и класс решаемых задач.
В практике проектирования ЕС ЭВМ под матричным процессором понимают устройство, предназначенное в основном для выполне-

ния операции $S = \sum_{l=1}^{i} X_i Y_i$. Такой матричный процессор функционирует на основе арифметического конвейера и отличается от структур, описанных в § 2.2, тем, что для уменьшения стоимости и упрощения программирования арифметический конвейер значительно укорочен и конструктивно выполнен в виде автономного блока. Последний подключается через стандартные каналы ввода-вывода к базовой ЭВМ, в качестве которой может использоваться любая серийная ЭВМ. Собственной ОП матричный процессор не имеет. Его программа и данные располагаются в ОП БМ.

Способ подключения матричного процессора к БМ и его структура изображены на рис. 4.11. Матричный процессор вместо одного из каналов подключается на внешнюю шину и содержит регистры для управления адресом оперативной памяти, буферные регистры для промежуточного хранения данных *БХ*, *БU, БУ* и двухступенчатое АЛУ.

Команда матричного процессора (рис. 4.12) состоит из четырех слов по 8 байт каждое. Описание массивов операндов *X*, *U*, *Y* определяет начальный адрес каждого массива, шаг по массиву (поле “Индекс”) и длину массива. Поле “Формат” указывает тип операндов (фиксированная, плавающая запятая и т. д.). Первое слово команды матричного процессора имеет структуру стандартной команды канала ЕС ЭВМ, однако назначение полей несколько иное. Так, поле “КОП” задает код арифметико-логической операции над массивами *X*, *U*; поле “Адрес описания массивов” задает начальный адрес описания массивов в ОП, а поле “Длина описания” — его длину.

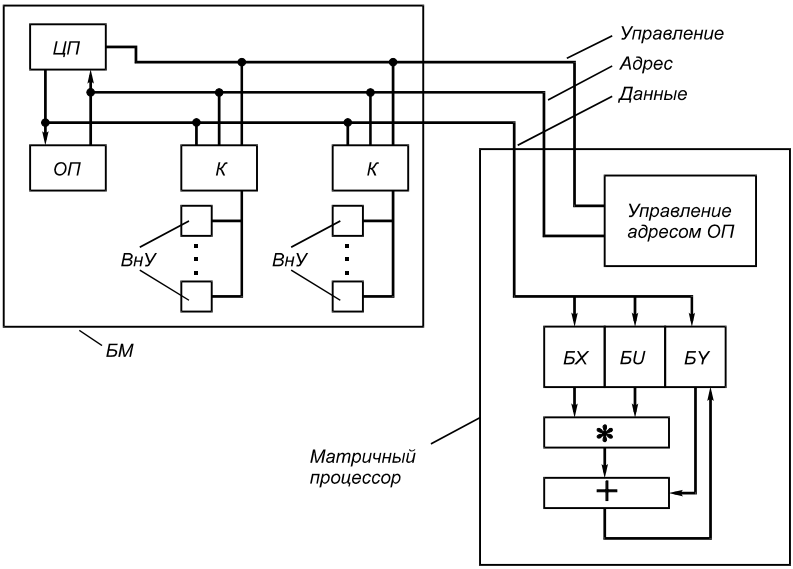


Рис. 4.11. Способ подключения и структура матричного процессора:
К — каналы ввода-вывода; * — умножитель; + — сумматор

Рис. 4.12. Структура команды матричного процессора

Программирование для матричного процессора состоит в том, что в программу для БМ вставляются в нужных местах команды матричного процессора, которые имеют вид

CALL APAM < код операции, описание массивов >

Это означает, что любая команда матричного процессора эквивалентна некоторой подпрограмме. Если в процессе выполнения

программы в БМ встречена команда *CALL APAM,* в матричный процессор передается информация об адресах массивов *X*, *U*, *Y*, после чего матричный процессор самостоятельно вырабатывает адреса ОП, выбирает из ОП в *БХ*, *БУ* операнды и размещает в ОП из *БУ* результаты. После выполнения команды матричного процессора центральный процессор продолжает программу базовой ЭВМ.

Система команд матричного процессора, которая позволяет решать сложные задачи в области геофизики, метеорологии, радиолокации, медицины и т. д., приведена в табл. 4.3. Знак [ ] означает необязательную часть операции, которая может быть опущена при использовании команд матричного процессора

Ассоциативные ЭВМ. Все ПМ потенциально обладают возможностью *ассоциативной обработки*, но в машинах широкого применения эти свойства не всегда явно выражены. Тогда для ускорения специальных видов ассоциативной обработки в ПМ вводят дополнительные средства и такие матрицы называются ассоциативными ЭВМ. Примером подобной ЭВМ является машина STARAN.

В машине STARAN, используется БМ, имеется ЦУУ, однако само процессорное поле устроено иначе (рис. 4.13).

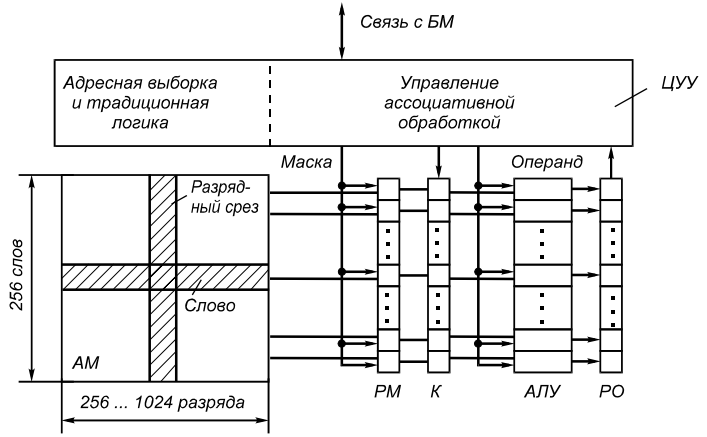


Рис. 4.13. Структура АМ

Таблица 4.3.

Система команд матричного процессора

Тип команды	Код	Команда	Выполняемая

			операция
Арифметические операции над векторами и матрицами	VEM	Поэлементное умножение векторов	$y(i) =[y(i)]+u(i)x(i)$
	VES	Поэлементное сложение векторов	$y(i)= u(i) + x(i)$
	SMY	Умножение вектора на скаляр	$y(i) = [y(i)] + x(i)u$
	SSA	Возведение элементов вектора в квадрат	$y(i) = [y(i)] + x(i)[x(i)]$
	SSQ	Суммирование квадратов элементов вектора	$y=[y] + \sum^{n} x(i)[x(i)]$
	SVE	Суммирование элементов вектора	$y = [y] + \sum^{n} x(i)$
	VIP	Скалярное произведение векторов	$y = [y] + \sum^{n} x(i)u(i)$
	PMM	Частичное умножение матриц	Сложные операции
	CEM	Умножение комплексных векторов	
Специальные операции	CVM	Свернутое умножение	
	DEQ	Разностное уравнение	
	MAX	Сканирование вектора	
	FF2	БПФ по основанию 2	
	FF4	БПФ по основанию 4	
	FF4	Обратное БПФ по основанию 4	

Ассоциативная матрица обычно содержит до 256 слов достаточно большой разрядности, к которым можно обращаться традиционным адресным способом. Однако в дополнение к этому в АМ все слова могут одновременно поразрядно сдвигаться вправо и влево. Именно это свойство и усиливает возможности ассоциативной обработки. В ассоциативном процессоре имеется ряд дополнительных регистров и устройств, обеспечивающих выполнение не-

обходимых арифметико-логических операций сразу над одноименными разрядами всех слов.

Рассмотрим, как в ассоциативном процессоре выполняется операция поиска по образцу. Каждый разряд операнда-образца из ЦУУ одновременно передается на все АЛУ, куда синхронно с ним поступает разрядный срез из АМ. Каждое АЛУ производит сравнение своей пары разрядов и в случае их несовпадения устанавливает в единицу свой разряд регистра отклика (РО). После выполнения нужного числа сдвигов АМ в РО нулевыми будут обозначены те слова, содержимое которых совпадает с образцом.

Анализируя содержимое РО, можно установить номер первого совпавшего слова, номер всех совпавших слов, качество совпавших слов и т. д. Сравнение может производиться не по всей длине слова, а по части. С помощью регистра маски (РМ) отдельные слова могут маскироваться, а коммутатор К позволяет осуществлять обмен между АМ и массивом АЛУ с заданным сдвигом.

Таким образом, ассоциативную память можно определить как память, из которой находящиеся в ней данные могут быть выбраны на основании их содержания или части их содержания, а не по адресам данных.

Типичными операциями поиска и сравнения, выполняемыми ассоциативным процессором, являются следующие: “равно — не равно”, “меньше, чем — больше, чем”, “не больше, чем — не меньше, чем”, “максимальная величина — минимальная величина”, “между границами — вне границ”, “следующая величина больше — следующая величина меньше” и др. Перечисленные операции позволяют осуществлять с помощью ассоциативного процессора сложные виды числовой и нечисловой обработки.

Систолические массивы. Достижения микроэлектроники позволяют размещать на одном кристалле большое количество простых ПЭ. Скорости срабатывания ПЭ высоки, поэтому возрастает сложность создания коммутационных сетей, связывающих ПЭ. В некоторых специализированных ЭВМ можно отказаться от коммутаторов. Массивы ПЭ с непосредственными соединениями между близлежащими ПЭ называются *систолическими*. Такие массивы исключительно эффективны, но каждый из них ориентирован на решение весьма узкого класса задач.

Рассмотрим, как можно построить систолический массив для решения некоторой задачи. Пусть, например, требуется создать устройство для вычисления матрицы $D=C+AB$, где

$$A = \begin{Vmatrix} a_{11} & a_{12} & & 0 \\ a_{21} & a_{22} & a_{23} & \\ a_{31} & a_{32} & \dots & \dots \\ & a_{42} & \dots & \dots \\ 0 & & \dots & \end{Vmatrix}, \quad B = \begin{Vmatrix} b_{11} & b_{12} & b_{13} & 0 \\ b_{21} & b_{22} & \dots & \dots \\ & b_{32} & \dots & \dots \\ 0 & & \dots & \end{Vmatrix},$$

$$C = \begin{Vmatrix} c_{12} & c_{12} & c_{13} & c_{14} & 0 \\ c_{21} & c_{22} & c_{23} & \dots & \dots \\ c_{31} & c_{32} & & \dots & \dots \\ c_{41} & \dots & & & \\ 0 & \dots & & & \end{Vmatrix}.$$

Здесь все матрицы — ленточные, порядка n. Матрица A имеет одну диагональ выше и две диагонали ниже главной; матрица B — одну диагональ ниже и две диагонали выше главной; матрица C по три диагонали выше и ниже главной.

Пусть каждый ПЭ может выполнять скалярную операцию $c+ab$ и одновременно осуществлять передачу данных. Каждый ПЭ, следовательно, должен иметь три входа: a, b, c и три выхода: a, b, c. Входные (in) и выходные (out) данные связаны соотношениями

$$a_{out} = a_{in}, \quad b_{out} = b_{in}, \quad c_{out} = c_{in} + a_{in}b_{in}.$$

Если в момент выполнения операции какие-то данные не поступили, то будем считать, что они доопределяются нулями. Предположим далее, что все ПЭ расположены на плоскости и каждый из них соединен с шестью соседними (рис. 4.14). Если расположить данные, как показано на рисунке, то схема будет вычислять матрицу D.

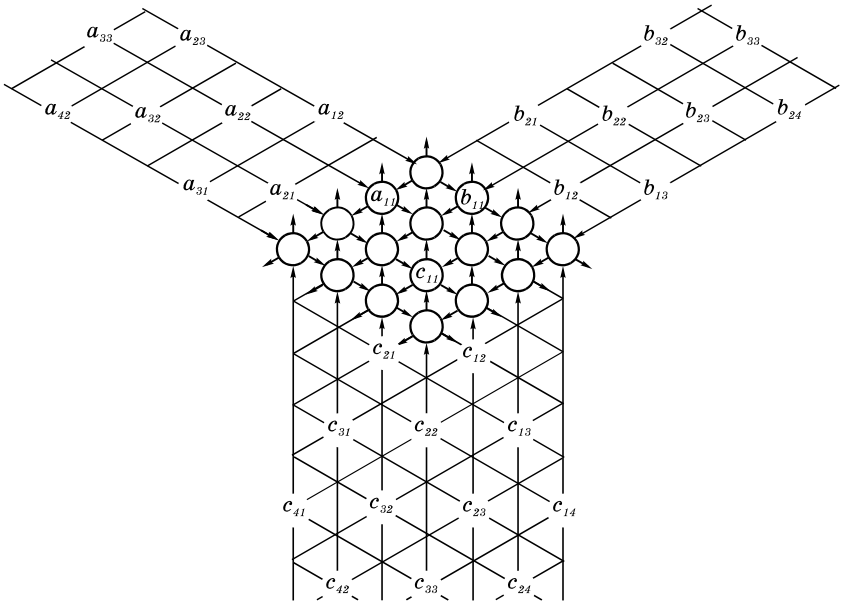


Рис.4.14. Систолический массив, реализующий операцию $D=C+AB$

Поясним работу массива. Массив работает по тактам. За каждый такт все данные перемещаются в соседние узлы по направлениям, указанным стрелками.

На рисунке показано состояние систолического массива в некоторый момент времени. В следующий такт все данные переместятся на один узел и элементы a_{11}, b_{11}, c_{11} окажутся в одном ПЭ, находящемся на пересечении штриховых линий. Следовательно, будет вычислено выражение $c_{11} + a_{11}b_{11}$. В этот же такт данные a_{12} и b_{21} вплотную приблизятся в ПЭ, находящемся в вершине систолического массива. В следующий такт все данные снова переместятся на один узел в направлении стрелок и в верхнем ПЭ окажутся a_{12} и b_{21} и результат предыдущего срабатывания ПЭ, находящегося снизу, т.е. $c_{11} + a_{11}b_{11}$. Следовательно, будет вычислено выражение $c_{11} + a_{11}b_{11} + a_{12}b_{21}$. Это есть элемент d_{11} матрицы D.

Продолжая потактное рассмотрение процесса, можно убедиться, что на выходах ПЭ, соответствующих верхней границе

систолического массива, периодически через три такта выдаются элементы матрицы D, при этом на каждом выходе появляются элементы одной и той же диагонали. Примерно через $3n$ тактов будет закончено вычисление всей матрицы D. При этом загруженность каждой систолической ячейки асимптотически равна 1/3.

Систолический массив имеет черты как процессорных матриц (совокупность связанных ПЭ, выполняющих единую команду), так и явные признаки конвейерного вычислителя (потактное получение результата).

Многопроцессорные системы с программируемой архитектурой. В этих случаях в отличие от систолических матриц можно программно устанавливать связи между процессорными элементами и функции, выполняемые данными ПЭ. Это значительно расширяет возможности массива процессорных элементов.

Важнейшей составной частью многопроцессорной системы с *программируемой архитектурой* является универсальная коммутационная среда (УКС) [8], которая состоит из однотипных, соединенных друг с другом регулярным образом автоматических коммутационных ячеек, характеризующихся коллективным поведением.

Структура многопроцессорной системы, использующей универсальную коммутацию, приведена на рис. 4.15. Настройка системы на выполнение заданной задачи производится в два этапа.

Первый этап заключается в распределении крупных операций между ПЭ и в настройке данных ПЭ на эти операции. Примерами крупных операций являются: интегрирование, дифференцирование, матричные операции, быстрое преобразование Фурье. Использование таких макроопераций значительно упрощает программирование.

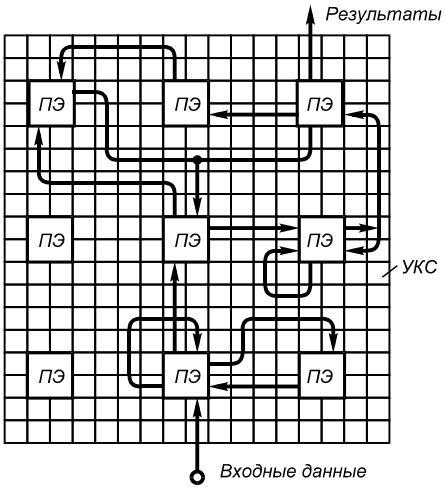


Рис. 4.15. Структура многопроцессорной системы с программируемой архитектурой

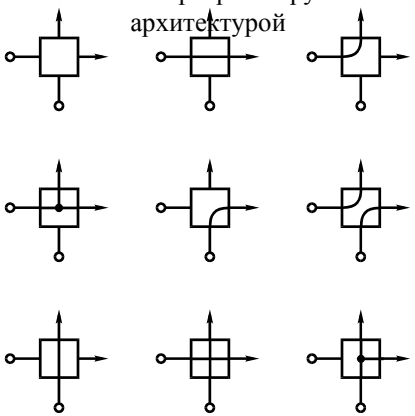


Рис.4.16. Варианты внутренней коммутации простейшего КЭ

Второй этап заключается в настройке необходимых каналов связи между ПЭ в УКС. После этого осуществляется автоматический режим работы многопроцессорной системы, в течение которого на вход подается входная информация, а с выхода снимаются результаты.

Как указывалось ранее, УКС принципиально является наиболее важной частью систем с программируемой структурой. Ознакомимся с некоторыми способами построения и настройки такой среды. Для простоты будем рассматривать плоскую УКС, хотя могут быть использованы и многомерные УКС.

Простейший КЭ, необходимый для построения плоской УКС, содержит два входа и два выхода и связан с четырьмя соседними КЭ. В таком элементе возможны девять вариантов внутренней коммутации (рис. 4.16), что и обеспечивает возможность установления. Произвольных связей между ПЭ, в том числе и таких, которые изображены на рис. 4.15.

Рассмотрим настройку и функционирование плоской УКС.

Для образования каналов передачи информации намечается непрерывная цепочка свободных КЭ, соединяющих входы и выходы нужных ПЭ. При этом цепочка должна обходить как неисправные КЭ, так и элементы, уже вошедшие в другие каналы связи. После выбора канала связи во все выбранные КЭ вводится код настройки, затем канал связи становится подобным сдвиговому регистру, в котором информация потактно передается от одного КЭ к другому.

Настройка КЭ может производиться как вручную, так и автоматически. Для внешней настройки необходимо специальное внутреннее устройство управления, которое будет громоздким из-за сравнительной сложности алгоритма настройки. Поэтому более предпочтительной является самонастройка УКС.

Для автоматического образования одного канала связи между двумя ПЭ можно воспользоваться волновым принципом. Процедура распространения волны заключается в следующем. Вначале в УКС указываются входной и выходной КЭ. Затем входной элемент соединяется со всеми исправными и незанятыми соседними элементами, образуя волну подсоединенных КЭ. Если в ней не оказалось нужного выходного элемента, то каждый элемент волны соединяется с соседними, образуя новый фронт. Продвижение волны продолжается до тех пор, пока ее фронт не достигнет нужного выходного КЭ, после чего в элементы, составляющие кратчайший путь между двумя ПЭ, заносятся коды настройки, а остальные установленные связи разрываются.

Таким образом, последовательно устанавливаются все связи, необходимые для решения некоторой задачи.

Эффективность систем с программируемой архитектурой зависит от величины коэффициента $K = t_з / t_н$, где $t_з$ — время решения задачи; $t_н$ — время настройки системы. Чем больше K, тем эффективнее система. Время $t_н$ для систем с программируемой архитектурой велико, поэтому повысить K можно только увеличением $t_з$.

Один из путей увеличения $t_з$ состоит в том, что после настройки система длительное время используется для решения одной и той же задачи, при этом в каждый такт на вход системы поступает новый набор исходных данных, а с выхода системы в каждый такт снимаются новые результаты. Этим и объясняется очень высокое быстродействие при решении крупных задач.

Следовательно, наиболее выгодным для систем с программируемой архитектурой является режим конвейерной обработки. В то же время они принадлежат к многопроцессорным системам типа МКМД.

Контрольные вопросы

1. Объясните сущность скалярного параллелизма. Что такое длинная и сверхдлинная команды?
2. Сопоставьте классы команд CISC, RISC и MISC.
3. Назовите состав и определите функции ступеней скалярного конвейера.
4. Какое влияние на быстродействие конвейера оказывает зависимость по данным между командами?
5. Опишите влияние условных переходов на эффективность конвейера. Как ослабить это влияние?
6. Какова эффективность конвейера при выполнении многотактных команд?
7. Определите принципы построения универсальной КЭШ-памяти.
8. Каковы особенности конвейера на базе системы команд типа RISC?
9. Охарактеризуйте структуру суперскалярного микропроцессора с двумя конвейерами.
10. В чем заключается статический способ генерации программ для суперскалярных процессоров?
11. Опишите динамические способы генерации программ для суперскалярных процессоров.
12. Что такое матричные процессоры?
13. В чем сущность систолических структур?
14. Каковы отличительные признаки многопроцессорных систем с программируемой архитектурой?

Глава 5

ПРОГРАММНОЕ ОБЕСПЕЧЕНИЕ ПАРАЛЛЕЛЬНЫХ ЭВМ

§ 5.1. Особенности программного обеспечения параллельных ЭВМ

Параллелизм оказывает существенное влияние на архитектуру вычислительных средств.

Под вычислительными средствами будем понимать аппаратуру ЭВМ и ее программное обеспечение. Влияние параллелизма и прежде всего системы команд на аппаратуру было описано в предыдущих главах, в этой главе будет рассмотрено влияние параллелизма на программное обеспечение.

Программное обеспечение традиционной универсальной ЭВМ можно разделить на три части, часто называемые системным программным обеспечением: операционные системы, системы программирования и прикладные пакеты.

Ядро операционной системы (ОС) мультипрограммной ЭВМ содержит следующие компоненты:

1. Средства управления ресурсами системы (драйверы ввода-вывода, динамическое управление памятью, система прерывания процессора и др.).

2. Средства управления процессами (создание, уничтожение, синхронизация процессов).

3. Средства планирования и распределения ресурсов между задачами, заданиями, процессами.

4. Системы управления файлами.

5. Интерфейс пользователя, оператора (управляющие средства, командные языки).

6. Средства восстановления и реконфигурации ЭВМ и ОС после появления сбоев и неисправностей.

В зависимости от режима работы мультипрограммная ОС приобретает специализацию. В частности, для пакетного режима усиливаются функции планирования, оптимизируется распределение времени между пользователями в системах разделения времени, совершенствуются средства прерывания в системах реального времени.

Системы программирования включают языки программирования, компиляторы, отладчики, средства объединения и загрузки программ, редакторы, библиотеки подпрограмм и функций разных этапов выполнения программы. Интегрированные системы программирования обычно включают также собственную систему управления файлами, интерфейс программиста и некоторые другие функции, характерные для ОС.

Прикладные пакеты разрабатываются и поставляются как фирмами, которые разрабатывают оборудование, так и фирмами-

разработчиками программного обеспечения. Существует большое количество пакетов различного назначения, например, пакеты для автоматизации проектирования, текстовые редакторы, графические пакеты, пакеты научной математики, базы данных, пакеты для цифровой обработки сигналов, для обработки экономической информации и др. Эти пакеты, как правило, имеют хорошую научную основу, имеют большой объем программ и сложную организацию. В них используются специальные языки описания предметной области, интерфейс пользователя и некоторые элементы управления, свойственные операционным системам.

Параллелизм в основном затрагивает систему программирования, причем, в наибольшей степени это проявляется при программировании одной задачи, поскольку основное назначение параллельных систем — уменьшение времени выполнения отдельных трудоемких задач. Поэтому, если в последовательной ЭВМ процесс программирования можно, образно говоря, считать одномерным, так как в этом случае достаточно только обрабатывать последовательность арифметико-логических операций, то в параллельной ЭВМ необходимо еще отслеживать размещение процессов в пространстве ресурсов и их синхронизацию во времени, то есть здесь процесс программирования является, по крайней мере, трехмерным.

На систему программирования ложатся функции создания языков программирования и трансляторов, которые во многих случаях должны обеспечивать функции автоматического распараллеливания последовательных программ (программ, а не алгоритмов, так как распараллеливание алгоритмов относится к области параллельной вычислительной математики и рассматривается в главе 6).

При появлении параллельных ЭВМ приходится перерабатывать ранее написанные прикладные пакеты, однако вопросы их мобильности в настоящем пособии не рассматриваются.

Таким образом глава 5 в основном посвящена системам программирования.

§ 5. 2. Векторные языки

Наиболее важным и трудоемким для векторных ЭВМ является создание языков и трансляторов.

На рис. 5.1 показана зависимость быстродействия конвейерной ЭВМ от размера L матрицы при выполнении одного и того же алгоритма (отыскание LU-разложения), но записанного на ассемблере (кривая 1), машинно-ориентированном параллельном ЯВУ (кривая 2), проблемно-ориентированном параллельном ЯВУ (кривая 3), последовательном Фортране с подпрограммами на параллельном ассемблере (кривая 4), последовательном Фортране с векторизацией (кривая 5), на Фортране без векторизации (кривая 6). Несмотря на очень большой разрыв в производительности, обеспечиваемой различными языками, на практике по тем или иным причинам используются все названные языки.

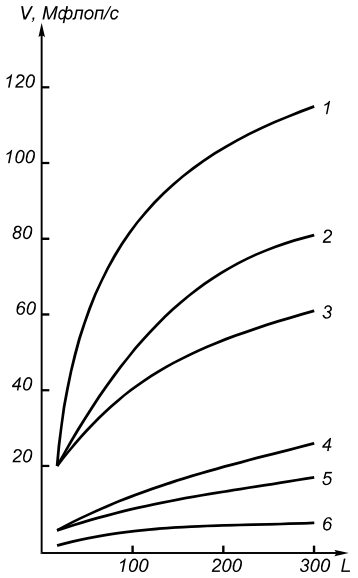


Рис.5.1. Производительность конвейерной ЭВМ при выполнении матричной операции по программам, написанным на разных языках

Языки типа ассемблер в символической форме отражают машинную систему команд параллельной ЭВМ, т. е. все особенности структуры ЭВМ: число процессоров, состав и распределение регистров и ОП, систему коммутации, особенности структуры ПЭ, средства специализированного размещения данных и т. д. Языки такого типа используются на всех параллельных ЭВМ и позволяют детально приспособить параллельный алгоритм к особенностям структуры конкретной параллельной ЭВМ. Программы, написанные на ассемблере, обеспечивают наивысшую производительность ЭВМ.

Ассемблер, несмотря на высокую эффективность программ, написанных на этом языке, не может использоваться для массового прикладного программирования по следующим причинам:

1) программирование на ассемблере является очень трудоемким даже для последовательных ЭВМ, не говоря уже о параллельных, где необходимо дополнительно программировать размещение данных и синхронизацию процессов;

2) в вычислительной технике происходит регулярная смена элементной базы, что сказывается на системе команд ЭВМ, в особенности параллельных (это означает переработку всего запаса программ, написанных на ассемблере);

3) программы, написанные на ассемблере, не мобильны, т. е. их нельзя выполнять на параллельных ЭВМ других классов, а часто и внутри одного класса ЭВМ, но для разных размеров ПП.

Поэтому, как правило, программы на ассемблере используются:

1) при создании системного математического обеспечения;

2) для написания программных библиотек стандартных функций и численных методов;

3) для оформления гнезд циклов с наибольшим временем счета в программах, написанных на ЯВУ;

4) для написания небольшого числа программ постоянного пользования в специализированных ЭВМ.

Параллельные ЯВУ в большинстве случаев строятся как расширение стандартных вариантов Фортрана, хотя в некоторых случаях используются расширения на базе языков Паскаль, Ада, Си. Иногда параллельный язык строится как новый, не имея в качестве базы стандартного последовательного языка.

Проблемно-ориентированные языки не имеют структур данных и операций, отражающих свойства конкретной параллельной ЭВМ. Если ЯВУ является векторным, то написанные на нем программы должны с равной эффективностью выполняться как на конвейерных ЭВМ, так и на ПМ. В проблемно-ориентированных языках все конструкции данных и операции над ними соответствуют математическому понятию этих данных и операций. Размерности данных, как правило, не ограничены, в таких языках часто отсутствуют средства для размещения пользователем данных в ПП. Проблемно-ориентированные языки достаточно просты для программирования.

Машинно-ориентированные языки в сравнении с проблемно-ориентированными языками позволяют создавать значительно

более эффективные программы, предоставляя пользователю средства, отражающие структуру параллельной ЭВМ.

В машинно-ориентированных языках часто вводятся:

1) переменные типа “суперслово”. Число элементарных слов в суперслове строго соответствует размеру ПП (64 для ЭВМ ILLIAC-IV). Пользователь имеет возможность представить данные в виде суперслова, обеспечивая, таким образом, лучшее использование ОП и увеличенное быстродействие;

2) операторы для размещения в ПЭ данных по “срезам” различных индексов массивов;

3) средства типа “масок” для включения-выключения отдельных ПЭ.

Написание программ на машинно-ориентированных ЯВУ является более трудоемким процессом, чем написание программ на проблемно-ориентированных ЯВУ.

Для специализированных применений ЭВМ часто используются системы программирования, содержащие библиотеку программ ограниченного размера, написанную на ассемблере. Подпрограммы библиотеки вызываются из основной программы, оформленной на последовательном Фортране или другом ЯВУ и предназначенной для выполнения в базовой ЭВМ. Этот метод использования библиотечных программ обладает средними характеристиками по трудоемкости, эффективности и универсальности.

При разработке параллельного ЯВУ необходимо:

- выбрать новые структуры данных и определить методы доступа к структурам и элементам этих структур данных;
- определить допустимые арифметико-логические и управляющие операции над введенными структурами данных;
- определить базовые методы размещения и коммутации данных в ПП;
- предложить изобразительные средства для оформления структур данных и операций над ними.

Рассмотрим наиболее общие методы решения этих вопросов в параллельных ЯВУ [2].

Основными объектами параллельной обработки в параллельных ЭВМ являются массивы различной размерности, а также подструктуры этих массивов: плоскости строки, столбцы диагонали и

отдельные элементы. Следовательно, для выборки данных объектов в параллельном ЯВУ должны существовать специальные механизмы.

Выборка объектов пониженного ранга. Элементы массива разного ранга выбираются посредством индексации одной или нескольких размерностей массива. В частности, если задана матрица $A(N, N)$, то:

- обозначение A относится ко всей матрице;
- $A(I)$ относится ко всей I-й строке матрицы A (вектор);
- $A(J)$ относится к J-му столбцу матрицы A (вектор);
- $A(I, J)$ относится к одному элементу матрицы A (скаляр).

Наибольший интерес в языках представляет процесс получения векторов (одномерных массивов), так как именно они являются основным объектом обработки в векторных ЭВМ.

В некоторых языках синтаксис механизма задания векторов определяется специальным символом ∗, который устанавливается в позиции опущенного индекса. Так, для приведенного выше примера получаем: $A(*, *)$, $A(I, *)$, $A(*, J)$, $A(I, J)$. Однако часто соответствующий индекс просто опускается.

Примеры подобных выборок даны на рис. 5.2.

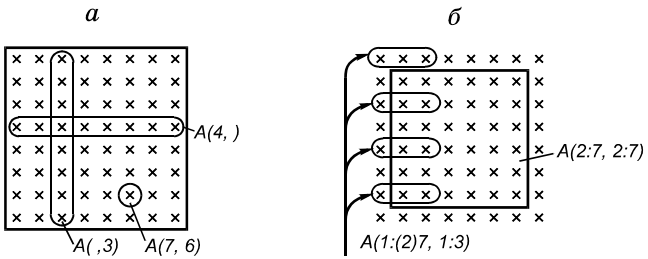


Рис. 5.2. Пример выборки, понижающей ранг (а) и сужающей диапазон (б), из массива 8×8

Выборка диапазона значений. Выборка диапазона значений производится для матрицы $A(N{\times}N)$ с помощью конструкций вида $A(2 : N\text{-}1, 2 : N\text{-}1)$ или $A(1 : (2)N, 1 : 3)$. В обоих примерах исполь-

зуемый синтаксис указывает начало, шаг (:), если он есть, и конечную границу диапазона изменения данного индекса.

Поэтому первое выражение позволяет осуществить выборку внутренних точек матрицы, а второе — выбрать первые три элемента из нечетных строк (см. рис. 5.2).

Выборка с помощью целочисленных массивов. Применение этого метода предоставляет большие возможности для выделения подструктур. Пусть даны масссивы целых чисел *IV*, *JV*. Размерность этих массивов может быть произвольной, но числа в них должны быть меньше или равны *N*, так как они являются индексами элементов массива *A*.

Возьмем для примера следующие исходные массивы и векторы:

$A \rightarrow$	*a*	*b*	*c*	*d*		*IV*	1	1	1	3
	e	*f*	*g*	*h*						
	i	*j*	*k*	*l*						
	m	*n*	*o*	*p*		*JV*	3	2	1	

Тогда возможны следующие выражения:

а) $A(I, JV(I)) = A(3, JV(3)) \Rightarrow i$
б) $A(*, IV(*)) \Rightarrow a\ e\ i\ o$
в) $A(IV(*), *) \Rightarrow a\ b\ c\ l$

Выражение а) позволяет выбрать единичный элемент матрицы *A*, если задан номер строки *I*. Этот элемент имеет индексы *A*(*I*, *K*), где *K* — число, записанное на *I*-м месте массива *JV*. Выражения б) и в) позволяют из матрицы *A* извлечь проекцию в виде вектора. Каждый элемент вектора получается аналогично случаю а), но для всех индексов данной размерности.

Выборка с помощью элементов булевых массивов. Выборка осуществляется в зависимости от состояния логического селектора, который может являться элементом логического массива или результатом вычисления логического выражения.

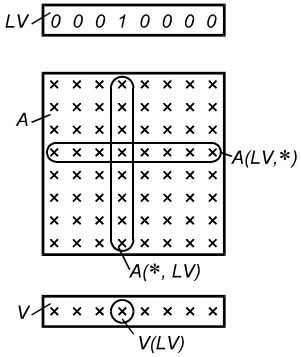


Рис. 5.3. Пример выборки булева вектора *LV*

Несколько примеров такой выборки дано на рис. 5.3.

Индексация сдвига. Эта процедура выравнивает положение массивов относительно друг друга с помощью индексных операций и часто используется при решении дифференциальных уравнений в частных производных сеточным методом. Здесь каждый узел сетки корректируется по среднему значению от четырех соседних узлов, что можно выразить следующим образом:

$$A=(A(*+1,\ *) + A(*\ -\ 1,\ *) + A(*,\ *\ -\ 1) + A(*,\ *\ +\ 1))/4.$$

Все элементы *A* обновляются одновременно в соответствии со значением четырех соседних элементов, выбираемых путем сдвига массива *A* влево (+) и вправо (–) в каждой размерности. Для внутренних значений *A* это можно выразить последовательной программой:

```
      DO 1 I = 2, N - 1
      DO 1 J = 2, N - 1
1     ANEW(I, J) = (A(I+1, J)+A(I-1, J)+A(I, J+1)+A(I, J-1))/4.
```

Краевые эффекты, как правило, рассматриваются отдельно в последовательной программе, но это можно сделать путем сдвигов и в параллельной программе, если принять некоторую геометрию соединений краев ПМ. (Примеры таких соединений приведены в § 2.3). Конечно, сдвиг может осуществляться в случае необходимости на любую величину путем записи выражения $*+K$, где K — величина сдвига.

Повышение ранга структуры данных. Часто для упрощения вычислений необходимо многократно повторить, размножить экземпляр скаляра или массива. Такая функция называется *XPND* (“расширитель”). Выражение *XPND* (*A*, *K*, *N*) обозначает массив, сформированный путем повторения *N* раз вектора *A* по своей *K*-й размерности. Например, если *V* — вектор из *N* элементов, то *XPND* (*V*, 1, *N*) — матрица *N*×*N*, сформированная повторением *V* в качестве строк матрицы, а *XPND* (*V*, 2, *N*) сформирована повторением *V* в качестве столбцов матрицы.

Переформирование массивов. Иногда необходимы операции между массивами различной размерности, но содержащими одинаковое общее число элементов. Для переформирования массивов используется оператор *MAP* $A(m_1,\text{K},m_q)$, который преобразует массив $A(n_1,\text{K},n_p)$ в матрицу $A(m_1,\text{K},m_q)$, где q и p — число размерностей нового и старого представления матрицы *A* соответственно, при этом всегда $\sum_{k=1}^{p} n_k = \sum_{k=1}^{q} m_k$.

Ниже дается пример преобразования одномерного массива *A* в двумерный:

```
DIMENSION A(N)
. . .
MAP A (N/2, 2)
. . .
```

Часто возникает и другая задача: большой массив разделяется на части, каждая из которых используется под собственным именем; при этом все данные хранятся в памяти только в области исходного массива, не повторяясь. Операция переименования производится с помощью оператора DEFINE, который записывается следующим образом:

DEFINE $a_1 \cdot e_1 = b_1,\ a_2 \cdot e_2 = b_2,\ \ldots,\ a_k \cdot e_k = b_k$,

где каждое a_i— имя массива, помещаемого на массив b_i; e_i — мерность массива; массив b_i записывается в виде $n_i(p_1,\text{K},p_m)$; n_i — имя массива; p_j — выражение вида *Fl*, где $1 \le l \le m$.

Например, запись

```
DIMENSION B (10, 10, 3)
. . .
DEFINE A (10, 10) = B (F1, F2, 3)
. . .
```

означает, что массив $A(10, 10)$ есть часть массива B, причем $F1$ и $F2$ обозначают соответственно совпадение первой размерности массива A с первой размерностью массива B и второй размерности A — со второй размерностью B. Тогда обращение к элементу $A(4, 5)$ будет в действительности вызывать обращение к элементу $B(4, 5, 3)$.

Размещение данных. От того, каким образом размещены данные в ПП, зависит время выполнения программы. В параллельных ЯВУ размещение данных обычно задается неявно и извлекается транслятором из векторных выражений. Например, следующие выражения обозначают:

а) $A(I, *)$ — матрица A должна быть размещена так, чтобы все элементы любой ее строки находились в разных ПЭ. Это главное условие параллельного доступа к элементам строки, во всяком случае, если длина строки меньше или равна числу процессоров. Если это не так, то элементы строки располагаются слоями. Элементы столбца располагаются в одном ПЭ;

б) $A(*, J)$ — матрица A должна быть размещена параллельно по столбцам;

в) $A(*, *)$ — необходим параллельный доступ как по столбцам, так и по строкам. В этом случае транслятор строит скошенное размещение матрицы A.

Размещение массивов не всегда является очевидным, как в приведенных выше случаях, поэтому в некоторых языках имеются операторы явного задания способов размещения данных. Например, в языке *IVTRAN* (параллельное расширение Фортрана) используются следующие символы: $ — признак основной размерности; # — признак подстрочной размерности. Если параметр основной, то увеличение его на единицу будет увеличивать на единицу номер ПЭ, адрес в модуле памяти увеличиваться не будет.

Если параметр подстрочный, то увеличение его на единицу не меняет номера ПЭ, а увеличивает адрес в модуле памяти на единицу.

С учетом сказанного приведенные ниже выражения обозначают следующее:

1) [$(1), (2)] — размерность на первой позиции будет основной, хотя по обеим координатам имеется параллельный доступ к срезам. Такое размещение целесообразно, когда в программе чаще обращаются к первой и реже ко второй координате;

2) [$(2), (1)] — обращение чаще производится ко второй координате;

3) [(2), # (1)] — по первой координате параллельный доступ не нужен, что и отмечено символом #.

Арифметико-логические операции. Параллельные расширения ЯВУ используют выражения и операторы присваивания, имеющие ту же структуру, что и в последовательных языках, но операндами в них являются векторы. Приведем примеры бинарных векторных операторов:

а) $C1(*) = A1(*) + B1(*)$ — поэлементное сложение двух векторов, т. е. $C1(I) = A1(I) + B1(I)$, $I = 1, 2, ..., N$;

б) $A2(*) = K*A1(*)$ — умножение каждого элемента вектора на константу, т. е. $A2(I) = K*A1(I)$, $I = 1, 2, ..., N$;

в) $C(I, *) = A(I, *) * B(*, J)$ — поэлементное умножение векоров, полученных из матриц A и B, т. е. $C = (I, K) = = A(I, K)*B(K, J)$, где $K = 1, 2, ..., N$.

Бинарные (с двумя операндами) операции в подавляющем большинстве параллельных ЯВУ выполняются только над векторами, как в приведенных примерах. В некоторых языках эти операции могут выполняться и над массивами большей размерности. Однако при любой размерности массивов-операндов над ними выполняются только поэлементные операции. Поэтому оператор

$$C(*, *) = A(*, *) * B(*, *)$$

означает всего лишь попарное умножение элементов двух матриц с одинаковыми индексами, но ни в коем случае не операцию умножения матриц в математическом смысле, которая должна выражаться программой, состоящей из векторных унарных и бинарных операторов.

Унарные (с одним операндом) векторные операции не являются поэлементными и имеют специфический смысл. Примеры некоторых унарных операций приведены в табл. 5.1. Эти операции могут выполняться и над векторами, получаемыми из многомерных массивов.

Таблица 5.1.

Унарные операции в расширениях Фортрана

Функция	Операция	Математический смысл
SUM(A)	+	$A_1.+\ldots+A_k$
PROD(A)	*	$A_1*\ldots*A_k$
AND(B)	$\wedge$	$B_1\wedge\ldots\wedge B_k$
OR(B)	$\vee$	$B_1\vee\ldots\vee B_k$
MAX(A)	$\geq$	MAX($A_1,\ldots,A_k$)
MIN(A)	$\leq$	MIN($A_1,\ldots,A_k$)

Условные переходы в параллельных ЭВМ организуются с помощью оператора IF следующим образом

$$\text{IF } (f\{l_1,\text{K},l_k\})\ m_1,m_2$$

где m_1, m_2 — метки перехода по адресам программы; f — логическая функция k переменных (обычно k равно размерности процессорного поля, например $k = 64$).

Каждый ПЭi имеет бит режима, который устанавливается в 0 или 1 в зависимости от результата выполнения операции именно в этом процессоре, например:

$$l_i = f_1(A(I).LT.0.0) \qquad (5.1)$$

Логическая переменная l_i = 1, если условие в правой части выполняется, и l_i = 0, если условие не выполняется. Таким образом, после выполнения в каждом процессоре операции (5. 1) в ПП формируется логический вектор $\{l_1,\text{K},l_k\}$, который затем считы-

вается в ЦУУ. В качестве функции $f\{l_1,\text{K},l_k\}$ могут применяться различные логические операции, например:

1) ALL $\{l_1,\text{K},l_k\}$ означает, что логическое выражение в операторе является истинным, если все $l_i = 1$;

2) ANY $\{l_1,\text{K},l_k\}$ означает, что функция истинна, если хотя бы один $l_i = 1$.

Конечно, в параллельных расширениях Фортрана должны быть обеспечены возможности вызова параллельных подпрограмм, написанных на различных языках, в том числе и ассемблере. При этом минимальная длина подпрограммы может равняться одному оператору языка ассемблер.

Описанные выше принципы построения параллельных ЯВУ являются основой для построения стандарта параллельного ЯВУ и в различных сочетаниях применяются в языках для существующих параллельных машин CRAY-1, CYBER-205, BSP, DAP, ПС-2000.

Как уже говорилось, мобильность программ для параллельных ЭВМ имеет принципиальное значение. Для обеспечения мобильности применяются два метода: метод стандартного Фортрана и метод трансляции программ с одного диалекта на другой.

Стандартный последовательный Фортран является средой, в которой относительно легко осуществляется перенос программ. Однако использование векторизующего компилятора для последовательного языка все-таки дает программы с низкой производительностью. Кроме того, стандартный Фортран не всегда позволяет эффективно закодировать специфические для данной ЭВМ операции.

Например, цикл

```
      DO 10 I = 1, 64                                    (5.2)
10    IF (A(I).LE.1.OE-6) A(I) = 1.0E-6
```

стандартного Фортрана записывается на Фортране для ЭВМ CRAY-1 с помощью нестандартной функции слияния *CUMPG*, иначе цикл не векторизуется:

```
      DO 10 I = 1, 64
10    A(I) = CUMPG(A(I), 1.0E-6, 1.0E-6 - A(I))
```

Таким образом, даже среда стандартного Фортрана не обеспечивает переноса программ без их изменения.

Метод трансляции программ позволяет применять различные параллельные диалекты ЯВУ и автоматическую трансляцию с одного диалекта на другой. При этом требуется M^2 трансляторов, где M — число диалектов ЯВУ. Если использовать промежуточный язык, то необходимо только около M трансляторов. Главный недостаток метода трансляции заключается в низкой эффективности трансляции (на уровне "среднего" программиста).

Лучшим средством, гарантирующим автоматический перенос программ, является разработка стандартов на параллельные ЯВУ, учитывающих особенности структур различных ЭВМ.

§ 5.3. Векторизующие компиляторы

Один из наиболее распространенных методов программирования для параллельных ЭВМ является метод оформления параллельных алгоритмов на последовательном ЯВУ. Полученные таким образом программы автоматически преобразуются в параллельные программы на машинном языке. Эффективность автоматического преобразования последовательных программ в параллельные относительно невелика, но для многих параллельных ЭВМ названный метод программирования является основным (совместно с программированием на ассемблере) по следующим трем причинам.

1. Параллельные алгоритмы, оформленные в программы на стандартных последовательных языках (чаще всего на Фортране), относительно легко переносимы не только внутри семейства параллельных ЭВМ одного типа, но и между ЭВМ различных типов. В частности, такие программы могут отлаживаться и выполняться на последовательных ЭВМ.

2. Программирование на распространенных последовательных языках хорошо освоено, существует много методической литературы. Отладка программ может производиться на ЭВМ разных типов: ЕС ЭВМ, персональных ЭВМ и др.

3. Автоматические распараллеливатели последовательных программ позволяют транслировать не только новые программы, но и большой запас пакетов прикладных программ, написанных ранее для последовательных машин. Если стоимость ПП невелика по сравне-

нию со стоимостью всей установки, то экономически оправдано распараллеливание и эксплуатация пакетов с низким уровнем машинного параллелизма.

Анализ последовательной программы с целью извлечения параллелизма из программных конструкций называется *векторизацией*, а программные средства, предназначенные для этого, — *векторизующими компиляторами* (ВК).

Векторизующие компиляторы строятся для векторных ЭВМ, т. е. для конвейерных ЭВМ и ПМ.

Наибольшим внутренним параллелизмом, который можно использовать для векторизации программ, являются циклические участки, так как на них приходится основное время вычислений, и в случае распараллеливания вычисления в каждой ветви производится по одному и тому же алгоритму.

В методах распараллеливания циклов используется довольно сложный математический аппарат. Наиболее распространенными методами распараллеливания являются: метод параллелепипедов, метод координат, метод гиперплоскостей. Объектами векторизации в этих методах являются циклы типа DO в Фортране.

При постановке в общем виде задачи распараллеливания циклов вводится понятие "пространство итераций" — n-мерное целочисленное пространство с координатными осями I_1,K,I_n, соответствующими индексным переменным исходного цикла. Каждая итерация (повторение тела цикла при различных значениях индексов) представляет собой точку в этом пространстве и характеризуется значением вектора (i_1,K,i_n), где i_j — номер выполнения итерации на j-м уровне вложенности. Исходный тесногнездовой цикл имеет вид

```
      DO 1 I1 = 1, M1
      . . .
      DO 1 In = 1, Mn                    (5. 3)
      S(i1, ..., in)
1     CONTINUE
```

Шаг изменения цикла предполагается равным единице. Тело цикла $S(i_1,\text{K},i_n)$ состоит из последовательности пронумерованных по порядку операторов. На тело цикла, как правило, наклады-

ваются некоторые ограничения, зависящие от типа ЭВМ и метода распараллеливания: все индексные выражения должны быть линейными функциями от параметров цикла; не допускаются условные и безусловные переходы из тела цикла за его пределы, а внутри цикла передача управления может осуществляться только вперед, т. е. операторы IF и GO TO нельзя применять для организации цикла. Не допускается использование в теле цикла операторов ввода-вывода и обращений к подпрограммам.

Для цикла (5.3) пространство итераций представляет собой набор целочисленных векторов $I = \{i_1, \text{K}, i_n\}$, $1 \le i_j \le M_j$. Распараллеливание цикла (5.3) заключается в разбиении этого пространства на подобласти, внутри которых все итерации могут выполняться одновременно и при этом будет сохраняться порядок информационных связей исходного цикла. Порции независимых итераций можно искать в виде *n*-мерных параллелепипедов, гиперплоскостей и т. д. в соответствии с методом распараллеливания.

Необходимое условие параллельного выполнения *i*-й и *j*-й итераций цикла записывается в виде

$$(\text{OUT(i)} \wedge \text{IN(j)}) \vee (\text{IN(i)} \wedge \text{OUT(j)}) \vee (\text{OUT(i)} \wedge \text{OUT(j)}) = \varnothing \qquad (5.4)$$

Здесь IN(*i*) и OUT(*i*) — множества входных и выходных переменных *i*-й итерации. Два первых дизъюнктивных члена из (5.2) задают учет информационной зависимости между итерациями, т. е. параллельное выполнение может быть невозможным, если одна и та же переменная используется в теле цикла как входная и как выходная. Третий член учитывает конкуренционную зависимость, которая появляется в теле цикла в качестве выходной больше одного раза.

Отношение зависимости между итерациями можно представить в виде графа *D*. Вершины графа соответствуют итерациям цикла. Вершины *i* и *j* соединяет дуга, если они зависимы, и тогда $D(i, j) = 1$. Задачу распараллеливания цикла можно сформулировать как задачу разбиения графа *D* на несвязные подграфы *Di*. Вершины, входящие в один подграф, должны быть независимыми и иметь смежные номера.

Проиллюстрируем применение *метода параллелепипедов* на примере следующего цикла [10]:

```
      DO 1 I = 2, 5
      DO 1 J = 2, 4                                    (5. 5)
1     X (I, J) = X (I - 1, J - 1) ** 2
```

Пространство итераций и информационные связи приведены на рис. 5.4, а. В данном случае вершины, лежащие вдоль диагонали прямоугольника, информационно связаны. Так, в итерации с номером (2. 2) генерируется переменная $X(2, 2)$, которая затем используется в качестве входной переменной в итерации (3. 3). Разбиение итераций на параллелепипеды можно осуществить вдоль осей I и J (см. рис. 5.4, а).

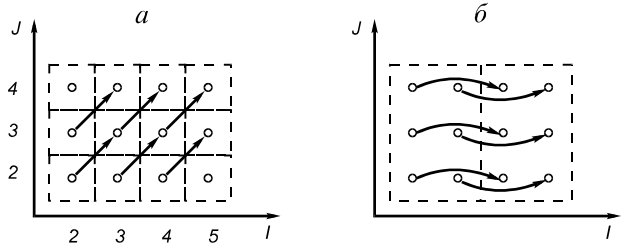


Рис.5.4. Разбиение итераций:
а — для цикла (5.5); *б* — для цикла (5.6)

Проиллюстрируем применение метода параллелепипедов для другого цикла:

```
      DO 1 I = 2, 5
      DO 1 J = 2, 4                                    (5. 6)
1     X (I, J) = X (I - 2, J) ** 2
```

Пространство итераций и информационные связи приводятся на рис. 5.4, б. Разбиение на параллелепипеды можно провести вдоль оси I с шагом, равным двум, а вдоль оси J — с максимально возможным шагом (в данном случае равным трем). Таким образом, внутри каждого параллелепипеда оказалось по шесть независимых итераций, которые можно выполнить параллельно.

Если в цикле есть информационные связи между операторами, запрещающие параллельное выполнение, то можно попытаться устранить эти связи, применив *метод координат*. Данный метод предполагает осуществление в теле цикла некоторых преобразований, а именно: перестановку операторов, введение дополнитель-

ных переменных и массивов. Подобные преобразования производятся таким образом, чтобы изменить направление или устранить запрещающие связи между операторами в различных итерациях. При этом результат вычислений не должен изменяться.

В качестве примера использования метода координат рассмотрим следующий цикл:

```
    DO 4 I = 2, N
1   X (I) = Y (I) + Z (I)
2   Z (I) = Y (I - 1)
3   Y (I) = X (I + 1) ** 2
4   CONTINUE
```

Запрещающими связями в данном примере является связь между использованием переменной Y(*I*-1) во втором и генерацией *Y*(*I*) в третьем операторах, а также связь генерации *X*(*I*) в первом операторе с использованием генерации *X*(*I* + 1) в третьем операторе. Направление первой связи по *Y* можно изменить на противоположное перестановкой второго и третьего операторов. Вторую связь по *X* можно устранить введением дополнительного массива *T*. Таким образом, эквивалентный исходному преобразованный цикл, который можно выполнить одновременно для всех значений индекса *I*, будет иметь такой вид:

```
    DO 4 I = 2, N
    R (I) = X (I + 1)
1   X(I) = Y (I) + Z (I)
3   Y (I) = R (I) ** 2
2   Z (I) = Y (I - 1)
4   CONTINUE
```

Необходимо отметить, что методом параллелепипедов и методом координат можно распараллеливать как одномерные, так и многомерные тесногнездовые циклы.

В рассматриваемых методах распараллеливания индексные выражения элементов массивов должны быть линейными функциями относительно параметров цикла, т. е. иметь вид $A*I \pm B$, где A и B — целые константы; I — параметр цикла *DO*. Необходимо также, чтобы в индексных выражениях использовались одноименные индексные переменные. Не допускается применение нелиней-

ных индексов типа *X*(*I*∗*J*∗*K*) или косвенной индексации, например *X*(*N*(*I*)).

Основным препятствием к распараллеливанию циклов является так называемое *условие Рассела* — использование в теле цикла простой неиндексированной переменной раньше, чем этой переменной присваивается в цикле некоторое значение, например:

```
    DO 1 I = 1, N                    DO 1 I = 1, N
1   S = S + X (I)        или          X (I) = S ∗ Y (I)
                                 1   S = Z (I) ** 2
```

Распараллеливанию препятствуют и обратные информационные и конкуренционные связи между операторами тела цикла. Рассмотрим пример:

```
    DO 1 I = 1, N
1   X (I) = X (I - 1)
```

Итерации этого цикла связаны обратной информационной зависимостью с шагом, равным единице. Вообще говоря, конструкции вида

```
    DO 1 I = 1, N
    . . .
    X (I) = X (I + K)
    . . .
1   CONTINUE
```

где *K* — целочисленная переменная, векторизуются только в том случае, если знак переменной *K* совпадает со знаком инкремента цикла. В большинстве же случаев знак числа *K* не может быть определен на этапе компиляции, поэтому такие конструкции считаются невекторизуемыми. Из соотношения (5.4) следует, что если в теле цикла нет одноименных входных и выходных переменных, то подобный цикл векторизуется. Если в цикле генерация простой переменной встречается раньше, чем ее использование, то этот цикл также векторизуется:

```
    DO 1 I = 1, N
    S = X (I) + Y (I)
1   Z (I) = Z (I) - S
```

Информационная зависимость между итерациями появляется только в том случае, если в теле цикла имеются одноименные входные и выходные переменные с несовпадающими индексными выражениями, например *X*(*I*) и *X*(*I* - 1). Направление связи (прямое или обратное) зависит от номеров операторов, в которых используются эти переменные. Если совпадающие индексные выражения и связи между итерациями отсутствуют, то такие циклы распараллеливаются:

```
      DO 1 I = 1, N
      X (I) = Y (I + 1) + Z (I)
1     Y (I + 1) = X (I) ** 2
```

В этом случае одноименные пары *X*(*I*) и *Y*(*I* + 1) имеют совпадающие индексы и не препятствуют векторизации.

Рассмотрим часто встречающуюся на практике циклическую конструкцию, в которой переменная целого типа используется в качестве неявного параметра цикла:

```
      J = 0
      DO 1 I = 1, N
      J = J + 2
1     X (I) = Y (I)
```

В этом случае в массив *X* заносятся четные элементы из массива *Y*. Роль второго параметра цикла играет переменная *J,* и хотя для *J* формально выполняется условие Рассела, этот цикл можно распараллелить.

Иногда цикл, в котором выполняется условие Рассела, оказывается вложенным в другой цикл, например:

```
      DO 1 I = 1, N
      S = 0
      DO 2 J = 1, N
2     S = S + X (I, J)
      Y (I) = S
1     CONTINUE
```

В вектор *Y* заносятся значения суммы элементов строк матрицы *X*. Внутренний цикл по *J* не распараллеливается (выполняется условие Рассела для переменной *S*), а внешний цикл по *I* можно

распараллелить, так как переменная S в нем получает значение перед входом во внутренний цикл.

Рассмотрим общую структуру ВК (рис. 5.5). Как правило, ВК состоит из синтаксического анализатора, распараллеливателя циклов, модуля оценки качества распараллеливания, генераторов программ на параллельном ЯВУ или в машинных кодах.

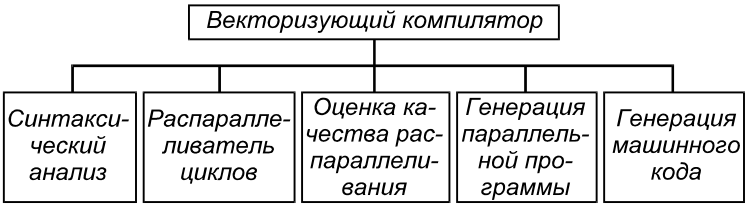


Рис.5.5. Состав функций векторизующего компилятора

На первом этапе работы ВК производится синтаксический анализ исходной последовательной программы и выдаются сообщения о найденных ошибках, заполняются таблицы входных и выходных переменных для всех циклов, а также проверяется выполнение необходимых требований: линейность индексных выражений, отсутствие операторов и обращений к подпрограммам и т. п. Кроме того, на этапе синтаксического анализа строится так называемый временной профиль программы, в котором оценивается время выполнения программы. Недостающую информацию для такой оценки компилятор может запрашивать в диалоговом режиме. Очевидно, что циклы, имеющие наибольший вес, необходимо распараллеливать в первую очередь.

На втором этапе циклы, для которых выполняются все ограничения, анализируются блоком распараллеливания. В этом блоке реализован один или несколько методов распараллеливания. Анализ производится начиная с вложенных циклов. Для циклов, которые не распараллеливаются, выдается список “узких мест”, т. е. информация о конкретных причинах: условие Рассела, информационная зависимость итераций и т. д.

На третьем этапе оценивается достигнутая степень распараллеливания для каждого цикла и на основе этой информации вычисляется ускорение параллельной программы по сравнению с исходной последовательной.

На четвертом этапе, если достигнутое ускорение удовлетворяет программиста, ВК может сгенерировать параллельную программу на ЯВУ (например, на параллельном Фортране) или программу в машинных кодах, уже готовую к выполнению. Если же достигнутая степень параллелизма неудовлетворительна, то программист, используя полученный список “узких мест”, может попробовать преобразовать циклы таким образом, чтобы они стали векторизуемыми. Если это не удается сделать, то необходимо менять алгоритм решения задачи.

Рассмотренная структура ВК является весьма обобщенной. В реальных ВК некоторые модули могут отсутствовать (например, модуль оценки качества или генератор программ на параллельном ЯВУ). Кроме того в ВК могут быть реализованы и некоторые дополнительные функции.

Как правило, ВК обрабатывает программу, написанную на стандартном ЯВУ: Фортран, PL-1, Паскаль, Си и др. В этом случае, чтобы включить в систему программирования ЭВМ новый язык, для него приходится реализовывать свой ВК.

Однако существуют и многоязыковые ВК. При использовании таких ВК программа с ЯВУ сначала транслируется в некоторую промежуточную форму (промежуточный язык), удобную для распараллеливания, и вся дальнейшая обработка производится на уровне промежуточного языка. В данном случае для включения нового языка в систему программирования необходимо к существующему ВК добавить транслятор с этого языка в промежуточную форму.

Векторизующий компилятор может иметь и несколько модулей генерации машинного кода, каждый из которых генерирует код для своего вида ЭВМ. Таким образом, возможно построение универсальных многоязыковых ВК, производящих распараллеливание программ на нескольких ЯВУ для разных типов ЭВМ.

§ 5.4. Языки программирования для транспьютерных систем

Разнообразие и широкое распространение транспьютерных систем предъявляют повышенные требования к системам программирования для транспьютеров.

Прежде чем перейти к рассмотрению систем программирования, опишем концентуальную модель вычислений в мультитранспьютерной системе:

1. Вычислительный процесс представляет собой параллельное выполнение нескольких подпроцессов на различных транспьютерах в асинхронном режиме. Синхронизация осуществляется путем обмена сообщениями. Каждый параллельный процесс в мультитранспьютере может, в свою очередь, быть разбит на более мелкие подпроцессы, которые выполняются квазипараллельно в одном транспьютере в режиме разделения времени.

2. В однотранспьютерной системе запуск параллельных ветвей осуществляется командой “стартовать процесс” (операция *THREAD* в ЯВУ), которая по своим функциям подобна оператору FORK: заводится критический блок, и новый процесс планировщиком включается в очередь процессов, претендующих на время процессора. Команда “завершить процесс” соответствует оператору JOIN: процесс исключается из очереди, из счетчика критического блока вычитается единица и, если содержимое счетчика оказывается равным нулю, завершается вышестоящий процесс. Обмен между квазипараллельными процессами осуществляется через программные каналы, следовательно, общие переменные отсутствуют, то есть реализуется модуль МКМД-ЭВМ с раздельной памятью. При этом конкуренции за ресурсы нет, так как их распределение производится программистом на этапе написания программы.

Процессы разделяются на: срочные, которые выполняются по очереди до полного завершения каждого из них; и на несрочные, которые обслуживаются по кольцу с равным интервалом времени. Управление ими осуществляется встроенным планировщиком, реализованным микропрограммно или аппаратно.

В случае необходимости реализовать разделяемую общую память (например, буфер для нескольких процессов) пользователь обязан сам организовать синхронизацию с помощью семафоров.

3. Реальный физический параллелизм процессов достигается в мультитранспьютере (МТ). Чтобы обеспечить этот параллелизм, пользователь должен создать сеть транспьютеров и разместить части задачи в узлах этой сети.

Программное обеспечение транспьютерных систем включает системы программирования, операционные системы, пакеты при-

кладных программ и некоторые специальные средства. Основным элементом систем программирования являются языки программирования. В дальнейшем будут рассмотрены: язык Оккам-2, особенности языков высокого уровня Фортран и Си для транспьютеров, возможности программирования средством операционной системы Helios.

Оккам-2. Идеология транспьютеров содержит две составляющие: традиционную и параллельную. Традиционная часть соответствует работе процессора одиночного транспьютера, который является обычной последовательной ЭВМ. Если в системе появляется более одного транспьютера, необходим параллелизм в программах, каналы, распределение ресурсов. Эти две составляющие выражены и в языках программирования, в частности, в Оккаме.

В Оккаме имеются все средства для последовательного програмирования: константы, переменные, массивы, операторы присваивания, операторы циклов, вызывы подпрограмм и другое. Эту часть в дальнейшем мы рассматривать не будем.

Новое, “параллельное” в Оккаме — это в первую очередь [16]:

1. Конструкции выбора ALT, параллельного выполнения *PAR* и повторители *FOR*.

2. Операторы ввода *c*?*x* и выводы *c*!*x*, используемые для синхронизации взаимодействующих процессов по однонаправленным линиям связи.

3. Средства для описания конфигурации мультитранспьютера, распределения параллельных частей задачи по элементам МТ, то есть средства конфигурирования.

Поскольку конструкции по пунктам 1 и 2 рассматривались уже в § 3.2, то более подробно остановимся на средствах конфигурирования, в связи с тем, что в указанном параграфе они рассмотрены кратко.

Распределение процессов между несколькими процессорами в транспьютерной сети называется *конфигурированием* программы. В Оккам принят статический подход к конфигурированию, при котором программист заранее на этом этапе программирования должен определить, на каких узлах сети необходимо разместить процессы. В некоторых областях применения может быть более подходящим динамический подход, где до выполнения не определяется, на каком процессоре должен выполняться каждый процесс.

Если необходимо, язык Оккам можно использовать для кодирования механизмов динамической загрузки и выполнения процессов.

Для описания того, как программа распределяется по процессорам в сети, можно воспользоваться несколькими различными подходами. Например, программист может подготовить три отдельных файла, содержащих:

1) описание топологии сети;

2) описание структуры программы, то есть описание состава и связей независимых процессов;

3) описание того, как отображаются процессы и каналы на процессоры и линии связи.

В языке Оккам-2 эти три аспекта фактически объединены в одно описание, которое называется *"Описание конфигурации"*. Текст описывает Оккам-программу пользователя на самом высоком уровне и содержит достаточно информации для отображения программы на транспьютерную сеть. Это описание затем используется компилятором для создания загружаемого файла пользовательской программы.

Для того, чтобы сконфигурированная программа была загружена в сеть, необходимо, чтобы один из процессоров в сети был соединен с главной ЭВМ. Этот процессор называется *корневым процессором*. Между корневым процессором и всеми остальными в сети должен быть путь через транспьютерные линии связи. Корневой процессор любой сети должен быть описан как первый процессор в конфигурации. Нет необходимости включать в описание конфигурации линию связи, соединяющую главную ЭВМ с корневым процессором.

При описании конфигурации используются три оператора языка Оккам:

PLACED PAR
PROCESSOR *номер тип. процессора*
PLACE *канал* AT *адрес*

Операторы PLACED PAR и PROCESSOR совместно описывают распределение логических частей задачи по физическим транспьютерам, а оператор PLACE AT закрепляет логические связи между частями задачи, называемые каналами, за физическими связями транспьютеров.

Рассмотрим пример описания конфигурации для распределения кода программы между четырьмя транспьютерами в сети. Пример состоит из двух различных процессов: control и work, которые должны быть описаны на уровне описания конфигурации.

Процедура control выполняется на корневом транспьютере и включает процессы, соединенные с главной ЭВМ для посылки управляющих сообщений программе, выполняющейся на главной ЭВМ. Процедура work выполняется на остальных трех транспьютерах.

На рис. 5.6 показана логическая структура программы, хотя она может выполняться и на меньшем числе транспьютеров, в том числе и на одном. Все зависит от распределения частей программы по процессорам, задаваемым описанием конфигурации. Дальнейшее рассмотрение относится к исполнению программы четырьмя транспьютерами. Разбиение программы на логические блоки, которые можно выполнять параллельно, производится программистом.

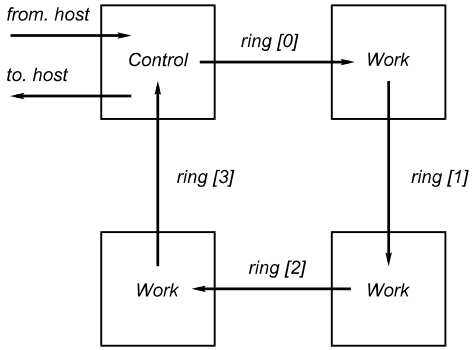


Рис.5.6. Логическая структура программы

Описание конфигурации содержит следующие компоненты:

- описание процедуры control;
- описание процедуры work;
- операторы PLACED PAR;
- отображение каналов на линии связи.

Процедура control имеет два канала — to. host и from. host, которые соединяют ее с главной ЭВМ. Эти каналы имеют канальный протокол MONITOR. Кроме того, имеется один канал для связи с конвейером, состоящим из процедур work, и один канал, связанный с последним процессом в конвейере. Эти каналы имеют канальный протокол PIPELINE. Интерфейс процедуры control выглядит следующим образом:

```
PROC control (CHAN OF MONITOR from. host, to. host
CHAN OF PIPELINE to. ring, from. ring)
```

Процедура work имеет вводной канал и выводной канал, каждый из которых имеет протокол PIPELINE. Интерфейс процедуры work имеет следующий вид:

```
PROC work (CHAN OF PIPELINE in, out)
```

Для описания конфигурации необходимо определить некоторые каналы, соединяющие все процессы друг с другом. Должны быть описаны два канала для связи прикладной программы с главной ЭВМ; они получили название from. host и to. host. Также должны быть описаны четыре канала для связи четырех транспьютеров друг с другом; они описаны как массив каналов от ring[0] до ring[3]. Все эти каналы показаны на рис. 5.6. Первый канал ring[0] соединяет процессор 0 с процессором 1 и так далее по кольцу. Ниже показано простое описание конфигурации без отображения каналов на линии связи:

```
CHAN OF MONITOR from. host, to. host:
[4]CHAN OF PIPELINE ring:
PLACED PAR
    PROCESSOR 0 T414
        control (from. host, to. host, rihg[0], ring[3])
    PROCESSOR 1 T414
        work (ring[0], ring[1])
    PROCESSOR 2 T414
        work (rihg[1], ring[2])
    PROCESSOR 3 T414
        work (ring[2], ring[3])
```

Теперь надо отобразить каналы на линии связи. Для этого необходимо для каждого процессора перечислить входные и выходные каналы. На рис. 5.6 даны имена каналов. Что же касается физических связей между транспьютерами, то они определяются топологией сети транспьютеров. Топология может разрабатываться как для решения одной задачи, так и для решения нескольких задач. Во втором случае физические связи не меняются при переходе от задачи к задаче. Но в любом случае перед выполнением задачи физические связи должны быть установлены вручную с помощью разъемов. В некоторых случаях физические связи между транспьютерами устанавливаются с помощью программируемого коммутатора.

Если “интеллект” компилятора позволяет производить закрепление логических каналов за физическими связями, то описание топологии должно включаться в состав описания конфигурации. В противном случае это закрепление производится программистом вручную.

Добавив полученное таким образом отображение линий связи к приведенному выше частичному описанию, получим следующее описание конфигурации:

```
CHAN OF MONITOR from. host, to. host:
[4] CHAN OF PIPELINE ring:
PLACED PAR
  PROCESSOR 0 T414
    PLACE from. host AT link0in:
    PLACE to. host AT link0out:
    PLACE ring[0] AT link2out:
    PLACE ring[3] AT link3in:
    control(from. host, to. host, ring[0], ring[3])
  PROCESSOR 1 T414
    PLACE ring[0] AT link3in:                    (5.5)
    PLACE ring[1] AT link2out:
```

```
        work (ring[0], ring[1])
    PROCESSOR 2 T414
        PLACE ring[1] AT link3in:
        PLACE ring[2] AT link2out:
        work (ring[1], ring[2])
    PROCESSOR 3 T414
        PLACE ring[2] AT link3in:
        PLACE ring[3] AT link2out:
        work (ring[2], ring[3])
```

Наконец, заметим, что все три процессора, на которых выполняются процессы work, имеют очень похожую структуру. Можно воспользоваться этой регулярностью и описать все три процессора одним оператором с использованием конструкции *replicated* PLACE PAR. Ниже приведена конечная версия описания конфигурации, где используется конструкция повторения с индексом *i*, которая принимает значения 1, 2 и 3:

```
CHAN OF MONITOR from. host, to. host:
[4]CHAN OF PIPELINE ring:
PLACED PAR
    PROCESSOR 0 T414
        PLACE from. host AT link0in:
        PLACE to. host AT link0out:
        PLACE ring[0] AT link2out:
        PLACE ring[3] AT link3in:                          (5.6)
        control(from. host, to. host, ring[0], ring[3])
    PLACED PAR i = 1 FOR 3
        PROCESSOR i T414
            PLACE ring[i -1] AT link3in:
```

```
PLACE ring[i] AT link2out:
work (ring[i - 1], ring[i])
```

Языки высокого уровня. Несмотря на эффективность языка Оккам постоянно существует стремление использовать для программирования транспьютеров традиционные последовательные ЯВУ, особенно если учесть, что вычисления в пределах одного транспьютера являются чисто последовательными. Здесь осуществлено несколько подходов [15].

Наиболее очевидный способ заключается в том, чтобы описывать взаимодействие транспьютеров на Оккаме, а описание работы внутри одного транспьютера на Фортране или Си включать как фрагмент в эту программу. Этот способ особенно привлекателен, потому что имеется уже большой объем прикладного программного обеспечения на ЯВУ. Однако включение фрагментов на ЯВУ в программу на Оккаме распространения не получил по следующим причинам: необходимым оказалось изучение нового и достаточно необычного языка Оккам. Кроме того, включение фрагментов на ЯВУ в Оккам является сложной, кропотливой процедурой, чреватой ошибками.

Другой подход состоял в том, чтобы создавать языки, в которых содержались бы собственные средства для описания параллелизма процессов и распределения ресурсов между этими процессами.

Главное требование к таким ЯВУ, пусть даже в ущерб эффективности, чтобы они строились на базе распространенных последовательных ЯВУ с минимумом изменений и дополнений.

Средства для описания параллелизма процессов могут вводиться двумя способами: путем внесения изменения в синтаксис языка и расширением языка за счет введения специальных процедур. Примером первого подхода является использование операторов типа par {операторы}. Операторы в скобках задают процессы, которые будут выполняться квазипараллельно (в режиме разделения времени). Совместно с *par* { } могут использоваться повторители *for*.

Для работы с каналами транспьютера введен новый тип данных — канал, который используется в операторах присваивания.

Присваивание обычной переменной значения переменной типа канал означает чтение из канала значения и размещение его в первой из упомянутых переменных. Обратное присваивание задает помещение в канал, имя которого определяется именем переменной типа канал, значения переменной обычного типа, указанной справа от знака присваивания. При описании канальной переменной с указанием адреса физического канала можно таким образом определить передачу сообщения между процессами в разных транспьютерах, связанных этим каналом.

В этом же языке имеется конструкция *CASE*, аналогичная по функциям конструкции *ALT* в Оккаме.

Другой подход в описании параллелизма процессов состоит в расширении обычных ЯВУ за счет специальных процедур. Для определенности будем рассматривать язык Фортран. В нем используются следующие процедуры этапа исполнения: *THREAD*, *SEMA*, *TIMER*, *CHAN*, *NET*, *ALT*. Рассмотрим функции, выполняемые некоторыми из этих процедур.

Процедура *THREAD* имеет разновидности *F77 THREAD START*, *F77 THREAD CREATE* и ряд других.

Вызов процедуры сопровождается набором аргументов, например:

```
CALL F77 THREAD START (SUB, WSARRAY, WSSIZE,
FLAGGS, NARGS, ARG1, . . . , ARGN)
```

Здесь *SUB* — имя запускаемой процедуры, которая будет выполняться в режиме разделения времени на одном транспьютере совместно с основной ветвью; *WSARRAY*, *WSSIZE* — начальный адрес и размер рабочей области; *FLAGGS* — приоритет ветви, *NARG* — количество аргументов процедуры, *ARG1*, ..., *ARGN* — формальные параметры процедуры. Из приведенных выше описаний видно, что процедура *THREAD* в общем выполняет те же функции, что и команды “стартовать процесс” и “завершить процесс” языка Оккам.

Управление запускаемой ветвью осуществляется встроенным планировщиком транспьютера.

Все ветви, запускаемые поочередно с помощью *THREAD*, выполняются в одном транспьютере.

Своеобразие языка Фортран состоит в том, что подпрограммы языка Фортран не реентерабельны, поэтому процедуры *THREAD* являются разделяемым многими ветвями ресурсом и защищаются от одновременного использования семафором, который создается с помощью процедуры *F*77 *SEMA*.

Процедура *CHAN* также имеет ряд модификаций. Модификация:

F77 CHAN IN MESSAGE (LENGTH, BUFFER, ICHANADDR)

позволяет ввести сообщение через канал с именем *ICHANADDR*, а модификация:

F77 CHAN OUT MESSAGE (LENGTH, BUFFER, ICHANADDR)

— вывести сообщение через канал. Таким образом, обе процедуры выполняют те же функции, что и операторы *c*?*x* и *c*!*x* в языке Оккам.

С помощью процедуры *CHAN* обеспечивается связь как между ветвями (*THREAD*) одной задачи, выполняемой в одном транспьютере, так и между ветвями, выполняемыми в разных транспьютерах. Эти ветви называются в параллельном Фортране *TASK*.

Выше были представлены два способа описания параллельных процессов: путем расширения синтаксиса за счет новых операторов и путем использования библиотечных процедур. Однако, для обоих языков остался нерешенным вопрос о распределении ресурсов между параллельными частями задач. В МКМД-ЭВМ с общей памятью ресурсы распределяются динамически (см. § 3.1). В языке Оккам распределение задается пользователем статически на этапе написания программы с помощью языка конфигурации. Аналогичный язык используется и в упоминавшихся выше ЯВУ.

В языке Фортран имеются средства, позволяющие организовать выполнения по принципу “фермы”. В “ферме” имеется программа “мастер”, которая выдает порции работы программам, называемым “работники” по мере их освобождения от предыдущей порции работы. Количество программы “работник” столько, сколько транспьютеров в системе. В режиме “ферма” от пользователя не требуется распределять ресурсы, это делается автоматически программным обеспечением мультитранспьютера.

Helios. Одной из наиболее распространных операционных систем для транспьютеров является система *Helios*, разработанная фирмой Perihelion Software Ltd. Helios называется распределенной, поскольку ее ядро находится в каждом транспьютере мультитранспьютерной сети.

Helios аккумулирует в себе все возможности параллельного программирования, описанные выше, и добавляет новые.

Helios построена на базе распространенной ОС *UNIX* и включает в себя все возможности *UNIX*, в том числе и язык управления и программирования SHELL. С другой стороны, HELIOS является надстройкой над операционной системой базовой машины, которая используется мультитранспьютером как средство ввода-вывода и для связи с пользователем. Обычно, это ПЭВМ типа IBM PC. Следовательно, HELIOS является надстройкой над MS DOS.

Helios выполняет все функции обычных операционных систем: организацию ввода-вывода, поддержку дисков, управление файлами, редактирование, связь с пользователем и др.

В Helios имеется ряд возможностей для параллельного программирования: на языке Shell, на языках Фортран, Си. Однако, особенностью этой ОС является наличие *языка CDL* (Component Distribution Lauguage), предназначенного для описания связей между подзадачами, которые в CDL называются компонентами.

Компоненты могут быть написаны на традиционных ЯВУ или на языке Shell.

Программа на CDL перечисляет компоненты, подлежащие выполнению в транспьютерной сети, задает порядок их выполнения, описывает связи между ними. Рассмотрим основные конструкции CDL.

CDL содержит описательную и исполнительную части. Описательная часть позволяет описать компоненты задачи с нестандартными требованиями, например, описание:

```
component numbercrunch {
    processor T800;
    memory 32768;
    streams < left, > right
    }
```

требует, чтобы компонент numbercrunch был расположен на транспьютере T800 с объемом памяти не мене 32 Кбайт. Кроме того, этот компонент связан с внешним миром двумя потоками сообщений, имена которых указаны после ключевого слова streams: из потока left производится чтение сообщения (знак “<” указывает на чтение), а в поток right производится запись сообщений (знак “>” указывает на запись сообщений).

Характер взаимодействия между компонентами определяется параллельными конструкторами и повторителями.

Если в задании больше двух компонентов, то каждая пара рядом стоящих конструкторов должна быть разделена одним из четырех конструкторов. Список конструкторов в порядке возрастания старшинства выглядит следующим образом: ∧∧, |||, |, <>.

Простой конструктор задает параллельное исполнение двух компонент, которые не обмениваются сообщениями, например:

```
square ^^ square
```

Здесь показано, что два независимых компонента случайным образом получают данные из входного потока, а их выходные данные также будут перемежаться в выходном потоке.

Конструктор “|||” определяет режим работы “ферма”, например, выражение:

```
power [4] ||| square
```

обозначает, что выходные данные от компонента power будут по мере поступления автоматически пересылаться случайным образом выбранному одному из четырех компонентов square, которые ожидают поступления информации на свой вход.

Конструктор “|” описывает паралельный конвейер из двух компонент, например, выражение:

```
square | square
```

позволяет получить на выходе величину, если на вход подается значение *x*.

Двухсторонний конструктор “<>” описывает двухсторонний обмен между двумя компонентами. Левый компонент считается главным, правый — подчиненным. Главный компонент передает

сообщение подчиненному, который затем, подобно подпрограмме, возвращает ответ. При этом, главный компонент может продолжать работу сразу после пересылки сообщения, то есть параллельно с подчиненным.

Для удобства определения в задании несколько подзаданий из одинаковых и однотипно связанных компонент используются повторители. Повторитель содержит число в квадратных скобках, например: задание

A [3] <> B

эквивалентно заданию

A <> B <> B <> B

Повторитель может быть также задан с помощью диапазона значений. Например, повторитель [*i*<1, *j*<1] задает четыре значения: {0, 0}, {0, 1}, {1, 0}, {1, 1}. Символ "<" здесь указывает на верхнюю (включаемую) границу диапазона.

Используя такого рода поименованные повторители, можно строить сложные массивы копий компонента и определять их взаимные связи. Например, с использованием поименованных повторителей *i* и *j* двумерный массив размера 4×4 копий компонента *B*, связанных двусторонними потоками сообщение в top, мог бы выглядеть так:

```
component B [i, j]{
    streams, , , ,
                    < riqht {i, j}              > riqht {i, (j+1)%4},
                    < left {i, (j+1)%4},        > left {i, j}
                    < down {i, j},              > down {(i+1)%4, j}
                    < up {(i+1)%4, j}           > up {i, j},
    ;                                                              (5.7)
}
<> [i<4, j<4] B{i, j}
```

В (5.7) знак %4 означает "по модулю 4".

После ключевого слова *streams* имеется ряд запятых, между которыми не содержатся имена потоков. Это означает, что в CDL эти имена подразумеваются по умолчанию: первое пропущенное имя обозначает поток, принимаемый со стандартного устройства

ввода *stdin*; второе пропущенное имя определяет поток, выводимый на стандартное устройство вывода *stdout*; третье — описывает поток, выводимый на стандартное устройство сообщений об ошибках *stderr*; наконец, четвертое место соответствует потоку, выводимому на стандартное устройство для отладки *stddbg*.

Последний оператор в (5.7) соответствует исполнительной части этой программы и задает характер взаимодействия всех компонентов.

Программа (5.7) на языке CDL затем с помощью компилятора CDL, входящего в состав Helios, автоматически преобразуется в программу загрузки элементов транспьютера соответстующими компонентами, между которыми устанавливаются физические связи, реализующие связи из (5.7). Загрузка производится с учетом физической конфигурации ресурсов, хранящихся в конфигурационном файле, о котором упоминалось ранее. Естественно, физические связи должны допускать те связи, которые требуются согласно программе на CDL. Если это не выполняется, то данная программа на CDL не может быть выполнена. В этом случае необходимо либо менять схему физических связей, либо "подгонять" программу на CDL на существующую систему физических связей.

Среди прикладных пакетов имеются многочисленные пакеты, написанные на ЯВУ и ассемблере для научных и инженерных расчетов, для обработки графических изображений, для цифровой обработки сигналов и др.

§ 5. 5. Программное обеспечение и служебные алгоритмы суперскалярных процессоров

Наиболее очевидным достоинством суперскалярных вычислительных систем является традиционная система программирования на последовательных языках. Это делает возможным массовое использование суперскалярного параллелизма. Однако параллельный характер вычислений предъявляет дополнительные требования к системному программному обеспечению. Системное обеспечение должно выполнять следующие новые функции:

1. Выявлять и представлять в явной форме скрытый параллелизм последовательных программ, то есть обеспечить построение ЯПФ. Этот этап обычно является машинно-независимым.

2. Обеспечивать планирование (упаковку) полученной ЯПФ на имеющиеся ресурсы. Этот этап является машинно-зависимым. В результате планирования также получается ЯПФ, но в каждом ее ярусе содержатся только те операции, для которых в данном такте имеются ресурсы. Планирование производится с учетом длительности выполнения операций.

3. Поскольку уровень параллелизма обычных пользовательских программ невысок и не всегда этот параллелизм позволяет загрузить имеющиеся ресурсы (конвейеры), то одной из наиболее важных задач суперскалярного программирования является задача повышения уровня параллелизма последовательных программ.

4. Суперскалярные МП, как правило, имеют новую систему команд и "выжить" они могут только в том случае, если для них будет быстро создан большой объем не только системного, но и прикладного программного обеспечения. Поскольку разработка огромного числа прикладных пакетов для новой системы команд слишком трудоемка, возникает задача автоматического переноса этих пакетов, уже созданных в основном для МП *i*80×86 и МС 680×0.

Некоторые способы решения этих задач и будут рассмотрены далее.

Алгоритмы построения ЯПФ в значительной степени зависят от того, на языке какого уровня записано распараллеливаемое выражение.

Существует много способов распараллеливания арифметико-логических выражений на ЯВУ. Все они прямо или косвенно (через польскую запись) используют анализ приоритетов выполняемых арифметико-логических операций. Рассмотрим один из прямых методов.

Присвоим операциям следующие приоритеты P:

1. P (↑) = 3
2. P (∗), P(/) = 2
3. P (+), P(-) = 1

Алгоритм состоит в том, что исходное выражение многократно сканируется слева направо. При каждом сканировании выделяются наиболее приоритетные операции. Рассмотрим пример:

ВХОД	: a+b+c∗d∗l∗f+g	
ПРОХОД	1: T1+T2∗T3+g;	T1=a+b; T2=c∗d; T3=l∗f
	2: T4+t5; T4=T2∗T3;	T5=T1+g
	3: T6;	T6=T4+T5

Полученная ЯПФ представлена на рис. 5.7.

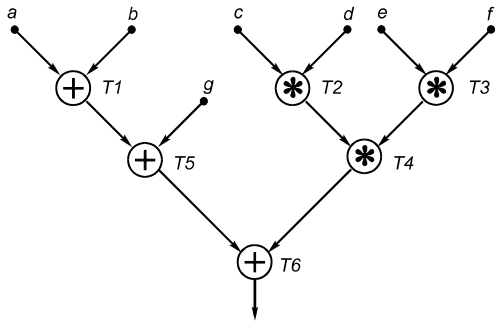


Рис. 5.7. Пример ЯПФ построенной по методу приоритетов

Параллелизм команд программы, представленной на уровне ассемблера, также определяется графом информационных зависимостей. При этом на этапе распараллеливания могут применяться различные методы устранения зависимостей с целью повышения параллелизма.

Построение ЯПФ для отрезка программы производится путем определения зависимости пар смежных команд при анализе программы сверху вниз согласно алгоритму на рис. 5.8. Для своего выполнения алгоритм требует порядка n^2 операций, где n — число команд в отрезке программы.

Ниже приводится пример последовательной программы (слева) и ее ЯПФ, построенной по описанному алгоритму (справа)

```
mov eax, [esi]          mov eax, [esi]    dec ecx
imul eax, [ebx]         imul eax, [ebx]   add esi, 4
add [edi], eax          add [edi], eax    add ebx, 4
add ebx, 4
add esi, 4
dec ecx
```

Построение ЯПФ и упаковка на ЯВУ производятся в общем случае для многоместных операторов, в то время как в реальной

вычислительной системе параллельные АЛУ или процессоры выполняют только двухместные операции, соответствующие командам машинного языка. Отсюда следует, что в реальной системе построение ЯПФ и упаковка должны производиться на уровне системы команд (ассемблер, двоичный код), а не на ЯВУ.

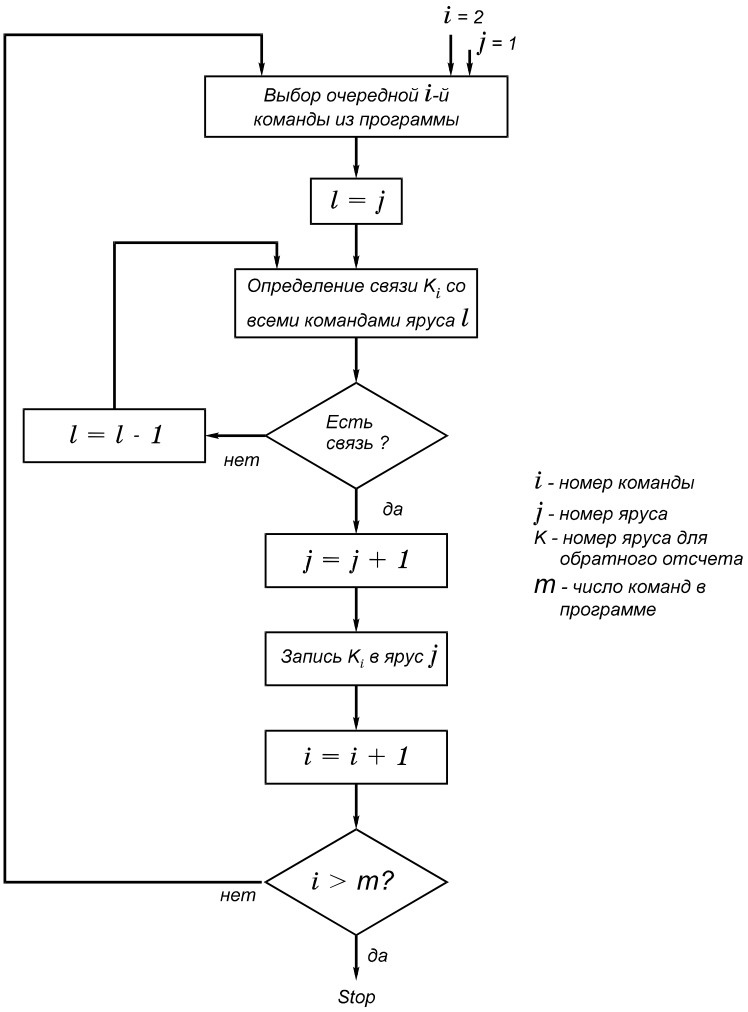


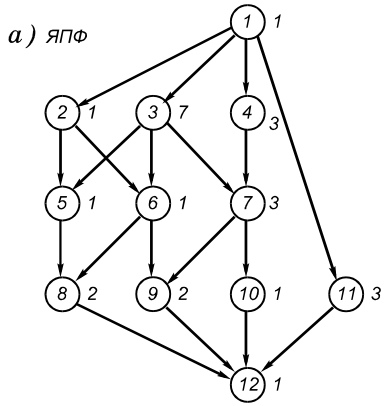
Рис.5.8. Алгоритм построения ЯПФ на языке машинного уровня

Планирование является оптимизационной задачей и уже при небольшом числе вершин информационного графа, точнее, ЯПФ, время перебора при распределении ресурсов будет недопустимо большим.

Чтобы уменьшить время планирования, используют эвристические алгоритмы, среди которых наибольшее распространение получили *списочные расписания* [21], которые при линейном росте времени планирования при увеличении числа вершин, дают результаты, отличающиеся от оптимальных всего на 10-15%.

В таких расписаниях оператором присваиваются приоритеты по тем или иным эвристическим правилам, после чего операторы упорядочиваются по убыванию или возрастанию приоритета в виде линейного списка. В процессе планирования затем осуществляется назначение операторов процессорам в порядке их извлечения из списка. Ниже дан пример построения списочного расписания.

На рис. 5.9, а представлена исходная ЯПФ. Номера вершин даны внутри кружков, а время исполнения — около вершин. На рис. 5.9, б приведены уровни вершин. Под уровнем вершины понимается длина наибольшего пути из этой вершины в конечную, то есть длина критического по времени пути из данной вершины в конечную.



б) Уровни вершин

Наиб. кр. вр. путь КОН		Наиб. кр. оп. путь КОН		Произв. выбор
П	В	П	В	В
14	1	5	1	9
13	3	4	4	7
9	4	4	2	11
6	7	4	3	2
5	2	3	5	12
4	6	3	7	10
4	5	3	6	5
4	11	2	10	3
3	9	2	9	6
3	8	2	8	1
2	10	2	11	8
1	12	1	12	4

1	2	3	4	5	6	7	8	9	10	11	12	номера вершины
14	5	13	9	4	4	6	3	3	2	4	1	уровни

Рис.5.9. Примеры двухпроцессорных списочных расписаний

в) Приоритеты и списки вершин
(П — приоритеты, В — номера вершин)

г) Реализация расписаний
(Р1, Р2 — процессоры, Х — простой процессора)

Наиб. кр. вр. путь КОН

Р1	1	3	3	3	3	3	3	3	7	7	7	9	9	12
Р2	Х	4	4	4	2	11	11	11	6	5	8	8	10	Х

Наиб. кр. оп. путь КОН

P1	1	4	4	4	11	11	11	X	X	5	6	8	8	9	9	12
P2	X	2	3	3	3	3	3	3	3	7	7	7	10	X	X	X

Произвольный выбор

P1	1	11	11	11	4	4	4	X	X	7	7	7	9	9	12	
P2	X	2	3	3	3	3	3	3	3	5	6	8	8	10	X	

Рис.5.9. Примеры двухпроцессорных списочных расписаний (продолжение)

Построим для примера три расписания из множества возможных.

Для каждого из них найдем характеристики всех вершин по соответствущим алгоритмам и примем значения этих характеристик в качестве величин приоритетов этих вершин.

В первом расписании величина приоритета вершины есть ее уровень. Это расписание НАИБ. КР. ВР. ПУТЬ КОН (наибольший критический по времени путь до конечного оператора). Во втором расписании величина приоритета вершины есть ее уровень без учета времени исполнения, то есть здесь веса всех вершин полагаются одинаковыми (единичными). Это расписание НАИБ. КР. ОП. ПУТЬ КОН (наибольший критический по количеству операторов путь до конечного оператора). В третьем расписании величины приоритетов полагаются одинаковыми и последовательность выборки вершин определяется случайно на равновероятной основе, но с учетом зависимостей вершин. Это расписание ПРОИЗВОЛЬНЫЙ ВЫБОР используется в целях сравнения с другими для определения их эффективности.

Об оптимальности расписания можно судить по количеству простоев процессоров. Первое расписание дает 2, второе — 5, третье — 4 такта простоя.

Серьезным препятствием для использования суперскалярных МП является низкий уровень параллелизма прикладных программ. Рассмотрим этот вопрос на примерах.

Скалярный параллелизм определяется внутри базового блока, под которым понимают линейный участок программы. Ниже при-

ведена программа, содержащая три базовых блока, а на рис. 5.10 построены ЯПФ этих ББ.

```
        DO 1 I = 1, 100
ББ1     Z(I) = A(I)**2*X + B(I) * X + C(I)
        IF (I - 30) 2, 2, 3
ББ2   2 R1(I) = Z1*I
        R(I) = R1(I) * Z(I)
        GO TO 1                                          (5.8)
ББ3   3 R1(I) = Z1 * I
        R2(I) = Z2 * I
        R3(I) = Z3 * I
        R1 = R3(I)*Z(I)**3 + R2(I)*Z(I)**2 + R1(I)*Z(I)
      1 CONTINUE
```

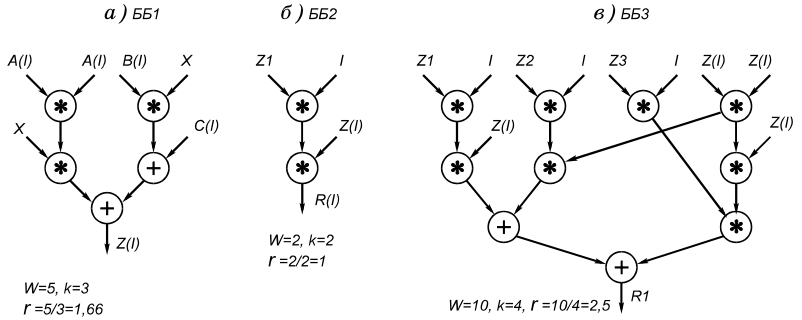


Рис.5.10. ЯПФ исходных базовых блоков

Если не учитывать длительность операций, то из рис.5.10 следует, что $w_{ББ3}>w_{ББ1}>w_{ББ2}$ (w — количество операций в ББ, k — число ярусов его ЯПФ) и $r_{ББ3}>r_{ББ1}>r_{ББ2}$. Это приводит к мысли, что параллелизм растет с увеличением размера ББ, следовательно, надо увеличивать длину ББ. Этого можно достигнуть двумя спо-

собами: путем объединения базовых блоков и путем развертки итераций цикла.

Рассмотрим обе возможности. Очевидно, цикл (5.8) можно разбить по параметру *I* на два независимых цикла (5.9) и (5.10).

```
      . . .
      DO 1 I = 1, 30
ББ4   Z(I) = A(I)**2*X + B(I)*X + C(I)
      R1(I) = Z1*I (5. 9)                                   (5.9)
      R(I) = R1(I)*Z(I)
    1 CONTINUE
```

```
      . . .
      DO 1 I = 31, 100
ББ5   Z(I) = A(I)**2*X + B(I)*X + C(I)
      R1(I) = Z1*I                                          (5.10)
      R2(I) = Z2*I
      R3(I) = Z3*I
      R(I) = R3(I)*Z(I)**3 + R2(I)*Z(I)**2 + R1(I)*Z(I)
    1 CONTINUE
```

В (5.9) блок ББ4 объединяет блоки ББ1 и ББ2, а в (5.10) блок ББ5 объединяет ББ1 и ББ3. Соответствующие ЯПФ представлены на рис. 5.11, а, б.

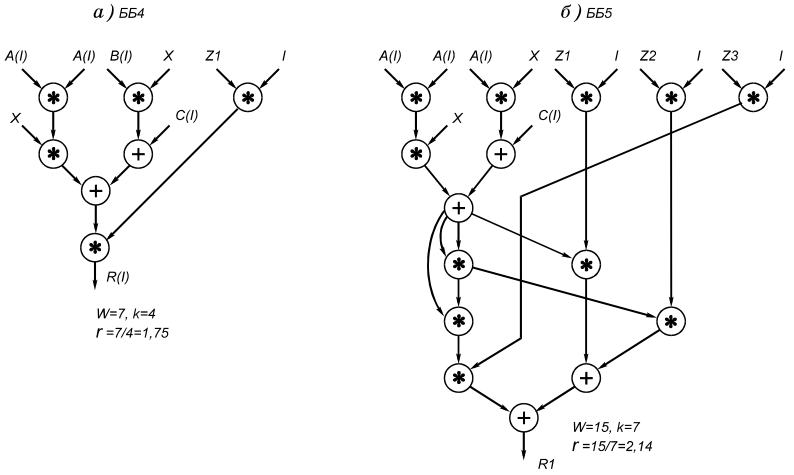


Рис. 5.11. ЯПФ укрупненных базовых блоков

Из рисунка следует, что ББ4 обладает лучшими характеристиками, чем ББ1 и ББ2, взятыми по отдельности, поскольку $r_{ББ4} > r_{ББ1,ББ2}$. Однако, для ББ5 дела обстоят по другому:

$$w_{ББ5} = w_{ББ1} + w_{ББ3} = 5+10 = 15$$
$$k_{ББ5} = k_{ББ1} + k_{ББ3} = 3+4 = 7,$$

поэтому и ускорение ББ5 занимает промежуточное значение: $r_{ББ1} > r_{ББ5} > r_{ББ3}$. Это означает, что для ББ5 улучшения характеристик не произошло. Однако, эксперименты показывают, что в большинстве случаев объединение блоков дает увеличение ускорения.

Цикл (5.11) является разверткой трех итераций цикла (5.9).

Соответствующая ЯПФ приведена на рис. 5.12. Ускорение возрастает более, чем линейным образом: $r_{ББ6} > 3 \cdot r_{ББ1}$. Правда, при подсчете операций в ББ6 индексные операции не учитывались.

```
      . . .
      DO 1 I = 1, 30, 3
ББ6   Z(I) = A(I)**2*X + B(I)*X + C(I)
      R (I) = Z1*I                                    (5. 11)
```

```
      R(I) = R1(I)*Z(I)
      Z(I+1) = A(I+1)**2*X + B(I+1)*X + C(I+1)
      R1(I+1) = X1 * (I+1)
      R(I+1) = R1(I+1) * Z(I+1)
      Z(I+2) = A(I+2)**2*X + B(I+2)*X + C(I+2)
      R1(I+2) = Z1*(I+2)
      Z(I+2) = R1(I+2)*Z(I+2)
1     CONTINUE
```

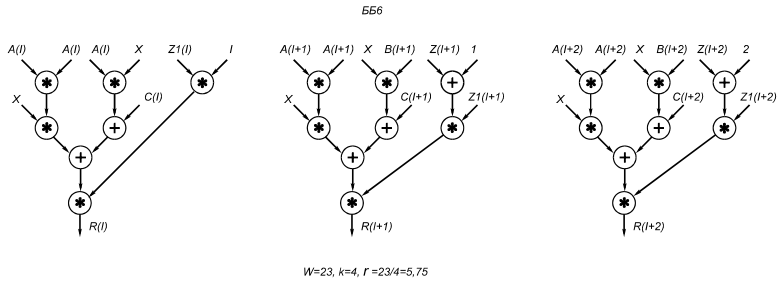


Рис. 5.12. ЯПФ развертки итераций

Развертка (5.11) включает не только параллелизм одного ББ, но и параллелизм трех смежных итераций, то есть в обращение вовлекается и векторный параллелизм, который, однако, рассматривается здесь как частный случай скалярного.

На рис. 5.13 точками представлена зависимость $r = f(w)$ для рассмотренных ранее ББ. Для получения более достоверных результатов проводились специальные исследования на большом объеме программного материала. Полученные результаты ограничены контуром на этом же рисунке. На основе рис. 5.13 с учетом вероятностных характеристик контура результатов можно получить следующую качественную зависимость:

$$r = a + b\,w, \qquad (5.\,12)$$

где a и b — константы ($a \approx 1$, $b \approx 0{,}15$)

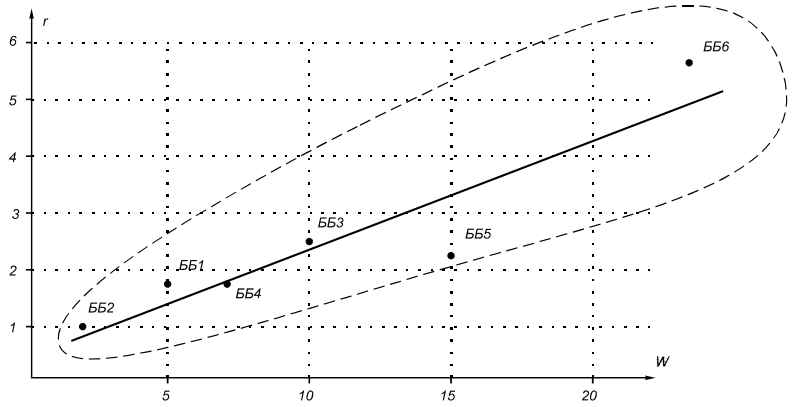


Рис.5.13 Рост ускорения в зависимости от размера базового блока

Таким образом, основной путь увеличения скалярного параллелизма программы — это удлинение ББ, а развертка — наиболее эффективный способ для этого. К сожалению, в большинстве случаев тело цикла содержит операторы переходов. Это препятствует как объединению ББ внутри тела цикла, так и выполнению развертки. Существуют частные способы преодоления этого препятствия. Один из них был продемонстрирован на примере разбиения программы (5.8) на два независимых цикла (5.9) и (5.10) благодаря тому, что оператор IF прямолинейно зависит от величины индекса цикла.

Более универсальный *метод планирования трасс* предложил в 80-е годы Фишер (Fisher, США) [22].

Рассмотрим этот метод на примере рис. 5.14, на котором представлена блок-схема тела цикла.

Как и прежде, в кружках представлены номера вершин, а рядом — вес вершины (время ее

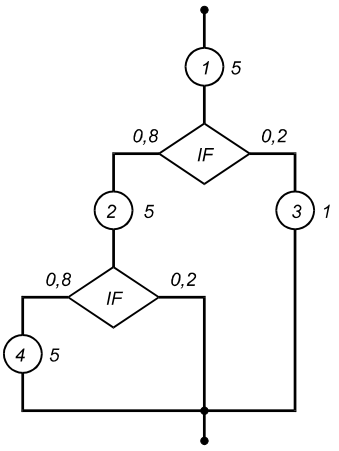


Рис. 5.14. Пример выбора трасс

исполнения); на выходах операторов переходов проставлены вероятности этих переходов.

Возможны следующие варианты исполнения тела цикла: 1-2-4, 1-2, 1-3. Путь 1-2-4 обладает наибольшим объемом вычислений (5+5+5) и является наиболее вероятным. Примем его в качестве главной трассы. Остальные пути будем считать простыми трассами.

Ограничимся рассмотрением метода планирования трасс только по отношению к главной трассе.

Если метод планирования трасс не применяется, то главная трасса состоит из трех независимых блоков, суммарное время выполнения которых будет в соответствии с (5.12) равным:

$$T_1 = 0{,}64\frac{3\cdot 5}{a+b\cdot 5} = 0{,}64\frac{3\cdot 5}{1+0{,}15\cdot 5} = 5{,}5$$

В методе планирования трасс предлагается считать главную трассу единым ББ, который выполняется с вероятностью 0,64. Тем не менее, при каждом выполнении этого объединенного блока проверяются условия выхода из главной трассы, и в случае необходимости выход осуществляется. Если переходов нет, то объединенный ББ выполняется за время

$$T_2 = 0{,}64\frac{3\cdot 5}{1+0{,}15\cdot 3\cdot 5} = 3$$

Таким образом, выигрыш во времени выполнения главной трассы составил $T_1 / T_2 = 1{,}8$ раз. В общем случае при объединении k блоков с равным временем исполнения w получаем:

$$\frac{T_1}{T_2} = \left(\frac{k\cdot w}{a+b\cdot w}\right) / \left(\frac{k\cdot w}{a+b\cdot k\cdot w}\right) = \frac{a+b\cdot kw}{a+b\cdot w} \xrightarrow[w\to\infty]{} k$$

При построении ЯПФ объединенного ББ и дальнейшем планировании команды могут перемещаться из одного исходного ББ в другой, оказываясь выше или ниже оператора перехода, что может привести к нарушению логики выполнения программы. Чтобы исключить возможность неправильных вычислений, вводятся компенсационные коды. Рассмотрим примеры (рис. 5.15)

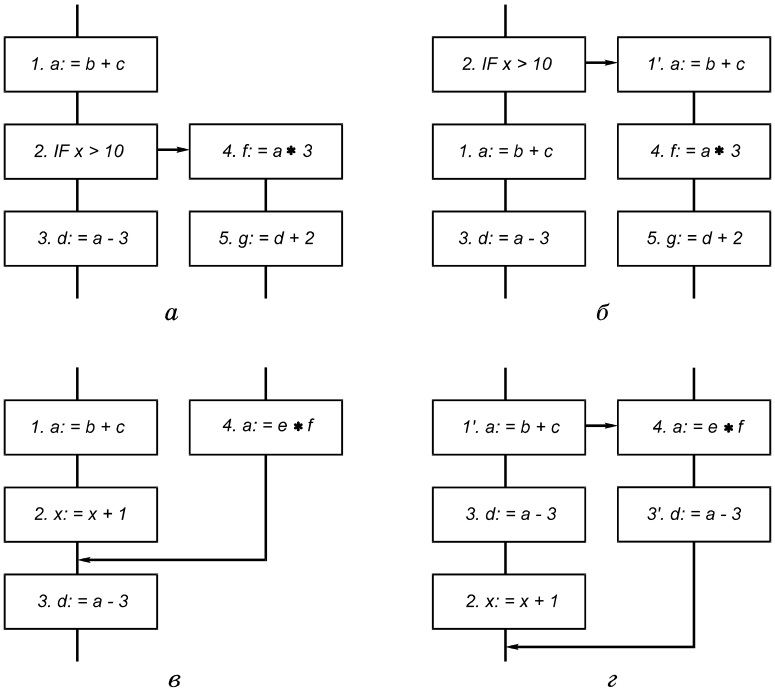


Рис.5.15. Способы введения компенсационных кодов

Пусть текущая трасса (рис. 5.15, а) состоит из операций 1, 2, 3. Предположим, что операция 1 не является срочной и перемещается поэтому ниже условного перехода 2. Но тогда операция 4 считает неверное значение *a*. Чтобы этого не произошло, компилятор водит компенсирующую операцию 1 (рис. 5.15, б).

Пусть теперь операция 3 перемещается выше IF. Тогда операция 5 считает неверное значение *d*. Если бы значение *d* не использовалось на расположенном вне трассы крае перехода, то перемещение операции 3 выше IF было бы допустимым.

Рассмотрим переходы в трассу извне. Пусть текущая трасса содержит операции 1, 2, 3 (рис. 5.15, в).

Предположим, что компилятор перемещает операцию 3 в положение между операциями 1 и 2. Тогда в операции 3 будет ис-

пользовано неверное значение *a*. Во избежание этого, необходимо ввести компенсирующий код 3. (рис. 5.15, г).

Существует полный набор правил введения компенсационных кодов, которые используются в компиляторах для метода планирования трасс. Компенсационные коды увеличивают размер машинной программы, но не увеличивают числа выполняемых в процессе вычислений операций.

Задача переноса программного обеспечения наиболее просто решается, если суперскалярный МП строится на базе системы команд типа *CISC*, как в случае МП Пентиум, для которого без переделки пригодно все ПО, написанное для семейства МП *i*80×86. Однако, большинство суперскалярных МП строится на более перспективной системе команд типа *RISC*, например, семейство МП Power PC (совместная разработка фирм APPLE, IBM, MOTOROLA) и семейство Alpha (фирма DEC). В этом случае проблема переноса значительно усложняется.

Для системного ПО (ОС, компиляторы, редакторы и др.) перенос осуществляется относительно просто, поскольку объем системного ПО ограничен и для него существуют и доступны исходные программы на ЯВУ или ассемблере. В этом случае перенос осуществляется на основе прямой перекомпиляции в коды нового МП.

Объем прикладного ПО значительно больше, чем системного, причем, прикладные пакеты обычно существуют в виде двоичных кодов, а исходные тексты недоступны. В этом случае перенос может осуществляться на основе эмуляции. Под э*муляцией* понимают преобразование команд исходной программы, представленной в двоичных кодах, в команды нового МП. Это преобразование выполняется в режиме интерпретации исходных команд непосредственно в процессе вычислений.

Пусть исходная программа в двоичных кодах представлена в системе команд CKi и ее требуется преобразовать в двоичный код программы в системе команд CKj. Тогда эмуляцией называется непосредственное выполнение исходной программы в режиме покомандной интерпретации, то есть каждая команда из CKi заменяется на одну или несколько команд из CKj, которые сразу же и выполняются. Эмулятор реализуется программно или микропро-

граммно. Естественно, режим интерпретации значительно (в несколько раз) уменьшает скорость вычислений.

При переносе пакета, кроме непосредственной эмуляции кодов операции команд, нужно выполнить еще ряд условий, осложняющих эмуляцию:

1. Обычно пакет требует определенного (иногда, специфичного) набора ВнУ, библиотечных подпрограмм и функций, некоторых функций операционной системы, специализированного для пакета человеко-машинного интерфейса. Поэтому, если пакет переносится с ЭВМ с архитектурой Ai на ЭВМ с архитектурой Aj, то последняя должна удовлетворять всем требованиям пакета. Иногда это требует большой переработки пакета.

2. В процессе эмуляции должны быть отображены способы прерываний, способы вызова подпрограмм, система флагов и другие особенности исходной системы команд.

3. Обычно исходные программы представлены в системе команд типа *CISC*, которая использует небольшое число РОН (4...8). Такие программы обладают небольшим параллелизмом, поскольку из-за малого числа РОН в них внесены дополнительные зависимости по данным. В *RISC*-микропроцессорах число РОН равно 32 или больше, поэтому, чтобы увеличить параллелизм выполнений, динамически в процессе эмуляции производится переименование регистров.

Основной объем прикладных пакетов представлен в кодах МП семейств *i*80×86 и МС 680×0. Для перевода этих пакетов на суперскалярные МП Power PC и Alpha разработано большое количество эмуляторов.

Контрольные вопросы

1. В чем состоят особенности программного обеспечения для параллельных ЭВМ?
2. Перечислите механизмы преобразования структур данных в векторных языках.
3. Перечислите основные арифметико-логические операции векторных языков.
4. В чем отличие метода параллелепипедов от метода координат при автоматическом распараллеливании программ?

5. Какие особенности программ препятствуют их векторизации?
6. Охарактеризуйте структуру векторизующего компилятора.
7. Какова концептуальная модель вычислений в многотранспьютерной системе?
8. Опишите средства конфигурирования в Оккаме.
9. Охарактеризуйте основные подходы создания ЯВУ для транспьютеров. Каковы средства описания распределения ресурсов в них?
10. Каковы основные функции системного программного обеспечения в суперскалярных системах?
11. В чем состоят алгоритмы построения ЯПФ?
12. Что такое планирование? Опишите метод списочных расписаний.
13. Какие существуют методы повышения скалярного параллелизма?
14. В чем сущность метода планирования трасс Фишера?

Глава 6

ПАРАЛЛЕЛЬНЫЕ АЛГОРИТМЫ

§ 6.1. Принципы создания параллельных алгоритмов

Для последовательных ЭВМ накоплен большой запас алгоритмов вычислительной математики, оформленных в виде программ на ЯВУ, ассемблере, языках описания алгоритмов.

К алгоритмам предъявлялся ряд требований, например: минимальное время исполнения задачи, минимальный объем памяти, минимизация обращений к внешним устройствам и т. д. Эти требования часто противоречили друг другу и приводили, в конце концов, к появлению ряда вычислительных алгоритмов для решения одной и той же задачи.

В связи с появлением параллельных ЭВМ возникла необходимость в разработке *параллельных алгоритмов*. Следует заметить, что признаки параллельных алгоритмов можно обнаружить и в пакетах программ для последовательных ЭВМ там, где параллелизм присутствует в естественном виде, например, в матричной алгебре, при решении систем уравнений и т. д. В этих случаях возможным является даже эффективное автоматическое преобразование последовательных программ в параллельные с помощью автоматических распараллеливателей или векторизующих компиляторов. Однако чаще необходимо специально разрабатывать и

обосновывать параллельные алгоритмы. Это и является задачей новой и бурно развивающейся отрасли вычислительной математики — *параллельной вычислительной математики*.

Для описания свойств параллельного алгоритма в литературе используется ряд характеристик. Внешне некоторые из них сходны с характеристиками параллельных ЭВМ, описанных в § 1.2, однако отличаются от последних по существу, поскольку не связаны с аппаратурой. Рассмотрим некоторые характеристики алгоритмов.

1. Параллелизм алгоритма, задачи, степень параллелизма. Эти понятия определяют число операций l, которые можно выполнять одновременно, параллельно. Причем при определении l считается, что алгоритм выполняется на идеализированном параллельном компьютере, в котором отсутствуют задержки на коммутацию и конфликты при обращении к памяти, а число процессоров не ограничивает параллелизм. В таком компьютере параллелизм зависит только от внутренних свойств алгоритма. Будем называть такие алгоритмы параллельными математическими алгоритмами в отличие от прикладных параллельных алгоритмов, которые выполняются на параллельных ЭВМ с конечным и часто небольшим числом процессоров. В главе 6 рассматриваются в основном математические алгоритмы.

Если l характеризует часть задачи с основным объемом вычислений, эту величину называют потенциальным параллелизмом и выражают как функцию $f(e)$ или $f(n)$, где e — общее количество арифметико-логических операций в задаче; n — размерность задачи, например, длина вектора, размер стороны квадратной матрицы, число уравнений в системе.

Представление $l = f(e)$ позволяет выбрать один из численных методов среди множества возможных методов для решения данной задачи. Представление $l = f(n)$ позволяет оценить варианты внутри одного численного метода.

Величина l оценивается только по порядку и может принимать, например, такие значения: $0(\log n)$, $0(n)$, $0(n^2)$, где $0(Z)$ обозначает "порядка величины Z".

2. Время исполнения параллельного алгоритма (глубина параллелизма, алгоритмическая сложность). Эти понятия отражают время исполнения параллельного алгоритма q, выраженное в тактах. Величина q информативнее ширины параллелизма l. Однако и

q не несет полной информации о качестве параллельного алгоритма. Если l описывает алгоритм по горизонтали, то q — по вертикали.

Качественно q можно определить как

$$q = e/l \qquad (6.1)$$

Формула (6.1) указывает на средства, позволяющие получить лучший параллельный алгоритм: необходимо уменьшать e и увеличивать l. В некоторых случаях при реорганизации алгоритма эффект можно получить и увеличением e, когда l растет быстрее, чем e: $q' = (e + \Delta e)/(l + \Delta l)$, $q' < q$.

Глубину параллелизма также выражают как функцию $f(e)$ или $f(n)$. Параллельные алгоритмы можно разделить на алгоритмы с фиксированной глубиной параллелизма, алгоритмы с глубиной типа $0(\log n)$, алгоритмы с большей глубиной параллелизма.

В алгоритмах с фиксированной глубиной параллелизма величина q не зависит от объема вычислений или размера структуры данных.

Примерами таких алгоритмов являются простейшие бинарные операции над векторами $C(*) = A(*) \otimes B(*)$, где A, B, C — векторы длиной n; $\otimes$ — символ произвольной арифметико-логической операции. Такая операция будет выполняться за один такт при любом n.

Особенное значение получили алгоритмы с $q = 0(\log n)$. Примеры таких алгоритмов даны в § 6.2. Это алгоритмы каскадного суммирования, циклической редукции и некоторые другие, связанные с распараллеливанием рекурсивных алгоритмов.

Алгоритмы с большей глубиной параллелизма $l = 0(n \log n)$, $l = 0(n^2 \log n)$ и другие малоэффективны, и необходимо искать более быстрые алгоритмы.

3. Наиболее информативной характеристикой параллельного алгоритма является ускорение r, показывающее, во сколько раз применение параллельного алгоритма уменьшает время исполнения задачи по сравнению с последовательным алгоритмом: $r = e_{nc}/q$.

На рис. 6.1 в качественной форме отражено ускорение для задач с различной глубиной параллелизма. Кривая *1* относится к операциям типа $C(*) = A(*) \otimes B(*)$, кривая 2 — к задачам типа каскадных сумм, где общее число операций равно L, а кривая 3 — к умножению матриц одним из методов, для которого $e_{nc} = n$.

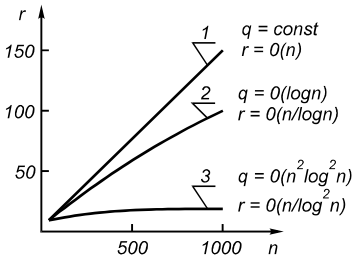


Рис. 6.1. Характер ускорения для параллельных алгоритмов с различной глубиной параллелизма

Для реальных параллельных ЭВМ на основе математических параллельных алгоритмов разрабатываются параллельные прикладные алгоритмы, характеристики которых хуже, чем у математических алгоритмов. Прикладные алгоритмы должны учитывать наличие в реальной ЭВМ оперативной памяти, системы коммутации и ограниченного числа процессоров. Это значительно усложняет создание прикладных алгоритмов, так как при их разработке по сравнению с математическими алгоритмами используются другие критерии. Для математических алгоритмов требовалось уменьшать e и максимально увеличивать l. Для прикладных алгоритмов нет смысла в слишком большом l, так как оно должно быть равно N. В результате если за счет ограничения l удастся уменьшить e, то это повысит качество разрабатываемого прикладного алгоритма.

Параллельные системы предназначены для решения задач большой размерности. Поэтому среди прочих источников погрешностей заметной становится погрешность округления. Известно, что среднеквадратичное значение погрешности округления $\sigma = 0(\beta\sqrt{e})$, где β — значение младшего разряда мантиссы числа. Для чисел с плавающей запятой одинарной точности $\beta = 2^{-24}$. Величина e для ЭВМ с быстродействием 100 млн. оп/с за время

работы около 1 ч достигает значения примерно 10. В результате σ может иметь порядок $0(2^{-24} \cdot 10^6) = 0(2^{-4})$, т. е. погрешность составляет несколько процентов и более, что не всегда допустимо, поэтому для параллельных вычислений часто используется двойная точность.

Далее будут рассмотрены способы построения параллельных алгоритмов для некоторых задач вычислительной математики: суммирование чисел, умножение матриц, решение дифференциальных уравнений в частных производных. Эти задачи характеризуются тем, что они охватывают основные методы вычислительной математики: рекурсию, прямые и итерационные методы решения.

§ 6.2. Некоторые виды параллельных алгоритмов

Рассмотрим отдельные методы распараллеливания рекурсивных алгоритмов, алгоритмов умножения матриц и решения дифференциальных уравнений в частных производных. Эти задачи характерны для практики и предъявляют особые требования к структуре параллельных ЭВМ [2].

Рекурсивные алгоритмы. *Рекурсия* — это последовательность вычислений, при которых значение очередного члена в последовательности зависит от одного или нескольких ранее вычисленных членов последовательности.

Вычисление рекурсии по определению имеет последовательный характер и параллелизм здесь достигается только реорганизацией вычислений. При этом реорганизация вычислений, как правило, приводит к увеличению общего числа операций.

Рассмотрим методику разработки параллельного алгоритма на примере линейной рекурсии первого порядка.

Пусть имеется рекуррентное соотношение

$$x_j = x_{j-1} + d_j,\ j=1,2,\ ...,\ n \qquad (6.2)$$

где $x_j = \sum_{k=1}^{j} d_k$. Этот метод вычислений последовательных сумм можно реализовать с помощью n — сложений в простой программе на Фортране:

```
      X(1) = D(1)
      DO 1 J = 2, N
1     X(J) = X(J - 1) + D(J)
```

На рис. 6.2 для n = 8 представлена зависимость между способом хранения, арифметическими операциями и операциями коммутации. Имеющаяся последовательность перемещается снизу вверх: операции, которые могут вычисляться параллельно, показаны на одном и том же уровне. Ясно, что на каждом временном уровне может быть выполнена только одна операция. Таким образом, алгоритм последовательных сумм имеет n - 1 операцию сложения со степенью параллелизма 1.

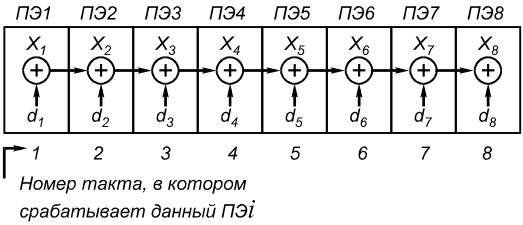


Рис. 6.2. Формирование всех частных сумм для восьми чисел

Чтобы дать количественную оценку схеме коммутации, предположим, что горизонтальная ось на рис. 6.2 указывает относительное размещение данных в памяти последовательной или параллельной ЭВМ. Единичная операция коммутации означает параллельный сдвиг всех элементов массива в соседние блоки памяти последовательной или параллельной ЭВМ. Единичный метод последовательных сумм требует коммутации одной единицы вправо на каждом временном уровне, поэтому метод последовательных сумм приводит к n - 1 операции коммутации со степенью параллелизма 1.

Параллельный метод решения рекурсивных задач изображен на рис. 6.3 для $n = 8$. Набор из n аккумуляторов (регистров, ячеек памяти) сначала загружается данными, которые должны суммироваться. На первом уровне копия содержимого аккумуляторов сдвигается на одну позицию вправо и складывается с несдвинутым содержимым аккумуляторов, чтобы сформировать сумму данных от смежных пар. На следующем уровне процесс повторяется, но со сдвигом на две позиции вправо, тем самым получаем сумму от групп по четыре числа. Поскольку производятся сдвиги, в левую часть заносятся нули. В дальнейшем на l-м временном уровне выполняется сдвиг на 2^l позиций, и на уровне $l = \log_2 n$ аккумуляторы содержат требуемые частные суммы.

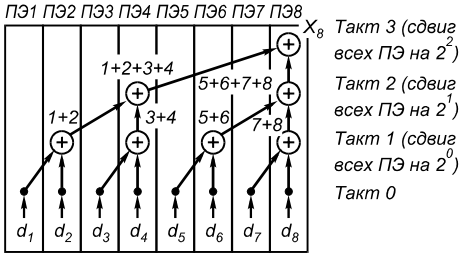


Рис. 6.3. Схема формирования сумм методом параллельных каскадных сумм

Метод параллельных каскадных сумм можно описать на векторном фортранодобном языке:

```
      X = D
      DO 1 L = 1, LOG 2N
1     X = X + SHIFTR (X, 2 ** (L - 1))
```

где X и D — векторы из n элементов; знак "+" — параллельное сложение по n элементам; *SHIFTR* (X, L) — векторная функция, которая помещает копию вектора X, полученную смещением L ячеек памяти вправо (*RIGHT*), во временный вектор *SHIFTR*. Элементы вектора X не затрагиваются. Поэтому метод каскадных частных сумм требует $\log_2 n$ сложений со степенью параллелизма n.

Если мы используем связи типа "к ближнему соседу", то на уровне l потребуется 2^{l-1} единичных операций коммутации, что дает в целом $1 + 2 + 4 + ... + n/2$ или $n - 1$ операций коммутации со степенью параллелизма n.

Для каждой операции метода каскадных частных сумм характерна максимально возможная степень параллелизма n, т. е. имеет место 100%-й параллелизм. Однако общее число скалярных арифметических операций увеличилось с n - 1 для метода последовательных сумм до $n\log_2 n$. Поэтому для последовательных ЭВМ метод каскадных сумм не может использоваться.

Метод каскадных частных сумм значительно упрощается в каждом специальном случае, когда требуется только один результат общей суммы (в нашем случае X_8). На рис. 6.3 отмечены те схемы перемещения и арифметические операции, которые связаны с подсчетом X_8, и они составляют только часть общего объема вычислений. Вычисления, связанные с общей суммой, формируют двоичное дерево (в виде каскада), где число операций и параллелизм делятся пополам на каждом l-м уровне вычисления. Для простоты примем, что n — степень 2. Тогда число операций для метода каскадных сумм составит одно сложение со степенью параллелизма $n\cdot 2^{-l}$ для l = 1, 2, ..., $\log_2 n$. Следовательно, число скалярных сложений в алгоритме: $n/2 + n/4 +...+ 2 + 1 = n - 1$, т. е. метод каскадной общей суммы имеет такое же число скалярных операций, что и метод последовательных сумм. Метод каскадной общей суммы обычно называют методом каскадной суммы.

Рассмотрим параллельный алгоритм для линейной рекурсии более общего характера, чем в (6.2):

$$x_j = a_j x_{j-1} + d_j,\ j=1,2,\ ...,\ n$$

Последовательная программа будет выглядеть так:

```
      X(1) = A(1) * X(0) + D(1)
      DO 1 J = 2, N
1     X(J) = A(J) * X(J - 1) + D(J)
```

Она требует $2n$ арифметических операций со степенью параллелизма 1 и n операций коммутации со степенью параллелизма 1 (рис. 6.4). Для простоты представлены различные типы арифметических операций, хотя можно было бы ограничиться и одним типом, так как, например, для конвейерных ЭВМ выдача результата

сложения или умножения занимает один такт одинаковой длительности.

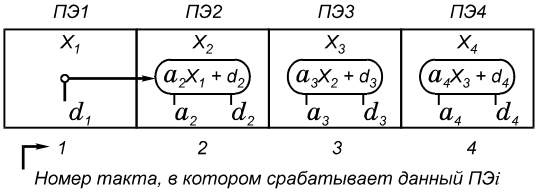


Рис. 6.4. Схема вычисления рекурсии общего вида последовательным методом

Для распараллеливания рекурсий подобного вида используется алгоритм циклической редукции, который позволяет выполнить указанную рекурсию за $3\log_2 n$ операций со степенью параллелизма n.

Умножение матриц. Матричное умножение хорошо иллюстрирует различные способы реорганизации вычислений для согласования с архитектурой ЭВМ, на которой этот алгоритм должен выполняться. Вычисление элементов результирующей матрицы C производится по следующей базовой формуле:

$$C_{i,j} = \sum_{k=1}^{n} A_{i,k} \cdot B_{k,j}\ ,\ 1 \le i;\ j \le n,$$

где i — номер строки; j — номер столбца.

Существует несколько методов умножения матриц.

Если правую часть приведенной формулы обозначить R, то все методы умножения матрицы описываются в виде гнезда из трех вложенных циклов (рассматриваются для простоты квадратные матрицы):

```
      DO 1 I = 1, N                DO 1 J = 1, N
      DO 1 J = 1, N                DO 1 I = 1, N
      DO 1 K = 1, N                DO 1 K = 1, N          (6.3, а, б)
1     C (I, J) = R           1     C (I, J) = R

      DO 1 I = 1, N                DO 1 J = 1, N
      DO 1 K = 1, N                DO 1 K = 1, N
```

```
      DO 1 J = 1, N                DO 1 I = 1, N                (6.3, в, г)
1     C (I, J) = R           1     C (I, J) = R

      DO 1 K = 1, N                DO 1 K = 1, N
      DO 1 I = 1, N                DO 1 J = 1, N
      DO 1 J = 1, N                DO 1 I = 1, N                (6.3, д, е)
1     C (I, J) = R           1     C (I, J) = R
```

В зависимости от расположения индекса *K* методы умножения матриц называются методами внутреннего произведения (6.3 а, б), среднего произведения (6.3 в, г), внешнего произведения (6.3 д, е).

Метод внутреннего произведения для перемножения матриц на последовательных ЭВМ основан на использовании гнезда циклов (6.3 а), где предполагается, что все элементы *C* (*I*, *J*) матрицы перед занесением программы устанавливаются в нуль. Оператор присваивания в программе (6.3 а) формирует внутреннее произведение *i*-й строки матрицы *A* и *j*-го столбца матрицы *B*. Это последовательное вычисление суммы множества чисел, которое описывалось ранее (пусть $d_k = A_{i,k} \cdot B_{k,j}$ в уравнении (6.2), тогда $X_n = C_{i,j}$). Здесь можно производить вычисления последовательно, как в программе (6.3 а), или использовать метод каскадных сумм. В некоторых ЭВМ предусмотрены даже команды внутреннего или скалярного произведения.

Умножению матриц присуща большая степень параллелизма, чем в случае вычисления одиночной суммы. Умножение матриц включает в себя вычисление *n* внутренних произведений. В частности, внутренний цикл по *K* в программе (6.3 а) может быть заменен векторной операцией:

```
      DO 1 I = 1, N
      DO 1 J = 1, N                                             (6.4)
1     C (I, J) = C (I, J) + A (I, *) * B (*, J)
```

где *A* (*I*, ∗) — вектор-строка матрицы *A*; *B* (∗, *J*) — вектор столбец матрицы *B*. Степень параллелизма этой векторной операции равна *n* при умножении. При сложении членов суммы результирующего вектора может применяться метод каскадных сумм.

Замена индексов *I* и *J* согласно (6.3) степени параллелизма не изменяет.

Метод среднего произведения основывается на перестановке порядка циклов *DO* в программе (6.3 а). Если переставить цикл по строкам в самую внутреннюю позицию, то получится программа, которая подсчитывает параллельно внутреннее произведение по всем элементам столбца матрицы *C* (см. (6.3 г)).

Члены в цикле по *I* могут вычисляться параллельно, поэтому цикл (6.3 г) может быть заменен векторным выражением:

```
      DO 1 J = 1, N
      DO 1 K = 1, N                                    (6.5)
1     C (*, J) = C (*, J) + A (*, K) * B (K, J)
```

где *C*(∗, *J*) и *A*(∗, *K*) — векторы, составленные из *J*-го и *K*-го столбцов матриц *C* и *A*. Операция сложения представляет собой параллельное сложение *n* элементов, а операция умножения — умножение скаляра *B*(*K*, *J*) на вектор *A*(∗, *K*). Здесь за каждое выполнение последнего оператора (*K* не меняется) все элементы столбца *C*(*I*, *J*) увеличиваются на одно произведение *A*(*I*, *K*)∗*B*(*K*, *J*). За *K* тактов полностью формируется столбец матрицы *J*.

Степень параллелизма программы для метода среднего произведения равна *n*, а степень параллелизма для исходного метода внутреннего произведения равна единице. Этому способствовало изменение порядка циклов *DO*. Можно было бы переместить цикл по *J* в середину, обеспечив тем самым параллельное вычисление всех внутренних произведений строки. Однако элементы столбца матрицы обычно хранятся в смежных ячейках памяти (в Фортране матрицы хранятся в памяти по столбцам), следовательно, конфликты на уровне блоков памяти уменьшатся, если векторные операции будут производиться над векторами столбцов. Поэтому лучше использовать программу (6.3).

Метод внешнего произведения основывается на вынесении цикла по *K* наружу, т. е. рассматривается вариант (6.3 д, е). Последний оператор можно заменить матричноподобным одиночным оператором, где один член внутреннего произведения подсчитывается параллельно для всех n^2 элементов матрицы *C*. Если всю матрицу обозначим *C* (∗, ∗), то получим

```
      DO 1 K = 1, N
1     C (*, *) = C (*, *) + A (*, K) * B (K, *)
```
(6.6)

где операция умножения представляет собой поэлементное перемножение матрицы $n \times n$, полученной дублированием K-го столбца матрицы A, и матрицы $n \times n$, полученной дублированием K-й строки матрицы B. Операция сложения представляет собой поэлементное сложение в матрице C (*, *).

Дифференциальные уравнения в частных производных. Дифференциальное уравнение в частных производных (ДУЧП) второго порядка в двух размерностях можно представить в общем виде

$$A(x,y)\frac{\partial^2 j}{\partial x^2} + B(x,y)\frac{\partial j}{\partial x} + C(x,y)\frac{\partial^2 j}{\partial y^2} + \\ + D(x,y)\frac{\partial j}{\partial y} + E(x,y)j = r(x,y), \tag{6.7}$$

где коэффициенты A, B, C, D, E — произвольные функции позиции.

Это уравнение включает в себя основные уравнения математической физики и техники (уравнения Гельмгольца, Пуассона, Лапласа, Шредингера и уравнение диффузии) в общепринятых координатных системах (декартовой (x, y), полярной (r, θ), цилиндрической (r, z), аксиально-симметричной сферической (r, θ) и сферической (θ, φ)). Если уравнение (6.7) продифференцируем на $n \times n$ сетке точек с помощью стандартных процедур, то получим множество алгебраических уравнений, каждое из которых имеет отношение к значениям переменных по пяти соответствующим точкам сетки:

$$a_{p,q}\varphi_{p,q-1} + b_{p,q}\varphi_{p,q+1} + c_{p,q}\varphi_{p-1,q} + \\ + d_{p,q}\varphi_{p+1,q} + e_{p,q}\varphi_{p,q} = f_{p,q}, \tag{6.8}$$

где целочисленные индексы p, q = 1, 2, ..., n обозначают точки сетки в направлениях x и y соответственно. Коэффициенты a, b, c, d, e изменяются от точки к точке сетки и относятся к функциям A, B, C, D, E. Для разделения точек сетки можно использовать частную

разностную аппроксимацию. Переменная в правой части $f_{p,q}$ представляет собой линейную комбинацию значений $\rho(x, y)$ в точках сетки вблизи (p, q). В простейшем случае она является значением $\rho(x, y)$ в точке сетки (p, q).

Итерационные процедуры описываются сначала предположительными значениями φ^{p-q} во всех точках сетки и дифференциальным уравнением (6.8). Итерации повторяются, и если успешно, то значения φ стремятся к решению уравнения (6.8) во всех точках сетки.

В самой простой процедуре значения φ во всех точках сетки одновременно настраиваются на значения, которые имели бы по уравнению (6.8), если допустить, что все соседние значения правильные, т.е. φ в каждой точке сетки заменяются новым значением:

$$\varphi^*_{p,q} = \frac{f_{p,q} - a_{p,q}\varphi_{p,q-1} - b_{p,q}\varphi_{p,q+1} - c_{p,q}\varphi_{p-1,q} - d_{p,q}\varphi_{p+1,q}}{e_{p,q}}. \quad (6.9)$$

Поскольку замена должна производиться одновременно, то все значения φ в правой части являются старыми значениями от последней итерации, а отмеченные звездочкой значения в левой части — новыми уточненными значениями.

Описанный метод называется *методом Якоби*, он идеально подходит для исполнения на параллельных ЭВМ. Одновременное выравнивание означает, что для всех точек уравнение (6.9) может выполняться параллельно с максимально возможным уровнем параллелизма n^2. Типичное значение n = 32...256, так что уровень параллелизма изменяется от 1024 до 65 536. Тот факт, что одиночная итерация метода одновременной замены может быть эффективно реализована на параллельных ЭВМ, вовсе не означает, что метод хорош для использования, так как необходимо учитывать число итераций, требуемых для получения удовлетворительной сходимости. Скорость сходимости нельзя найти для общего уравнения (6.8), но известны аналитические результаты для простого случая уравнения Пуассона в квадрате с нулевыми граничными значениями (условие Дирихле). Сказанное соответствует принятию

$a_{p,q} = b_{p,q} = c_{p,q} = d_{p,q} = 1$ и $e_{p,q} = -4$ для всех точек сетки и часто называется модельной задачей.

Обычно для модельной задачи используются прямые, а не итерационные методы решения, но на ней удобно продемонстрировать методику оценки и выбора алгоритма для параллельной ЭВМ.

Известно, что для модельной задачи число итераций t, необходимое для уменьшения погрешности на коэффициент 10^{-p}, равняется $t = pn^2/2$. Таким образом, чтобы по методу Якоби достичь уменьшения ошибки, например, в 10^{-3} раз, на типичной сетке 128×128 требуется 24 000 итераций. Такой медленный характер сходимости делает метод Якоби бесперспективным для параллельных ЭВМ, несмотря на его удобства для организации параллельных вычислений.

Наиболее широко для последовательных ЭВМ используется *метод последовательной повторной релаксации* (SOR). Для этого метода характерно использование взвешенного среднего от старых и новых (отмеченных звездочкой) значений:

$$\varphi_{pq}^{i} = \omega\varphi_{p,q}^{*} + (1-\omega)\varphi_{p,q}^{cm}, \qquad (6.10)$$

где ω — постоянный коэффициент релаксации (обычно лежащий в диапазоне $1 \le \omega \le 2$), который выбирается для увеличения скорости сходимости. Можно показать, что для модельной задачи наилучшая скорость сходимости достигается при

$$\omega = \frac{2}{1+(1-\rho^2)^{1/2}}, \qquad (6.11)$$

где ρ — коэффициент сходимости соответствующей итерации Якоби. Отсюда для модельной задачи $\rho = \cos(\sigma/n)$. Как правило, точки сетки обрабатываются последовательно, точка за точкой и строка за строкой. Улучшение сходимости связано с тем, что новые значения, как только их вычислили, заменяют старые; следовательно, значения в правой части уравнения (6.9), используемые для вычисления φ^{*}, представляют собой смесь старых и новых

значений, и уравнение (6.9) нельзя решить для всех точек в параллель, как это делалось по методу Якоби.

Из сказанного может сложиться мнение, что метод SOR, несмотря на хорошую сходимость, непригоден для реализации на параллельных ЭВМ.

Существуют варианты метода SOR для применения на параллельных ЭВМ. Лучшим из вариантов является четное-нечетное упорядочение по *методу Чебышева*. В соответствии с методом Чебышева точки делятся на две группы; четной или нечетной является сумма $p + q$ (рис. 6.5). Метод возобновляет половину итераций, во время каждой из которых выравнивается только половина точек (выборочно нечетное и четное множество точек) в соответствии с уравнениями (6.9)...(6.11). Кроме того, значение ω изменяется в каждой полуитерации:

$$\omega^0 = 1,$$

$$\omega = 1/(1 - 1/2 \cdot \rho^2),$$

$$\omega^{(k+1/2)} = 1/(1 - 1/4 \cdot \rho^2 \omega^{(k)}),\ \text{k=1/2, 1/4, ..., } \infty,$$

где k — номер итерации.

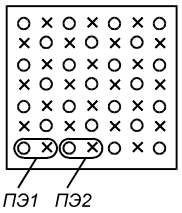


Рис. 6.5. Распределение данных при решении ДУЧП методом Чебышева. Кружками обозначены четные, крестиками — нечетные точки

Чтобы уменьшить погрешность до уровня 10^{-p}, число итераций для метода нечетного-четного упорядочения должно быть равно $t_{SOR} \approx np/3$. На основе этого уравнения можно подсчитать, что уменьшение погрешности до уровня 10^{-3} на 128×128 сетке потребовало бы 128 итераций по сравнению с 24 000 по методу Якоби. Это делает очевидным необходимость использования для параллельных ЭВМ хорошо сходящихся методов.

Метод Чебышева имеет дополнительные преимущества при рассмотрении его реализации на параллельных ЭВМ. Как следует

из уравнения (6.9), обозначенное звездочкой значение в нечетной точке сетки зависит только от старых значений в соответствующих четных точках, которые были подсчитаны во время последней полуитерации. Таким образом, все нечетные точки могут быть выравнены параллельно при уровне параллелизма $n^2/2$ за время одной полуитерации и аналогично все четные точки — параллельно в течение следующей полуитерации. Следовательно, время одной полной итерации пропорционально $n^2/2$.

Описанные методы могут быть расширены и для трех размерностей. Таким образом, при использовании параллельных ЭВМ следует учитывать методы с разной степенью параллелизма n, n^2, n^3.

Далее в сжатом виде приводится ряд известных результатов о параллелизме алгоритмов в следующих областях [23]: алгебра, целочисленная арифметика, ряды и многочлены, комбинаторика, теория графов, вычислительная геометрия, сортировка и поиск.

Все приводимые ниже алгоритмы позволяют получить представление о состоянии параллельной математики.

В качестве модели параллельной ЭВМ, на которой выполняются рассматриваемые алгоритмы, используется широко распространенная модель *параллельной машины с произвольным доступом к памяти* — PRAM (Parallel Random Access Machine). PRAM является эталонной моделью абстрактной параллельной ЭВМ. Она состоит из p идентичных RAM (называемых процессорами), имеющих общую память. Процессоры работают синхронно, и каждый из них имеет возможность переписать в свой сумматор содержимое любой ячейки памяти. Будем считать, что каждый процессор работает по своей программе, хранящейся в локальной памяти.

Машина PRAM удобна тем, что наиболее полно отражает черты реальных параллельных ЭВМ, абстрагируясь от технических ограничений. Имеются различные типы PRAM в зависимости от ответа на вопрос: что происходит, если несколько процессоров обращаются к одной ячейке памяти? В модели EREW PRAM одновременная запись или считывание из одной ячейки сразу в несколько процессоров запрещены. Это слабая модель PRAM. В модели CREW PRAM нескольким процессорам разрешается одно-

временное чтение одной ячейки памяти, но не разрешается одновременная запись в ячейку памяти.

Результаты научных исследований отражены в табл. 6.1. В ней n и m являются параметрами структуры данных.

Разумеется, приведенный перечень не исчерпывает все классы и подклассы уже разработанных алгоритмов. Кроме того, имеется и ряд нерешенных с точки зрения распараллеливания задач, среди которых следует выделить задачу о максимальном паросочетании в графе, о построении дерева поиска в глубину в произвольном графе и задачу о нахождении наибольшего общего делителя двух натуральных чисел. По-видимому, продвижение в указанных направлениях потребует разработки новых методов построения параллельных алгоритмов.

Таблица 6.1.

Характеристики некоторых параллельных алгоритмов

N пп	Наименование алгорита	Тип PRAM	Порядок времени вычислений	Число процессоров
1	2	3	4	5
	Алгебра			
1	вычисление линейной рекурсии первого порядка:	EREW	$0(\log n)$	$0(n/\log n)$
	$x_j - a_j x_{j-1} + b_j$,	EREW	$0(\log k \log n)$	
	j=1, ..., n линейная рекурсия k-го порядка			
2	Решение треугольной системы уравнений, обращение треугольной матрицы	EREW	$0(\log^2 n)$	$0(n^{\omega} / \log n)$
3	Вычисление коэффициентов характеристического уравнения матрицы	EREW	$0(\log^2 n)$	$0(n^4 / \log^2 n)$
4	Решение системы линейных уравнений, обращение матрицы	EREW	$0(\log^2 n)$	$0(n^4 / \log^2 n)$
5	Метод исключения Гаусса	-	$0(\log^2 n)$	$0(n^{\omega+1})$

6	Вычисление ранга матрицы	-	$0(\log^2 n)$	полиномиальное
7	Подобие двух матриц	-	$0(\log^2 n)$	
8	Нахождение LU-разложения симметричной матрицы	EREW	$0(\log^3 n)$	$0(n^4 / \log^2 n)$
9	Отыскание собственных значений Якобиевой матрицы с $\varepsilon > 0$	-	$0(\log^4 n + \log^3 n (\log\log 1/\varepsilon)^2)$	$0(n(\log^4 n + \log^3 n(\log\log$
	Ряды и многочлены			
1	Умножение многочленов $A_1(x)\ldots A_m(x)$, степени n	-	$0(\log(m\cdot n))$	$0(mn\log(mn))$
1	2	3	4	5
2	Вычисление степенных рядов (и членов)	-	$0(\log n)$	
3	Вычисление корней многочлена с погрешностью $\varepsilon > 0$, L — длина записи многочлена	-	$0(\log^3 n(\log n + + \log\log^2(L/\varepsilon))$	
	Комбинаторика			
1	ε — оптимальный рюкзак, n — размерность задачи	-	$0(\log n \log(n/\varepsilon))$	$0(n^3 / \varepsilon^2)$
2	Задача о покрытии с гарантированной оценкой отклонения не более, чем в $(1+\varepsilon)\log d$ раз	-	$0(\log^2 n \log m)$	$0(n)$
3	Нахождение ε — хорошей раскраски в задаче о балансировке множеств	-	$0(\log^3 n)$	полиномиальное число
	Теория графов			
1	Ранжирование списка	-	$0(\log n)$	$0(n/\log n)$
2	Эйлеров путь в дереве	-	$0(\log n)$	$0(n/\log n)$
3	Отыскание основного дерева минимального	CREW	$0(\log^2 n)$	

	дерева минимального веса			
4	Транзитивное замыкание	EREW	$0(\log^2 n)$	$0(n^{\omega}/\log n)$
5	Раскраска вершины в $\Delta + 1$ и Δ цветов	CREW	$0(\log^3 n \log\log n$	$0(n+m)$
6	Дерево поиска в глубину для графа	-	$0(\log^3 n)$	$0(n)$
	Сортировка и поиск			
1	Сортировка	EREW	$0(\log n)$	$0(n)$
2	Слияние для двух массивов размера n и m, $N = m+m$	EREW	$0(\frac{N}{P} + \log N)$	$P = 0(N / \log$

§ 6.3. Особенности выполнения параллельных алгоритмов на ЭВМ различных типов

Рассмотрим особенности выполнения задачи умножения матриц на конвейерных ЭВМ, процессорных матрицах и многопроцессорных ЭВМ типа МКМД. Эта задача выбрана из-за простоты, краткости и наглядности представления.

Конвейерные ЭВМ. В качестве образца конвейерной ЭВМ для экспериментов над параллельными алгоритмами возьмем упрощенный вариант ЭВМ CRAY-1. Напомним ее основные характеристики: такт синхронизации 12,5 нс; память состоит из 16 блоков, обращение к каждому блоку занимает 4 такта; имеется ряд функциональных устройств (векторные АЛУ для чисел с плавающей запятой и длиной конвейера 6 и 7 тактов; временем подключения этих устройств к векторным регистрам для простоты будем пренебрегать) и восемь векторных регистров, каждый по 64 слова. Более подробная информация о машине CRAY-1 представлена в § 2.2.

Приведем примеры стандартного размещения вектора длиной 16 элементов и матрицы размерностью 16×16 (рис. 6.6). Стандартным для расположения матриц в Фортране является размещение по столбцам. Если размер матрицы кратен числу блоков памяти, то стандартное размещение матрицы приводит к тому, что элементы

столбцов расположены в разных блоках памяти и их можно считывать по очереди с максимальной скоростью: один элемент за один такт. Но каждая строка оказывается полностью размещенной в одном блоке памяти, поэтому последовательные элементы строк считываются с минимальной скоростью: один элемент за четыре такта (см. рис. 6.6, а).

Определенные проблемы существуют и в размещении векторов (см. рис. 6.6, б). Последовательные элементы вектора a_1, a_2, a_3, ... и a_1, a_5, a_9, ... с кратностью 4 считываются с максимальной скоростью элементы a_1, a_9, a_{17}, ... с кратностью 8 считываются с половинной скоростью, так как за время цикла памяти (четыре такта) в каждый блок поступает два запроса; элементы a_1, a_{17}, ... с кратностью 16, расположенные в одном блоке, считываются со скоростью в четыре раза ниже максимальной. Эти ситуации могут встречаться при использовании методов каскадных сумм или циклической редукции.

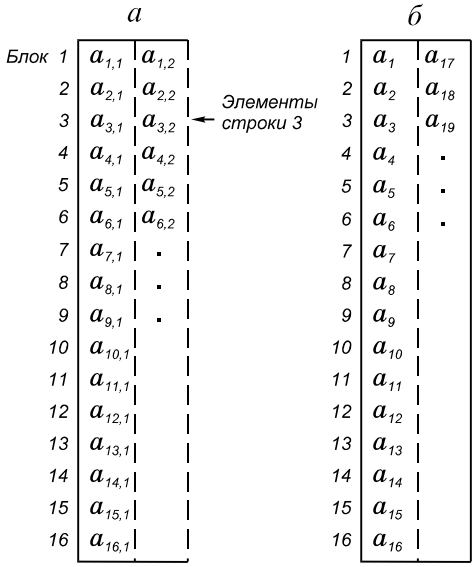


Рис. 6.6. Размещение матриц (*а*) и векторов (*б*) в многоблочной памяти конвейерной ЭВМ

В конвейерной ЭВМ выделяют три вида производительности:

1) скалярная производительность достигается только при использовании скалярных команд и регистров;

2) векторная производительность достигается при использовании программ с векторными командами, скорость выполнения которых ограничивается скоростью обмена с памятью;

3) сверхвекторная производительность достигается при использовании программы с векторными командами, скорость выполнения которых ограничивается готовностью векторных регистров или функциональных устройств.

В CRAY-1 скорости выполнения программ для скалярных и векторных команд заметно отличаются и равны 12 и 153 Мфлоп/с соответственно.

При умножении матриц нас будут интересовать только векторная и сверхвекторная производительности, причем основными средствами ускорения вычислений являются:

- обеспечение максимальной скорости обращения к памяти;
- уменьшение объема обращений к памяти и соответственно увеличение объема работы с векторными регистрами;
- увеличение длины векторов, чтобы уменьшить влияние времени “разгона” (определяется длиной конвейера функциональных устройств);
- максимальное использование зацепления операций.

Для простоты дальнейшего анализа рассмотрим матрицы размерностью 64×64. При использовании метода внутреннего произведения (см. (6.4)) необходимо выполнить следующий алгоритм для вычисления одного элемента матрицы C (рис. 6.7, а).

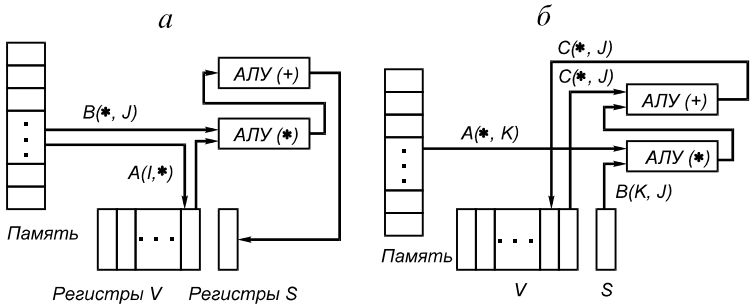


Рис. 6.7. Схема соединений конвейерной ЭВМ для выполнения операций умножения матриц:
а — метод внутреннего произведения; б — метод среднего произведения

Из памяти в векторный регистр выбирается строка $A(I, *)$ матрицы A. При стандартном размещении матриц все элементы строки оказываются в одном блоке памяти, поэтому на выборку строки затрачивается время (в тактах) $t_1=l_1+4n$, где n — размер стороны матрицы, а l_1 — время "разгона" (длина конвейера) памяти (для CRAY-1 $l_1 = 11$). Затем производится поэлементное умножение векторов в АЛУ (∗). Суммирование получаемых произведений в АЛУ (+) выполняется в зацеплении с предыдущей операцией. Результат $C(I, J)$ заносится в один из скалярных регистров S. Это потребует времени $t_2=l_2+l_3+n$, где $l_2 = 7$ и $l_3 = 6$ — время "разгона" конвейерных АЛУ сложения и умножения соответственно. Тогда время вычисления всей матрицы методом внутреннего произведения будет:

$$t_{вн} = (t_1 + t_2)n^2 = (l_1 + l_2 + l_3 + 5n)n^2 = 5{,}3n^3, \qquad (6.12)$$

если предположить, что $l_1 + l_2 + l_3 = 22 \approx 0{,}3$n.

При использовании описанного метода большое замедление вызывает выборка строки матрицы A. Если применим специальное (нестандартное) размещение матрицы A таким образом, чтобы смежные элементы строки были расположены в соседних блоках памяти, то обеспечим считывание каждого элемента строки за один такт, т. е. с максимальной скоростью. Тогда

$$t_{âí} = 2{,}3n^3. \qquad (6.13)$$

Но при этом требуется изменение стандартных трансляторов.

Более перспективными являются методы среднего произведения. Рассмотрим реализацию программы (6.5). Соответствующая схема для вычисления одного столбца матрицы приведена на рис. 6.7, б. При использовании метода среднего произведения столбец $C(*, J)$ постоянно располагается в регистрах V, а из блоков памяти выбираются только столбцы $A(*, K)$. Поскольку при стандартном расположении матриц элементы столбца располагаются в различных блоках памяти, то элементы $A(*, K)$ будут выбираться с максимальной скоростью. При вычислении элементов столбца

матрицы C методом среднего произведения совмещают три операции: считывание $A(*, K)$ из блоков памяти; умножение этого вектора на константу $B(K, J)$, получаемую из скалярного регистра S; сложение столбца $C(*, J)$ с результирующим вектором из АЛУ $(*)$. Поэтому среднее время вычисления всей матрицы C

$$t_{ср} = n^2(l_1 + l_2 + l_3 + n) = 1,3n^3. \qquad (6.14)$$

При определении времени умножения матриц опущены различного рода вспомогательные операции. В частности, в последнем случае не учтены операции считывания из памяти и запись в память столбцов матрицы C. Таким образом, полученные оценки времени умножения следует считать приблизительными.

Оценим быстродействие, получаемое по формуле (6.14). При вычислении произведения матриц выполняется n^3 операций умножения и n^3 операций сложения. Для конвейерных ЭВМ длительности тактов сложения и умножения равны, поэтому будем считать, что выполняется $2n^3$ операций и, следовательно, на одну арифметико-логическую операцию затрачивается

$$\frac{t_{ср}}{2n^3} = \frac{1,3}{2} = 0,65 \text{ такта.}$$

Для CRAY-1 при длительности такта 12,5 нс это соответствует быстродействию

$$V \frac{10^9}{0,65 \cdot 12,5} = 120 \text{ Мфлоп/с.}$$

Таким образом, метод среднего произведения обеспечивает сверхвекторную производительность и считается, будучи запрограммированным на ассемблере, наилучшим методом для ЭВМ типа CRAY-1. Его преимущества обеспечиваются большой глубиной зацепления и тем, что один из векторов (столбец матрицы C) постоянно расположен в векторных регистрах.

Метод внутреннего произведения обеспечивает векторную (см. (6.12)) и промежуточную между векторной и сверхвекторной (см. (6.13)) производительности.

Метод внешнего произведения не имеет преимуществ перед методом среднего произведения. Более того, возможность работы с векторными регистрами и глубокое зацепление делают метод среднего произведения предпочтительным.

Процессорные матрицы. При выборе наилучшего алгоритма для ПМ необходимо учитывать следующие структурные отличия процессорных матриц от конвейерных ЭВМ.

1. Быстродействие в ПМ достигается за счет числа процессоров N, а не за счет глубины конвейера.

2. Основными факторами, увеличивающими время выполнения алгоритма, являются конфликты в памяти и затраты на коммутацию, поэтому размещение данных в памяти является одним из наиболее существенных моментов для процессорных матриц.

Примем для дальнейших расчетов следующее время выполнения отдельных операций в ПЭ: сложение двух чисел — 1 такт, выборка чисел из памяти — 1 такт; умножение двух чисел — 2 такта; операция коммутации для N процессоров ($N \le 2^5$) — 1 такт; операция типа SUM — $2\log_2 N$ тактов.

Далее будет рассмотрена операция умножения матриц для следующих вариантов в предположении, что n — размер стороны матрицы данных; N — общее число процессоров в ПМ:

1) $N = n$. Этот вариант позволяет рассмотреть некоторые специальные методы размещения данных;

2) $N = n^2$, $N = n^3$. В большинстве случаев данные варианты имеют теоретический интерес, так как при $n = 100$ (это реальное число) требуется 10^4 или 10^6 процессоров, что пока еще нельзя осуществить;

3) $N < n$. Этот вариант является наиболее интересным с точки зрения практики.

Очевидно, что для любого N следует выбирать методы умножения матриц с наибольшим потенциальным параллелизмом.

В а р и а н т $N = n$. Для этого варианта следует применять алгоритмы с уровнем параллелизма 0 (n) и выше. Рассмотрим метод внутренних произведений (см. программу (6.4)). В основе метода лежит параллельная выборка строк матрицы A и столбцов матрицы B. Тогда алгоритм вычисления одного элемента $C(I, J)$ может быть таким:

1) параллельное считывание из памяти в регистры всех ПЭ*i*-й строки матрицы *A* — 1 такт;

2) считывание *J*-го столбца матрицы *B* в регистры ПЭ — 1 такт;

3) поэлементное перемножение выбранных строки и столбца —2 такта;

4) суммирование *n* частных произведений — $2\log_2 n$ тактов.

Следовательно, вычисление всех элементов матрицы *C* займет время (в тактах)

$$t_{вн} = n^2(4 + \log_2 n).$$

При использовании метода среднего произведения (см. (6.5)) матрицы *A* и *C* располагаются горизонтально по столбцам, а матрица *B* — в памяти ЦУУ. Поэтому алгоритм вычисления одного столбца матрицы *C* будет следующим:

1) запись очередного элемента B (K, J) во все ПЭ — 1 такт;

2) считывание из памяти в ПЭ *K*-го столбца матрицы *A* — 1 такт;

3) умножение *A*(∗, *K*) *B*(*K*, *J*) — 2 такта;

4) суммирование полученных частных произведений с соответствующими частными суммами элементов *J*-го столбца матрицы *C*, который постоянно располагается в регистрах ПЭ — 1 такт.

Таким образом, одна итерация вычисления *C*(∗, *J*) занимает 5 тактов; полное вычисление столбца *C*(∗, *J*) займет 5 тактов, а вычисление всей матрицы *C*

$$t_{ср} = 5n^2 \text{ тактов.} \qquad (6.15)$$

Очевидно, что $t_{ср} < t_{вн}$ для всех $n \geq 2$. Преимущество метода средних произведений — в отсутствии итерации вычисления каскадной суммы.

Вариант $N = n^2$. Для этого варианта следует применять алгоритмы с уровнем параллелизма $0(n^2)$ и выше. Для достижения такого параллелизма могут быть использованы различные варианты метода внешнего произведения (см. (6.6)) и комбинации метода среднего произведения. Рассмотрим метод внешнего произведения. Прямая реализация данного метода предполагает конфигура-

цию ПП в виде матрицы процессоров размерностью $n \times n$, причем в каждом ПЭi, j вычисляется один элемент матрицы $C(I, J)$. Для этого в памяти каждого ПЭi, j должны находится все элементы i-й строки матрицы A и j-го столбца матрицы B. Тогда вычисление всех элементов матрицы C потребует n итераций. За время одной итерации в каждом ПЭi, j выполняются следующие действия:

1) считывание из памяти элемента $A(I, K)$ — 1 такт;

2) считывание из памяти элемента $B(K, J)$ — 1 такт;

3) умножение $A(I, K) * B(K, J)$ в ПЭ — 2 такта;

4) подсуммирование полученного произведения к ранее накопленной сумме из K - 1 произведения — 1 такт.

Следовательно, вычисление матрицы займет время $t_{вн\ ш} = 5n$ тактов.

К сожалению, этот метод требует слишком большого объема памяти процессорного поля. Если суммарное число элементов в матрицах A и B равно $2n^2$, то вследствие многократного дублирования строк и столбцов матриц A и B в различных ПЭ суммарное число элементов возрастет до величины $2n^3$. Данное условие требует для матрицы размерностью 100×100 памяти 16 Мбайт, которая не всегда может быть доступна. Поэтому используются методы со степенью резервирования $\beta < n$, как в описанном выше варианте. Тогда $t_{вн\ ш} = f(\beta)$.

Для уменьшения резервирования данных часто используют модификации метода среднего произведения.

Вариант $N = n^3$. Этот вариант имеет больше теоретическое, чем прикладное, значение. В частности, он может использоваться при умножении матриц размерностью 16×16 на ЭВМ ICL DAP (размерность ПП 64×64). В этом случае все размещения матриц в ПП аналогичны размещению этих матриц в описанном выше методе внешнего произведения, но поскольку вместо одной матрицы процессоров будет 16 слоев ПМ, то все n^3 операции умножения могут быть выполнены одновременно, а затем все частные произведения суммируются за $0(\log_2 n)$ тактов. Следовательно, общее время выполнения операции умножения матриц составит $t = 0(1 + \log_2 n)$. Этот метод требует такого же избыточного резер-

вирования данных, как и метод среднего произведения при $N = n^2$.

В а р и а н т $N < n$. Следует сказать, что это наиболее реалистичная ситуация, так как обычно N колеблется от 8...16 до 100, в то время как размер матрицы n может колебаться от 100 до 1000.

Оценим ситуацию, когда n кратно числу процессоров N. Тогда можно использовать *метод клеточного умножения матриц*. Рассмотрим для простоты умножение матриц $A \times B = C$ размерностью 4×4:

$$\begin{array}{cccc} a_{11} & a_{12} & \mathrm{M}a_{13} & a_{14} \\ a_{21} & a_{22} & \mathrm{M}a_{23} & a_{24} \\ \ldots & \ldots & \mathrm{M}\ldots & \ldots \\ a_{31} & a_{32} & \mathrm{M}a_{33} & a_{34} \\ a_{41} & a_{42} & \mathrm{M}a_{43} & a_{44} \end{array} \times \begin{array}{cccc} b_{11} & b_{12} & \mathrm{M}b_{13} & b_{14} \\ b_{21} & b_{22} & \mathrm{M}b_{23} & b_{24} \\ \ldots & \ldots & \mathrm{M}\ldots & \ldots \\ b_{31} & b_{32} & \mathrm{M}b_{33} & b_{34} \\ b_{41} & b_{42} & \mathrm{M}b_{43} & b_{44} \end{array} = \begin{array}{cccc} c_{11} & c_{12} & \mathrm{M}c_{13} & c_{14} \\ c_{21} & c_{22} & \mathrm{M}c_{23} & c_{24} \\ \ldots & \ldots & \mathrm{M}\ldots & \ldots \\ c_{31} & c_{32} & \mathrm{M}c_{33} & c_{34} \\ c_{41} & c_{42} & \mathrm{M}c_{43} & c_{44} \end{array} \quad (6.16)$$

Вычисление любого элемента матрицы C производится по известной формуле

$$C(I, J) = C(I, J) + A(I, K) * B(K, J),\ K = 1, 2, 3, 4$$

В частности, для c_{11} получаем

$$c_{11} = a_{11}b_{11} + a_{12}b_{21} + a_{13}b_{31} + a_{14}b_{41} \qquad (6.17)$$

Разобьем исходные матрицы на клетки (как показано в (6.16) штриховыми линиями), которые обозначим $a_{i,j}$, где i, j — индексы, указывающие положение клеток в соответствии с обычными обозначениями для матриц. Тогда выражение (6.16) для матриц A, B, C заменяется следующим выражением для матриц A', B', C':

$$\begin{vmatrix} a'_{11} & a'_{12} \\ a'_{21} & a'_{22} \end{vmatrix} \times \begin{vmatrix} b'_{11} & b'_{12} \\ b'_{21} & b'_{22} \end{vmatrix} = \begin{vmatrix} c'_{11} & c'_{12} \\ c'_{21} & c'_{22} \end{vmatrix} \qquad (6.18)$$

Для (6.18) по обычным правилам выполнения матричных операций имеем:

$$\begin{aligned} c'_{11} &= a'_{11}b'_{11} + a'_{12}b'_{21},\ \ c'_{12} = a'_{11}b'_{12} + a'_{12}b'_{22}, \\ c'_{21} &= a'_{21}b'_{11} + a'_{22}b'_{21},\ \ c'_{22} = a'_{21}b'_{12} + a'_{22}b'_{22}. \end{aligned} \qquad (6.19)$$

Выражения (6.19) позволяют вычислить все элементы исходной матрицы C на основе матриц меньшего размера, в частности для c_{11} можно записать:

$$c'_{11} = \begin{vmatrix} c_{11} & c_{12} \\ c_{21} & c_{22} \end{vmatrix} = \begin{vmatrix} a_{11} & a_{12} \\ a_{21} & a_{22} \end{vmatrix} \times \begin{vmatrix} b_{11} & b_{12} \\ b_{21} & b_{22} \end{vmatrix} + \begin{vmatrix} a_{13} & a_{14} \\ a_{23} & a_{24} \end{vmatrix} \times \begin{vmatrix} b_{31} & b_{32} \\ b_{41} & b_{42} \end{vmatrix} \quad (6.20)$$

откуда, например, по обычным правилам для умножения матриц следует:

$$c_{11} = a_{11}b_{11} + a_{12}b_{21} + a_{13}b_{31} + a_{14}b_{41} \quad (6.21)$$

Выражения (6.17) и (6.21) идентичны, что подтверждает правильность приведенных выкладок.

Таким образом, при использовании метода клеток для умножения матриц необходимо в общем случае выполнить перечисленные ниже действия:

1. Исходные матрицы большого размера, когда сторона матрицы $n>>N$, разбиваются на клетки размерностью $N\times N$. Число таких клеток в каждой матрице будет $(n/N)^2$. Эти клетки размещаются в памяти и могут в дальнейшем обрабатываться с параллелизмом N.

2. Для получения разбиения составляются системы выражений типа (6.19). Каждое выражение будет содержать n/N слагаемых. Алгоритмы вычислений по системе выражений типа (6.19) закладываются в программу пользователя или автоматически генерируются транслятором с языка ЯВУ.

3. В процессе выполнения программы в соответствии с выражениями типа (6.20) из памяти выбираются и обрабатываются с параллелизмом N клетки, т. е. матрицы значительно меньшего размера, чем исходные, что и обеспечивает эффективность обработки. При этом умножение матриц в выражении (6.20) может производиться любым подходящим методом, например, методом среднего произведения.

Для некоторых методов вычислений метод клеточных матриц не может быть использован, в подобном случае применяется размещение данного вектора в памяти процессорного поля по слоям (см. § 2.4).

Обобщая полученные результаты, сравним время умножения матриц для конвейерных ЭВМ $t_к$ и матриц процессоров $t_м$. Предположим, что для каждого типа ЭВМ используются лучшие методы (см. (6.14) и (6.15) соответственно) и время одного такта равно 10 нс для конвейерной ЭВМ и 100 нс для ПМ. Вопрос о требуемых объемах оборудования при этом не рассматривается. Соотношение времени умножения матриц

$$r = \frac{t_м}{t_к} = \frac{5 \cdot n^2 \cdot 100}{1{,}3 \cdot n^3 \cdot 10} \approx \frac{39}{n}. \qquad (6.22)$$

Из формулы (6.22) следует, что при $n \geq 39$ процессорные матрицы предпочтительнее. Необходимо еще раз подчеркнуть, что подобные расчеты носят качественный характер.

Многопроцессорные ЭВМ. В многопроцессорных ЭВМ каждый процессор — полноценный процессор последовательного типа и выполняет независимую ветвь параллельной программы. Ветвь является обычной последовательной программой, как правило, не содержащей в себе операторов разделения на другие ветви или операторов векторной обработки. Однако при выполнении параллельной программы ветви обмениваются информацией. Это и есть главный источник увеличения времени выполнения параллельной программы.

Число процессоров в многопроцессорной системе обычно невелико: $N = 2...10$, но каждый процессор обладает большой вычислительной мощностью и содержит конвейер команд или арифметический конвейер.

При разбиении задачи на ветви для исполнения на многопроцессорной ЭВМ (декомпозиция задачи) соблюдаются определенные принципы.

1. Задача должна разбиваться по возможности на наиболее крупные и редко взаимодействующие блоки. При таком подходе схема используемого для декомпозиции последовательного алгоритма рассматривается как иерархическая структура, высший уровень которой — крупные блоки, следующий уровень — подблоки крупных блоков и т. п. Процесс распараллеливания начинается с самого высокого уровня. Переход на следующий уровень проис-

ходит только в том случае, если не удалось достичь нужной степени распараллеливания на данном уровне.

В качестве блоков одного уровня обычно рассматривают циклические структуры одинаковой вложенности (пример крупноблочно-иерархического представления алгоритма дан на рис. 6.8).

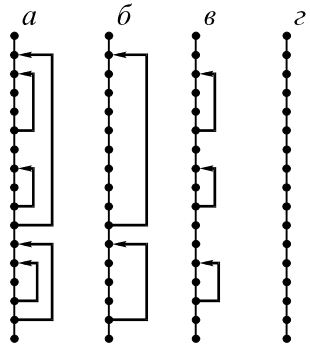


Рис. 6.8. Иерархический принцип построения алгоритма:
а — схема алгоритма; *б* — структурная единица первого уровня иерархии; *в, г* — структурные единицы второго уровня иерархии

Аналогично используется иерархия в структуре данных, если параллельный алгоритм строится на основе распределения по ветвям обрабатываемых данных, а не на основе операторов последовательного алгоритма.

2. Должно быть обеспечено однородное распределение массивов по ветвям параллельного алгоритма, поскольку при этом уменьшается время, затрачиваемое на взаимодействие ветвей. При таком распределении имеет место:

а) равенство объемов распределяемых частей массивов;

б) соответствие нумерации распределяемых частей массивов нумерации ветвей;

в) разделение массивов на l частей параллельными линиями или параллельными плоскостями (обычно l равно числу процессоров);

г) дублирование массивов или частей одинаковым образом.

Если рассмотреть двумерный массив, то к нему применимы следующие основные способы однородного распределения: горизонтальные полосы (ГП-распределение), циклические горизонтальные полосы, вертикальные полосы (ВП-распределение), скошенные полосы, горизонтальные и вертикальные полосы с дублированием, горизонтальные полосы с частичным дублированием и т. п. Некоторые из этих распределений показаны на рис. 6.9.

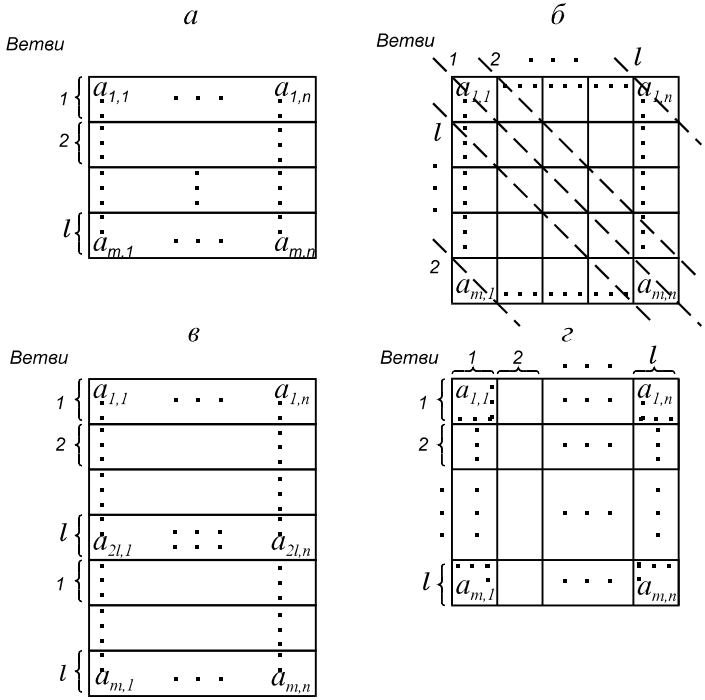


Рис. 6.9. Способы распределения двумерного массива:
а — горизонтальные полосы; *б* — скошенные полосы; *в* — циклические горизонтальные полосы; *г* — вертикальные и горизонтальные полосы с дублированием

Практикой установлено, что для многопроцессорных систем используются следующие пять основных схем обмена между ветвями [1]:

- трансляционный обмен (ТО) — “один — всем”;
- трансляционно-циклический обмен (ТЦО) — “каждый — всем”;
- коллекционный обмен (КО) — “все — одному”;
- парный стационарный обмен (ПСО) — “каждый — соседу”;
- парный нестационарный обмен (ПНО) — “каждый — любому”.

Продолжим рассмотрение задачи умножения матриц, но для многопроцессорных систем. Исходя из принципа крупноблочного разбиения, можно предположить, что наиболее подходящим для многопроцессорных систем будет метод внутреннего произведения с разбиением по внешним индексам. Разберем следующие варианты умножения матриц размерностью $n \times n$:

```
      DO 1 J = 1, n
      DO 1 I = 1, n
      DO 1 K = 1, n
1     C (I, J) = R
```

где R соответствует $\sum_{k=1}^{n} a_{i,k} b_{k,j}$. На первом уровне анализа имеется цикл по j. Его различные итерации независимы. Распределение индекса j по ветвям можно провести различным способом, в частности массивы B и C можно распределить ВП-способом. Дублирование матрицы A в каждой ветви обеспечивает ей возможность параллельных вычислений всех элементов матрицы C без обмена с другими ветвями, а ГП-распределение массива A и ведение обмена между ветвями позволяет избежать расхода памяти на дублирование. Схема вычислений приобретает следующий вид (для одного процессора):

```
      DO 1 J = 1, n/1
      DO 1 I = 1, n
      DO 2 K = 1, n
2     C(I, J) = R
1     O(I)
```

Здесь O(I) — оператор обмена, организующий рассылку строки i матрицы A во все ветви. Одна рассылка обеспечивает параллельное вычисление l элементов матрицы C. Долю обменов можно уменьшить, если поменять местами циклы по J и I:

```
      DO 1 I = 1, n
      DO 2 J = 1, n/1
      DO 2 K = 1,n
2     C(I, J) = R
1     O(I)
```

При такой схеме одна рассылка строки обеспечивает вычисление *n* элементов матрицы *C*. Тогда рассылка осуществляется по трансляционной схеме обмена (ТО).

Максимальное число ветвей, а значит, и параллелизм равны числу повторений тела цикла. Большая степень параллелизма возможна при переходе к следующему уровню, блоки которого определяются структурой отдельного "витка". Для матриц — это переход к среднему, а затем и к внутреннему циклу.

Задача умножения матриц носит явно выраженный векторный характер и для ее решения более предпочтительным являются векторные ЭВМ (конвейерные ЭВМ и ПМ). Для многопроцессорных систем в настоящем параграфе задача умножения матриц использовалась только из-за простоты оценки эффективности различных вариантов вычислений.

Многопроцессорные системы более эффективны для задач с неоднородными по составу операторов ветвями. Эта неоднородность отражает природу какого-либо явления и может быть известна заранее. Тогда в параллельной программе для запуска неидентичных ветвей используется оператор FORK. Неоднородность может проявляться и в процессе счета, в таком случае она отображается в программе с помощью операторв IF. Подобная ситуация встречается, например, при имитационном моделировании цифровых схем.

§ 6.4. Методы параллельной обработки нечисловой информации

Обработка нечисловой информации приобретает все большее значение в связи с созданием систем автоматизации и интеллектуализации труда в производственной и научной сферах.

В этом параграфе будут рассмотрены два примера:

1) построение распределенной базы данных реляционного типа на ПМ;

2) реализация некоторых алгоритмов автоматического программирования для машин с управлением от потока данных.

Распределенные базы данных. Основной операцией в БД является *транзакция* — получение ответа на запрос, представленный в определенной форме. В современных БД число запросов дости-

гает нескольких сотен в секунду, и на каждый запрос требуется выполнить около миллиона арифметико-логических операций. Следовательно, ЭВМ для обработки БД должны иметь быстродействие в десятки и сотни миллионов операций в секунду, а объем памяти может достигать нескольких гигабайт.

Структура современных ЭВМ для обработки БД, как правило, строится по принципу процессорной матрицы. Рассмотрим структуру, функционирование и программирование одной из таких ЭВМ, сходной по системе команд с известным процессором для реляционных баз данных *RAP* [1].

Существуют сетевые, иерархические и реляционные БД. Последние наиболее перспективны. Ознакомимся с некоторыми понятиями *реляционной алгебры*.

Основным понятием реляционной алгебры является *отношение* (рис. 6.10). Отношение содержит ряд столбцов, называемых доменами. Каждый домен является набором некоторых величин и имеет свое имя. Например, на рис. 6.10 домен Γ_1 может означать номера учебных групп, домен C_2 — номера зачетных книжек студентов, чьи фамилии находятся в домене A_3.

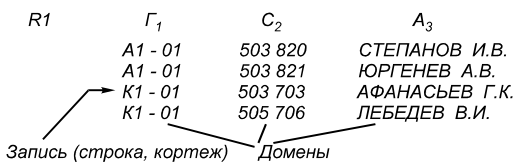


Рис. 6.10. Структура отношения:
$R1$ — имя отношения; Γ_1, C_2, A_3 — имена доменов

База данных есть совокупность отношений, связанных общими доменами и изменяющихся во времени. Отношения имеют различную размерность и обрабатываются для извлечения неявно заданной информации. Одинаковых строк в отношении не может быть.

Система обработки данных в БД называется системой управления базой данных (СУБД).

Над отношениями могут выполняться различные операции.

Пусть имеются два отношения *R* и *S*:

$R\,(A, B)$	$S\,(C, D)$
$a_1 b_1$	$c_1 d_1$
$a_2 b_2$	$a_1 b_1$

Тогда возможны следующие основные операции над отношениями R и S.

1. Объединение

$R \cup S(M, N)$
$a_1 b_1$
$a_2 b_2$
$c_1 d_1$

2. Пересечение

$R \cap S(M, N)$
$a_1 b_1$

3. Разность

$R / S(M,N)$
$a_2 b_2$

4. Произведение

$R(A, B) \otimes S(C) = R \otimes S(A, B, C)$		
$a_1 b_1$	c_1	$a_1 b_1 c_1$
$a_2 b_2$	c_2	$a_1 b_1 c_2$
		$a_2 b_2 c_1$
		$a_2 b_2 c_2$

5. Проекция (предназначена для изменения числа столбцов в отношении, например, для исключения чего-либо). Пусть дано отношение

$R(D_1,$	D_2	$D_3)$
a	2	f
b	1	g
c	3	f
d	3	g
e	2	f

Тогда пять его проекций будут таковы:

$R[1]M_1$	$R[2]M_1$	$R[3]M_1$	$R[3,2]M_1,M_2$		$R[3,2,2]M_1,M_2,M_3$		
a	2	f	f	2	f	2	2
b	1	g	g	1	g	1	1
c	3		f	3	f	3	3
d			g	3	g	3	3
e							

6. Соединение (по условиям =, ≠, <, ≤, >, ≥). Пусть даны

$R(A, B, C)$			$S(D, E)$	
a	2	1	2	f
b	1	2	3	g
c	2	5	4	f
c	2	3		

Тогда возможны следующие результаты:

$R\ [C = D]\ S(A, B, C, D, E)$

b	1	2	2	f
c	5	3	3	g

$R\ [B \neq C]\ S(A, B, C)$

a	2	1
b	1	2
c	2	5
c	5	3

$R\ [C < D]\ S(A, B, C, D, E)$

a	2	1	2	f
a	2	1	3	g
a	2	1	4	f
b	1	2	4	g
b	1	2	4	f
c	5	3	4	f

$R\ [B > C]\ S(A, B, C)$

a	2	1
c	5	3

Эти операции реляционной алгебры в том или ином виде входят в системы команд различных реляционных процессоров баз данных.

Структура параллельного реляционного процессора (ПРП), аналогичного RAP, совпадает с общей структурой процессорной матрицы.

В ответ на запрос на специальном языке запросов ПРП выполняет следующие действия: преобразует запрос во внутреннюю форму; генерирует программу поиска ответа на запрос; ищет ответ; преобразует ответ во внешнюю форму; выдает ответ. Причем собственно ПРП ищет только ответ, остальные действия осущест-

вляет базовая машина, формирующая программу поиска, которая реализуется ЦУУ и всеми ПЭ. Коммутатор необходим для сравнения результатов поиска в отдельных ПЭ.

В качестве ОП обычно используется сдвиговая память на приборах с зарядовой связью. К памяти возможен только последовательный доступ. Но с развитием методов поиска описаний отношений будет применяться и память с произвольным доступом, тогда роль коммутаторов возрастет.

Все ПЭ выполняют одну и ту же команду поиска, при этом каждый ПЭ*i* хранит имя отношения. По мере последовательного считывания записей из ОП*i* запись анализируется, и, если необходимо, заменяется новой, которая записывается в память. В каждом АЛУ имеется местная память для хранения доменов на время их обработки.

Отношение располагается в ОП*i* в форме последовательности кортежей (рис. 6.11). Поле *DF* используется для размещения флага удаления. Если *DF* = 1, то данный кортеж будет стерт, а программа “сбора мусора” учтет это место в памяти как свободное.

Кортеж считается *T*-маркированным, если в разрядах *ABCD* содержится определенная совокупность нулей и единиц, называемая *T*-образцом. Поле каждой величины имеет разрядность 8, 16 или 32, причем первые два бита задают его длину. Кортеж содержит не более 255 доменов. Последний кортеж в первом домене содержит комбинацию 11, которая задает конец записи. При наращивании записи эта комбинация перемещается к ее концу, при сокращении — к началу.

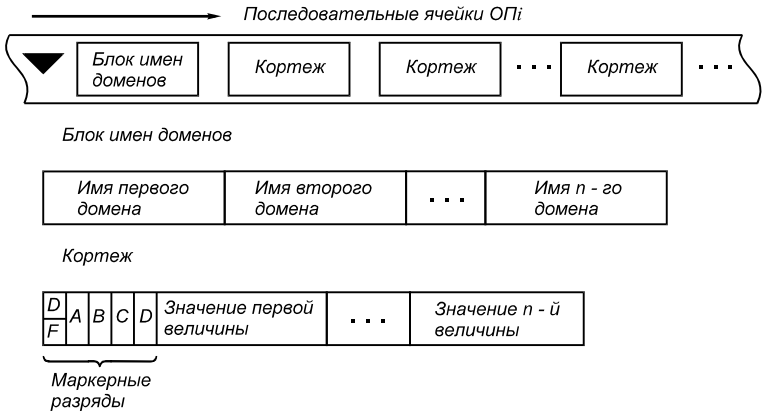


Рис. 6.11. Структура хранения отношения в памяти

Программа ПРП рассматривается как канальная программа БМ, только семантика этих команд другая. Команда имеет формат:

метка><КОП><маркеры>[<объект>:<условие>] [<параметр>]

Поле <метка> не обязательно. Оно используется только при переходах в программе. В поле < маркеры > указываются разряды, которые нужно установить, например, *MARK* (*A*) означает, что нужно разряд *A* установить в 1 во всех кортежах, удовлетворяющих заданному условию. Если задана операция *RESET* (*AB*), то надо сбросить разряды *A*, *B* во всех кортежах при выполнении условия.

Поле <объект> имеет вид: <Rn>(<D1>, <D2>, ..., <Dn>), где *Rn* — имя отношения; *Di* — имя домена отношения.

Логическое выражение может иметь форму: вырожденную; конъюнкции элементарных условий; дизъюнкции элементарных условий. Элементарное условие можно задавать в таком виде:

1) <имя> <операция сравнения> <операнд>

2) <*Rn*>. *MKED* (<*T*>)

3) <*Rn*>. *UNMKED* (<*T*>)

В поле <имя> задается выражение (<*Rn*>). <*Dn*>). В поле <операция сравнения> указываются следующие операции: =, ≠, <,

≤, >, ≥. В поле операнда может быть указан адрес операнда в БМ или ПЭ. Приведем пример условия:

СОТР. ЗАРП > Q & (СОТР. MKED(A))

Здесь выделяются кортежи отношения СОТР со значением зарплаты больше *Q*, в которых маркер *A* установлен в 1.

Для дальнейшего использования примем следующие сокращения:

СОТР — сотрудник,	УПР — управляющий,
ОТД — отдел,	НОМИЗ — шифр изделия,
ЗАРП — зарплата,	ФАМ — фамилия,
КОЛ — количество,	ИМИЗ — имя изделия.

Рассмотрим систему команд ПРП (всего 11 команд) на примере обработки БД (рис. 6.12):

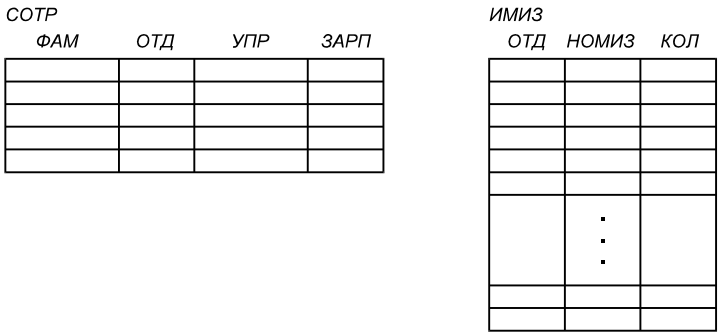


Рис. 6.12. Пример БД

1) *SELECT* <МАРКЕРЫ>[<*Rn*>: <УСЛОВИЕ>]

Пример:

S*ELECT MARK* (*AB*) [СОТР:СОТР.ОТД = 202]

В те кортежи, в которых имеются сотрудники из отдела 202, в *A* и *B* записывается 1.

2) *READ-ALL*<МАРКЕРЫ>[<*Rn*>(<*D*1>, ...): <УСЛОВИЕ>] [<АДРЕС>]

Пример 1:

READ-ALL [СОТР:УПР = КУЗНЕЦОВ] [*BUF*].

В буфер *BUF* посылаются данные обо всех сотрудниках отдела, возглавляемого Кузнецовым.

Пример 2:

READ-ALL [СОТР (ФАМ, ЗАРП): СОТР. *MKED*(*A*)] [*BUF*].

В буфер *BUF* записываются фамилии и зарплата отмеченных сотрудников.

3) *READ*[<*n*>]<МАРКЕРЫ>[<*Rn*>(<*Di*>,...): <УСЛОВИЕ>] [<АДРЕС>] — соответствует команде READ-ALL, но в буфер BUF посылается только *n* кортежей;

4) *READ-REG* [<список регистров>] [<адрес>] — записывает содержимое указанных регистров в БМ;

5) *CROSS-SELECT*<R1 МАРКЕРЫ>[<*R*1>:<*D*1> <ОПЕРАЦИЯ СРАВНЕНИЯ><*R*2>.<*D*2>] [<*R*2>. MKED (<T>)] — сложная команда. По ней устанавливаются маркеры в *R*1, если домен *D*1 из *R*1 сравнился с <*R*2>.<*D*2>, при этом рассматриваются только *T*-маркированные кортежи *R*2.

Пример:

SELECT MARK (A) [ИМИЗ. КОЛ > 100] CROSS SELECT MARK (A) [СОТР: СОТР. ОТД = ИМИЗ. ОТД [ИМИЗ. MKED (A)].

Будут промаркированы записи о служащих, которые продали более 100 единиц продукции.

6) *GET-FIRST-MARK* <МАРКЕРЫ> [<*R*n> (<*D*1>, ...): <*Da*>) <ОПЕРАЦИЯ СРАВНЕНИЯ> <*Db*>] [MKED(<*T*>] — выполняет операцию проектирования. Значение домена <*Db*> некоторого *T*-маркированного кортежа последовательно сравнивается со значением домена <*Da*> всех кортежей данного отношения. При совпадении устанавливаются маркеры кортежей, в которых произошло совпадение; сбрасывается *T*-маркер опорного кортежа; значения доментов (<*D*1>, ...) записываются в память АЛУ;

7) *SAVE*(<*n*>)[<*Rn*>(<*D*1>, ...): <УСЛОВИЕ>] [<СПИСОК РЕГИСТРОВ>] — в регистры записываются для *n* кортежей домены, удовлетворяющие условию;

8) *SUM*<МАРКЕРЫ>[<*Rn*>(<*Dn*>):<УСЛОВИЕ>] [<РЕГИСТР>]

COUNT <МАРКЕРЫ> [<*Rn*>(<*Dn*>): <УСЛОВИЕ>] [<РЕГИСТР>]

где *SUM* суммирует значения домена; *COUNT* подсчитывает число доменов;

9) *DELETE* [<*Rn*>:<УСЛОВИЕ>] — исключает все кортежи, удовлетворяющие условию;

10) ВС<МЕТКА>,<УСЛОВИЕ ПЕРЕХОДА> — обычный условный переход;

11) *EOQ* — завершает выполнение программы.

Рассмотрим три примера программ для ПРП. Первая программа исключает из отношения СОТР записи обо всех сотрудниках отдела Кузнецова:

```
SAVE [СОТР(УПР) : СОТР.ФАМ = КУЗНЕЦОВ] [REG 0]
DELETE [СОТР : СОТР. УПР = REG 0]
```

Вторая программа подсчитывает количество видов товарной продукции и общий объем товаров, проданных отделом, в котором служит Петров:

```
1. SELECT          MARK(A)[СОТР:СОТР.ФАМ = 'ПЕТРОВ':]
2. CROSS SELECT    MARK(A)[ИМИЗ:ОТД=СОТР.ОТД][СОТР]:
                   : MKED(A)]
3. COUNT           [ИМИЗ:MKED(A)][REG 1]
4. SUM             [ИМИЗ(КОЛ):MKED(A)][REG 2]
5. READ-REG        [REG 1, REG 2] [BUF]
6. SELECT          RESET (A) [ИМИЗ]
7. SELECT          RESET (A) [СОТР]
8. EOQ
```

По команде 1 для служащего Петрова маркируется кортеж СОТР.

По команде 2 маркируются кортежи ИМИЗ, относящиеся к отделу, в котором работает Петров. По командам 3,4 подсчитывается количество видов маркированной товарной продукции и общий объем последней, результаты заносятся в регистры 1, 2. По

команде 5 эти данные переписываются в область *BUF* базовой машины. Команды 6 и 7 устанавливают в 0 маркеры.

В третьей программе выполняются операции проектирования и условного перехода. Допустим нужно подсчитать число отделов, в которых заработок сотрудников больше *Q* руб. Используем такую программу:

1. SELECT	MARK(AB)[СОТР:СОТР.ЗАРП > Q]
2. LOOP-GET-FIRST-MARK	MARK(C)[СОТР:СОТР.ОТД = СОТР.ОТД] [MKED(B)]
3. SELECT	RESET(ABC)[СОТР.СОТР.MKED(C)]
4. BC	A, TEST(СОТР:MKED(B))
5. COUNT	[СОТР:СОТР.MKED(A)][REG 1]
6. EOQ	

Для конкретизации первая команда *SELECT* “маркирует”, например, Кузнецова и Вашкевича из четырнадцатого, Вершинину, Афанасьева и Лебедева из девятнадцатого и Ушакова из двадцатого отделов. При первом выполнении второй команды *GET-FIRST-MARK* будет найден один из пяти кортежей (например, Кузнецов). В результате станут *C*-маркированными все остальные *B*-маркированные кортежи того же отдела, при этом *B*-маркер Кузнецова будет сброшен. Третья команда *SELECT* сбросит *ABC* во всех *C*-маркированных кортежах (относящихся к Вашкевичу). При выполнении четвертой команды *BC* будут обнаружены *B*-маркированные кортежи. Затем все повторяется.

Пусть теперь вторая команда *GET-FIRST-MARK* найдет Афанасьева, выполнит *C*-маркирование кортежа Вершининой и Лебедева, сбросит *B*-маркеры кортежа Афанасьева. Третья команда *SELECT* сбросит *ABC* кортежей Вершининой и Лебедева. Опять следует повторение. Вторая команда *GET-FIRST-MARK* установит в 0 *B*-маркер Ушакова, однако кортежей, нуждающихся в маркировании, больше не обнаружит. Третья команда *SELECT* не найдет кортежей, удовлетворяющих условию. Четвертая команда *BC* не обнаружит *B*-маркированных кортежей. Тогда выполняется пятая команда *COUNT* и выдается ответ: 3 (Кузнецов, Афанасьев, Ушаков).

Автоматическая генерация программ. В ЭВМ пятого поколения предусматривается автоматическая генерация программ вычислений на основе заданного человеком условия задачи. Уже

разработаны и экспериментально проверены некоторые из таких методов.

Большую известность получил *метод вычислительных моделей* [11], предназначенный для задач с четко определенными отношениями. Система на основе вычислительных моделей содержит базу знаний, предметную и операционную области. В базе знаний находятся наиболее общие понятия и правила вывода, обеспечивающие логику работы для предметных областей всех видов.

В предметной области содержатся знания, относящиеся к конкретной области научной или инженерной деятельности. Область операций хранит условия задачи, начальные данные, копию нужных разделов предметной области и т.д.

Рассмотрим пример. Пусть предметная область есть модель прямолинейного движения: тело массой m движется прямолинейно с ускорением a в интервале $t_1 \ldots t_2$ с начальной скоростью v_1 и конечной скоростью v_2; v— средняя скорость; Δs — пройденный путь; f— сила инерции; ΔE— выполненная силой f работа; E_1 и E_2 — кинетическая энергия в начале и конце движения соответственно. Физические уравнения для этой предметной области таковы:

$$\begin{array}{lll} f = ma, & \Delta v = v_2 - v_1, & E_2 = mv_2^2 / 2, \\ \Delta E = f\Delta s, & E_1 = mv_1^2 / 2, & 2v = v_1 + v_2 \\ \Delta t = t_2 - t_1, & \Delta s = v\Delta t, & \\ \Delta v = a\Delta t, & \Delta E = E_2 - E_1, & \end{array} \qquad (6.23)$$

Предметная область для прямолинейного движения в графической форме представлена на рис. 6.13. Графическая модель наглядна, но в памяти ЭВМ она должна представляться в форме текста (6.23).

Задание на вычисления в этой предметной области формируются двояко:

1) “Зная ДВИЖЕНИЕ вычислить $v_1, \Delta E$ по m, Δt, a, v_2”;

2) “Зная ДВИЖЕНИЕ составить программу вычисления $v, \Delta E$ по $m, \Delta t, a, v_2$.”

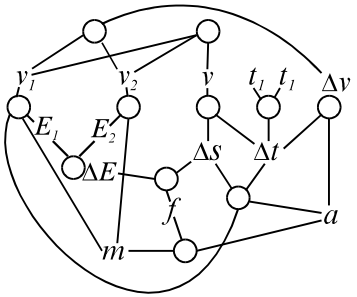


Рис. 6.13. Модель предметной области для прямолинейного движения.
Кружками обозначены отношения между переменными

В первом случае генерируется программа, по ней производятся вычисления и пользователь получает готовый результат. Во втором случае пользователь получает только программу вычислений.

Функционирование описанной выше вычислительной модели напоминает поиск одного или нескольких кратчайших путей в графе (см. рис. 6.13):

1) оператор вычислительной модели (отношение) может быть вычислен, если его входы определены;

2) оператор стоит применять только тогда, когда он дает новые результаты, которые ранее не вычислялись;

3) если есть различные пути вычисления результата, то с точки зрения правильности его получения не имеет значения, какой путь выбрать (но не с точки зрения длины программы).

Можно, например, получить две следующие программы (они являются различными путями в предметной области):

1. $\Delta v = a\Delta t$	1. $\Delta v = a\Delta t$
2. $f = m / a$	2. $v_1 = v_2 - \Delta v$
3. $v_1 = v_2 - \Delta v$	3. $E_1 = mv_1^2 / 2$
4. $v = v_1 + v_2 / 2$	4. $E_2 = mv_2^2 / 2$
5. $E_1 = mv_1^2 / 2$	5. $\Delta E = E_2 - E_1$
6. $E = mv_2^2 / 2$	
7. $\Delta E = E_2 - E_1$	

Существуют специальные приемы выбора более коротких (по некоторым критериям) программ (“прополки” различного рода). Принцип функционирования такой вычислительной модели строго

соответствует принципу функционирования описанных в § 3.3 ЭВМ с управлением от потока данных. В обоих случаях срабатывание *A* оператора происходит после того, как на все его входы поступили данные. Поэтому, используя систему обозначений § 3.3, каждый оператор (отношение) вычислительной модели необходимо представить в виде команды для DF-ЭВМ. Само же графическое описание вычислительной модели, по существу, соответствует описанию на графическом языке Денниса.

Функционирование вычислительной модели можно обеспечить и на многопроцессорных ЭВМ (типа МКМД). Но и в этом случае необходимо реализовать запуск операторов после появления данных, т.е. опять же обеспечить управление от потока данных.

Однако поскольку в МКМД-ЭВМ отсутствуют необходимые аппаратные средства, такие машины будет менее эффективны, чем ЭВМ с управлением от потока данных, в которых этот вид управления на всех уровнях поддерживается аппаратурой.

Контрольные вопросы

1. Перечислите основные характеристики параллельных алгоритмов.
2. Охарактеризуйте основные свойства рекурсивных алгоритмов.
3. Перечислите основные способы умножения матриц на параллельных ЭВМ.
4. В чем заключаются особенности алгоритмов вычисления ДУЧП на параллельных ЭВМ?
5. В чем заключаются особенности умножения матриц на конвейерных ЭВМ, процессорных матрицах и многопроцессорных ЭВМ?
6. Как выполняются операции нечисловой обработки в параллельных ЭВМ?
7. Что такое автоматическая генерация программ?

ЗАКЛЮЧЕНИЕ

Прогноз роста характеристик микропроцессоров на ближайшие годы указывает на следующие тенденции [24]:

Характеристика	Темп удвоения характеристики	Состояние на 1995 год	Перспектива на 2000 год
Частота синхронизации	3,6 года	10 МГц	300 Мгц
Шаг сетки кристалла	7 лет	0,65 мк	0,4 мк
Размер кристалла	5 лет	450 мм^2	900 мм^2
Число транзисторов на кристалле (процессор)	2 года	6 млн	50-100 млн
Разрядность шин данных	5 лет	160	-

Таким образом, к 2000 году за счет параллелизма и повышения тактовой частоты возможно получение быстродействия триллионы оп/с. Это позволяет перейти к новому способу оценки быстродействия ЭВМ, определяя его как задержку (время реакции) при решении крупных задач.

Высокие характеристики микропроцессоров во многом решают проблему аппаратуры, но выдвигают ряд новых проблем в области архитектуры и программирования. Не касаясь проблем, общих для всей вычислительной техники, перечислим некоторые проблемы, связанные непосредственно с параллелизмом:

1. Исследования численных методов показывают (см. гл. 6), что в них имеется достаточно большой уровень параллелизма.

С другой стороны, характеристики микропроцессоров таковы, что представляется возможным строить системы с сотнями и тысячами процессоров. Таким образом, развитие параллельных ЭВМ обосновано как со стороны алгоритмов, так и со стороны аппаратуры. Однако большое разнообразие параллельных архитектур приводит к распылению сил в разработке параллельной техники. Это означает, что вопросы унификации параллельных архитектур становятся центральными. Одной из наиболее перспективных является архитектура МКМД-ЭВМ на базе *n*-кубов, в узлах которой используются векторные и суперскалярные процессоры.

2. Параллельное программирование является слишком сложным для массового пользователя, поэтому вопросы автоматизации распараллеливания, планирования и отладки пользовательских программ становятся ведущими в разработке программного обеспечения параллельных ЭВМ.

3. Выживаемость любой новой архитектуры в первую очередь определяется возможностью быстрого создания большого объема прикладного программного обеспечения, поэтому разработка унифицированных пакетов и средств автоматизации переноса прикладных пакетов приобретает большое значение.

Все упомянутые выше проблемы в той или иной мере нашли отражение в настоящем издании.

ПРИЛОЖЕНИЕ 1

ХАРАКТЕРИСТИКИ ЭТАЛОННЫХ ПРОГРАММ

Ниже приводятся краткие описания эталонных программ (benchmarks) — тестов, используемых для оценки характеристик микропроцессоров и ЭВМ. Большинство описываемых тестов позволяет оценить эффективность векторизующих и оптимизирующих компиляторов. Некоторые тесты из-за большого объема не помещаются в КЭШ, что дополнительно позволяет проверить эффективность подсистемы обращения к КЭШ.

Dhrystone. Этот стандартный тест спроектирован для изменения как характеристик аппаратуры, так и эффективности компиляторов. Он включает с определенным весом вызовы процедур, циклы, логические и целочисленные арифметические операции. результат выражается в D/s (Dhrystone per second).

Whetstone. Тест спроектирован для оценки характеристик систем при интенсивном использовании операций с плавающей запятой. Это смесь плавающих и целочисленных операций, вычисления трансцедентных функций, обработки массивов и вызовов процедур. Тест основан на статистике научных программ, написанных на Фортране, имеет небольшой размер и помещается в КЭШ. Результат выражается в Mwhet/s (Millions Whetstone per Second).

SPECmark92. Содержит две составляющие: SPECint92 и SPECfp92. Тест SPECint оценивает характеристики процессора и является типичной смесью команд для коммерческих применений. Тест SPECfp используется для технических и научных приложений, где преобладают операции с плавающей запятой. SPECint дает среднее значение для 4-х программ, написанных на Си, результат выражается в МИПС. SPECfp дает среднее значение 6-ти программ, написанных на Фортране, результат выражается в МФлопс.

Linpack. Тест большого объема для научных расчетов на основе пакета для решения систем линейных уравнений. Преимущественно содержит операции с плавающей запятой. Результат — в МФлопс.

Livermore Fortran Kernel. Тест содержит 24 фрагмента на Фортране из реальных программ в области физики. Предназначен для суперЭВМ. Результат — в МФлопс.

Paradox. Этот тестовый файл содержит свыше 7000 имен, адресов и расчетных данных. Измеряется время пяти операций: загрузка и преобразование dBASE файла в текстовый файл, который затем передается на принтер.

AutoCAD. Этот тест содержит трехмерный чертеж из области архитектуры объемом 172 Кбайт. Тест включает пять операций: прорисовку, удаление скрытых линий и др. Измеряется время выполнения теста.

MathCAD. Тест предназначен для решения уравнений в области электроники с интенсивным использованием целочисленных и плавающих операций. Измеряются времена для загрузки и расчета документа, включающего операции над матрицами, двойное интегрирование, вычисление производных и быстрое преобразование Фурье.

ПРИЛОЖЕНИЕ 2

ХАРАКТЕРИСТИКИ ЭВМ ТИПА CRAY

Машины фирмы CRAY RESEARCH относятся к классу векторно-конвейерных ОКМД-ЭВМ, однако в системе может быть несколько таких процессоров, которые совместно образуют на более высоком уровне МКМД-ЭВМ с общей памятью.

Фирма CRAY RESEARCH была создана С.Креем в 1972 году и в 1976 году было начато производство ЭВМ CRAY-1. В 1982 году производство CRAY-1 было прекращено, в том же году было начато производство многопроцессорных ЭВМ CRAY-X-MP. В 1985 году начато производство ЭВМ CRAY-2, в 1990 году — CRAY-3, в 1993 году — CRAY-4.

Векторный процессор ЭВМ CRAY содержит скалярную и векторную часть. В соответствии с этим команды делятся на две группы: скалярные и векторные. Всего имеется около 100 (в старших моделях набор команд несколько расширен) команд. Команды имеют фиксированный 16 или 32-разрядный формат и разделены на группы для работы с РОН и для работы с памятью. Для увеличения быстродействия используются команды с зацеплением операций (в одной команде задано выполнение двух операций).

Объединение процессоров в многопроцессорную систему требует введения дополнительного оборудования и команд для синхронизации и обмена данными. Связь осуществляется по принципу каждый с каждым, память является разделяемой всеми процессорами. Ниже приводятся некоторые характеристики ЭВМ CRAY.

Таблица П. 2.1.

Характеристики ЭВМ CRAY-1

N пп	Наименование ЭВМ	Год выпуска	Объем памяти, млн. слов	Число блоков памяти	Частота синхрони-зации, МГц	Быстродействие, Мфлопс (LINPACK)
1	CRAY-1	1976	1	16	80	50
2	CRAY-1/S	1979	4	16	80	76
3	CRAY-1/M	1982	4	16	80	83

Таблица П. 2.2.

Характеристики ЭВМ CRAY-X-МП

N пп	Наименование ЭВМ	Год выпуска	Число процессоров	Объем памяти, млн слов	Число блоков памяти	Частота синхронизации, МГц	Быстродействие, Мфлопс, (LINPACK)
1	CRAY-X-МП-18	1984	1	8	32	105	171
2	CRAY-X-МП-216	1982	2	16	32	105	257
3	CRAY-X-МП-48	1984	4	8	32	105	480
4	CRAY-X-МП-416	1984	4	16	64	105	480

Таблица П. 2.3.

Характеристики старших семейств ЭВМ CRAY

N пп	Наименование ЭВМ	Год выпуска	Число процессоров	Объем памяти, млн слов	Частота синхронизации, МГц
1	CRAY-Y-MP	1988	8	32	166
2	CRAY-2	1985	4	256	240
3	CRAY-3	1990	16	2048	500
4	CRAY-4	1993	64	более 2048	1000

Для CRAY-1 и CRAY-X-MP разработаны операционные системы COS (Cray Operating System) и CTSS (CRAY Time Sharing System). Последняя используется на всех типах ЭВМ CRAY. Имеются трансляторы для Фортрана CTF (Cray Fortran Computer — с автоматической векторизацией циклов), с языка Паскаль (ISO), ассемблера (CALL). Имеется значительный объем пакетов прикладных программ.

Для CRAY-2 используется ОС UNIX, которая поддерживает компиляторы CTF, ISO, CAL и для языка Си.

ПРИЛОЖЕНИЕ 3

ХАРАКТЕРИСТИКИ ЭВМ НА ОСНОВЕ ПРОЦЕССОРНЫХ МАТРИЦ

N пп	Наименование ЭВМ, страна, год	Макс. число ПЭ	Разряд-ность ПЭ (бит)	Тип комму-татора	Быстро-действие	Объем памяти	Тип управляющей ЭВМ
1	ILLIAC-1V (США, Burroughs, 1974)	4×64	64	матрица	200 Мипс для 64ПЭ	2 К слов на 1 ПЭ	Унив. ЭВМ В6500
2	Propal-2 (Фран-ция, Tomson-CSF, 1976)	2048	1	на основе кольца	600 Мипс	32 Мбт	MITRA 15
3	BSP (США, Burroughs, 1979)	16	48	коордиинат-ный пере-ключатель	50 МФлопс	8 М слов	Собственная разработка
4	DAP (Англия, ICL, 1980)	32×32	1	Матрица	20 Мфлопс, 100 Мипс	2 Мбайт	ICL 2900
5	ПС-2000 (СССР, 1982)	64	24	Кольцо	200 млн коротких оп/с	16 К слов на 1 ПЭ	Мини-ЭВМ СМ-2
6	MPP (США, Goodyear, 1983)	128 ×128	1	Матрица	400 Мипс	Более 2 Мбайт	VAX 11/780
7	GF11 (США,			Сеть	20 Мфлопс		

	IBM, 1988)	576		Бенеша	на 1 ПЭ	1,1 Гбайт	-

ПРИЛОЖЕНИЕ 4

ХАРАКТЕРИСТИКИ СУПЕРЭВМ ТИПА ЭЛЬБРУС И ЕДИНОЙ СИСТЕМЫ ЭВМ

Машины типа “Эльбрус” относятся к классу многопроцессорных ЭВМ с общей памятью (МКМД) и разработаны в Институте точной механики и вычислительной техники АН СССР (ИТМВТ).

ИТМВТ имеет богатый опыт в разработке быстродействующих ЭВМ, им разработаны наиболее быстрые в шестидесятые годы ЭВМ “БЭСМ” и “Чегет”. В семидесятых годах начата разработка суперЭВМ семейства “Эльбрус”, основной особенностью которого является реализация языка высокого уровня “Эль-76”. Наибольшее развитие принципы семейства “Эльбрус” получили в ЭВМ “Эльбрус-3”, процессоры которой построены по принципу сверхдлинной команды. Это позволяет получать в процессоре до 7 результатов за 1 такт.

Все модели “Эльбрус” совместимы по системе программирования

№ п/п	Наименование ЭВМ	Год выпуска	Число процессоров	Объем памяти, Мбайт	Число блоков памяти	Частота синхронизации	Быстродействие
1	Эльбрус-1	1978	10	4	32		12 млн оп/сек
2	Эльбрус-2	1983	10	144	32		12 млн оп/сек
3	Эльбрус-3	-	16	2000		100	10 Гфлопс
4	Эльбрус-3Б	-	20				10 Гфлопс

Другое направление в разработке суперЭВМ реализуется головной организацией Единой системы ЭВМ (ЕС ЭВМ) — научно-исследовательским центром электронной вычислительной техники (НИЦЭВТ). НИЦЭВТ в 1986 году начал проект суперкомпьютера 1191 из четырех скалярных и одного общего векторного процессора с быстродействием до 1,2 Гфлопс, в котором реализовывался дружественный интерфейс ЕС ЭВМ. Из-за недостатка финансирования этот проект в 1991 году был заморожен, а на его основе бы-

ла развернута разработка семейства малогабаритных суперкомпьютеров ЕС119Х.Х. в частности, в 1993 году был создан опытный образец ЭВМ ЕС 1195 с быстродействием 50 Мфлопс и памятью объемом 256 Мбайт. Эта ЭВМ занимает площадь менее 1 $м^2$ и потребляет мощность не более 5 кВа. В 1995 году началось изготовление опытного образца векторного суперкомпьютера ЕС 1191.01 с пиковой производительностью 500 Мфлопс, проектируется система ЕС 1191.10 с производительностью 2 Гфлопса в минимальной конфигурации. Все модели ЕС 119Х.Х предназначены для работы по Unix или OS/2 и сконструированы в "офисном исполнении", то есть имеют малые габариты и энергопотребление.

Недостатком ЭВМ ЕС 119Х.Х является использование устаревшей элементной базы (1500 вентилей на кристалл), поэтому развитие идей ЕС 119Х.Х НИЦЭВТ начал работы по созданию суперкомпьютеров "АМУР" на базе КМОП — микросхем со степенью интеграции 200 тыс. вентилей на кристалл. Программа рассчитана на 3 года и позволит строить ЭВМ с быстродействием от 50 Мфлопс до 20 Гфлопс. Особенностью семейства "АМУР" является использование одного и того же комплекта из семи микросхем для построения всех моделей.

ПРИЛОЖЕНИЕ 5

ХАРАКТЕРИСТИКИ МКМД-ЭВМ НА ОСНОВЕ *n* - КУБОВ

Примером МКМД-ЭВМ с общей памятью являются многопроцессорные ЭВМ CRAY-X-MP, CRAY-2, 3, 4 (Приложение 1). Представителями ЭВМ с локальной памятью являются системы на базе транспьютеров (Приложение 4).

В настоящем приложении приведены характеристики МКМД-ЭВМ с общей и локальной памятью, построенные на основе n-кубов (иное название — гиперкуб, космический куб). Такие кубы обычно имеют 2^n узлов, где n — число связей каждого узла. В системе с 2^n узлами, все узлы связаны прямыми связями. Возможны системы, содержащие больше 2^n узлов, но с более простыми типами связей (кольцо, матрица, куб и т. д.).

Узел обычно состоит из вычислительного процессора и процессора связи. В качестве последнего часто используются транспьютеры. В качестве вычислительного процессора используются мощные МП фирм Intel (i386, i486, Pentium), Motorola (680x0), МП Power PC или процессоры оригинальной разработки.

Число систем на *n*-кубах достаточно велико, и характеристики таких процессоров постоянно растут как по числу узлов, так и по быстродействию каждого узла. Ниже приведены текущие характеристики систем на *n*-кубах наиболее известных фирм.

N пп	Наименование, фирма, год	Вид памяти	Число узлов	Тип МП узла	Объем памяти	Быстродействие	Состав ПО
1	FPS-T (FPSyst., 1985)	И	4096	умножитель, сумматор	1 Мбайт на узел	49 Гфлопс	ОС VMS, Оккам, Фортран
2	IPSC (Intel, 1985)	И	128	i286, 386, 486, Пентиум	1 Мбайт на узел	4 Мфлопс (i286)	ОС XENIX, языки для i80X86
3	BBN Butterfly (BBN, 1985)	О	256	МП 680X0	1-4 Мбайт на все узлы	256 Мипс	ОС Chrysalis, Си, Фортран
4	NCUBE (NCUBE 1988)	И	1024	32-разрядный МП	128 Кбайт на узел	500 Мфлопс 2 Гипс	ОС AXIS, VERTEX (для узлов), Си, Фортран
5	Система Parsytec-Motorola (1994)	О	128	Power PC 601/80, 604/100, 620		До 80 ГФлопс	Фортран, Си, Си+

Примечание: И — индивидуальная память, О — общая память

ПРИЛОЖЕНИЕ 6

ХАРАКТЕРИСТИКИ ТРАНСПЬЮТЕРОВ

Транспьютеры выпускаются фирмой INMOS, начиная с 1985 года. За исключением транспьютера T212, который имеет только две связи и 16-разрядные АЛУ, все остальные типы транспьютеров имеют 4 связи и 32 или 64-разрядные АЛУ. Система команд транспьютеров относится к классу CISC.

В семействе транспьютеров только T9000 является суперскалярным, то есть может выполнять более 1 команды за такт. Он имеет следующие особенности:

1. Введен 5-ступенчатый конвейер: выборка команды, генерация адреса, выборка операнда из памяти, операция АЛУ-I/АЛУ-F, запись результата.

2. В T9000 возможно одновременное выполнение нескольких команд как за счет конвейеризации, так и за счет распараллеливания операций на ступенях конвейера. Так, ступень 1 позволит извлечь значения двух локальных переменных, ступень 2 — вычислить два адреса, ступень 3 — извлечь два значения из памяти.

3. Внутренняя память имеет объем 16 Кбайт и в зависимости от необходимости может использоваться как КЭШ или адресная память.

4. Введено устройство группирования команд, которое выявляет пары параллельных инструкций в процессе исполнения программы.

5. Введен блок виртуальных каналов, число которых для пользователя не ограничено. Аппаратура динамически преобразует номера виртуальных каналов в номера физических каналов. В результате введения виртуальных каналов значительно возросла загрузка физических каналов.

6. В комплект T9000 входит микросхема коммутатора IMS C104, которая имеет 32×32 неблокируемых одноразрядных линий. C104 принимает на вход линии заголовков передаваемого пакета, за 700 нс по заголовку определяет точку выхода пакета, подключает этот выход и передает сообщение в темпе работы связей T9000. C104 позволяет строить многотранспьютерные системы практически неограниченного размера.

Транспьютер T9000 выпущен в серию в 1994 году. Назначение — резко поднять быстродействие за счет суперскалярной обработки и тем самым восстановить позиции в конкурентной борьбе с новыми RISC-МП.

В состоянии разработки находится новый транспьютер Hameleon (1996 год).

В комплект транспьютерных наборов входят вспомогательные кристаллы: IMS C011, C012 — для связи с ПЭВМ, коммутаторы C004, C104, ряд контроллеров.

Транспьютеры используются как для построения мощных многотранспьютерных систем (сотни и тысячи транспьютеров), так и для построения акселераторов для ПЭВМ. В области акселераторов работают фирмы: INMOS (семейство плат TRAM), Microway (Biputer, Quadputer), Quintek (Fast4, Fast9, Fast17) и многие другие фирмы.

Ниже в таблице приводятся некоторые характеристики транспьютеров.

N пп	Тип транспьютера, год	Состав АЛУ, объем внутренней памяти	Частота синхронизации, МГц	Быстродействие	Число транзисторов, млн
1	T414	АЛУ-I-32 разр.	15	10 Мипс	
	(1985)	2 Кбайт	20	0,1 МФлопс	0,2
2	T800	АЛУ-I-32 разр.	20		
		АЛУ-F-64 разр.	30	2,2 МФлопс	0,3
		4 Кбайт			
3	T9000	АЛУ-I-32 разр.	20		
	(1995)	АЛУ-F-64 разр.	40	200 Мипс	
		16 Кбайт	50	25 МФлопс	3,3

Примечание: АЛУ-I, АЛУ-F — соответственно АЛУ для целочисленной и плавающей арифметики

ПРИЛОЖЕНИЕ 7

ХАРАКТЕРИСТИКИ СУПЕРСКАЛЯРНЫХ МИКРОПРОЦЕССОРОВ

Первый суперскалярный МП i960 был выпущен фирмой Intel в 1987 году. Затем были разработаны МП SPARC (1987-1989 годы), MIPS (1988-1989 годы), МПi860 (1989 год), Motorola 88×00 (1988 год) и ряд других суперскалярных МП, в частности:

1. Микропроцессор Pentium был впервые поставлен фирмой Intel в 1993 году как продолжение семейства МП 80×86. Цель его создания — получение быстродействия RISC-МП и полная совместимость на уровне двоичных кодов с программным обеспечением, созданным для всех МП 80×86. В настоящее время разрабатываются МП P6 (4 конвейера, 1995 год) и P7 (6 конвейеров к 2000 году). МП P7 предположительно будет содержать 4 скалярных и 2 векторных процессора, которые будут функционировать на частоте 250 Мгц. Каждый из скалярных процессоров будет содержать 4 млн транзисторов и иметь скорость 700 MIPS. Внутренний КЭШ будет иметь объем 2 Мбайт. P6 и P7 снизу вверх будет полностью совместимы с МП 80×86.

2. Группа фирм AIM (APPLE + IBM + MOTOROLA) совместно разработали семейство МП POWER PC и выпустили его первый образец МП 661 в 1993 году. Семейство предназначено для создания новой идеологии ПЭВМ на базе RISC, которые в перспективе заменят ПЭВМ PC IBM, созданных на базе МП фирмы Intel.

3. Фирма DEC в 1992 году для создания мощных рабочих станций выпустила МП 21064 с тактовой частотой 250 Мгц, а затем более мощный МП — 21164.

4. В 1994 году фирма MIPS Computer, известная разработкой суперконвейерных МП, выпустила первый суперскалярный МП MIPS R8000 (MIPS — Microprocessor Without Interlocked Pipeline Stages), а затем МП R10000.

5. В 1994 году фирма Sun Microsystem Inc. в продолжение развития своей серии SPARC (Scalable Processor Architecture) выпустила мощный МП UltraSPARC.

6. В 1994-1995 годах фирмой Hewlett-Packard был выпущен МП PA7200 с высокими показателями быстродействия, предполагается к выпуску МП PA8000.

Все указанные МП являются суперскалярными и поэтому характеризуются рядом общих свойств, в частности:

1. Формирование группы команд для загрузки конвейеров производится динамически в каждом такте. Для этого аппаратно на этапе предвыборки и дешифрации производится анализ зависимости по данным смежных команд. В конвейеры для параллельного исполнения подбираются независимые команды, при этом допускается изменение порядка выполнения команд.

2. Все МП используют динамическое прогнозирование ветвлений на основе буфера истории переходов. Иногда используется одновременное выполнение альтернативных ветвей.

3. Некоторые МП строятся таким образом, что число физических регистров превышает число РОН, определенных архитектурно (PPC620, Mips R10000, P6). Это необходимо для реализации альтернативных ветвей при переходах и для устранения зависимостей по данным, вызванных недостатком РОН. В процессе выполнения команд необходимо производить переименование физических регистров, то есть они выступают в качестве виртуальных.

4. Для суперскалярных МП используются КМОП технология с шагом сетки 0,8...0,5 мк.

Большинство указанных МП выпускается в однокристальном исполнении, однако в целях получения более высокого быстродействия для МП PPC 620 использовано 10 кристаллов пяти типов, а для МП R8000 — 4 кристалла трех типов.

При разработке суперскалярных МП используются новые системы команд. Их число колеблется в широких пределах от 60...70 (UltraSPAC, Mips) до 220 (ALPHA DEC). Новизна системы команд создает проблемы в разарботке прикладного программного обеспечения.

Для МП Pentium без изменения переносится все системное и прикладное ПО на уровне двоичных кодов, созданное для линии 80×86. Однако при этом не используются все возможные архитектуры по МП, поэтому ряд фирм (WATCOM, SEMANTEC, MicroWay и др.) разработали оптимизирующие компиляторы, ко-

торые позволяют получить коды, работающие в 1,5-2 раза быстрее, чем немодифицированные.

Для МП Power PC и Alpha разработаны модификации операционных систем Windows NT, OS-2, Mac OS, Taligent, Magic Cops. Перевод громадного числа прикладных пакетов, написанных и существующих в кодах МП 80×86, в коды Power PC или Alpha не представляется возможным, по крайней мере, в короткое время. Поэтому для выполнения пакетов используются эмуляторы, например, Soft Windows, Wabi, Toolset и другие.

Для МП MIPS, SPARC и PA ранее наработан достаточно большой объем ПО, тем не менее и для них используются средства эмуляции.

Архитектура описанных выше суперскалярных МП приобретает традиционный характер, поэтому предпринимаются попытки освоить новые архитектуры. Одной из наиболее перспективных является разработка МП PA9000, производимая совместно фирмами Hewlett-Packard и Intel. Главная особенность PA9000 состоит в том , что генерация набора команд для одного такта полностью переносится в компилятор, что позволяет достичь высокого уровня оптимальности программы и значительно разгрузить кристалл от схем планирования и упаковки. Тем самым совершается переход к VLIW (Very Long Instruction Word) архитектуре МП. Выпуск PA9000 предполагается в 1998 году и он будет содержать около 20 конвейерных устройств. Предполагается совместимость на уровне двоичных кодов с МП i80×86 и PA7XXX и PA8XXX.

Ниже в табл. П 7.1 даны некоторые характеристики суперскалярных МП. В таблице приняты следующие сокращения: “Ц/П” — обозначает “целочисленный /с плавающей точкой”; “К/Д” — “команды/данные”. В граые 3 в скобках указана частота, при которой достигается приведенная в графе производительность.

Таблица П 7.1.

Структурные характеристики суперскалярных ЭВМ

№ п/п	Характеристика	Intel		IBM/Motorola		
		Pentium	P6	Power PC		
				601	604	620
1	Частота, МГц	66, 100	133	66, 100	100	133
2	Разрядность, бит	32	32	32	64	64
3	Производительность SPECint92/SPECfp92	65/57 (66)	200/200	105/125 (100)	160/165	225/300
4	Число АЛУ, Ц/П	2/1		1/1	3/1	3/1
5	Число ступеней, Ц/П	5/8	14	4/5	3	5
6	Число регистров, Ц/П	8/8	40+14	32/32	32/32	32/32
7	Число операций за такт	2	3	3	4	4
8	Объем КЭШ, Кбайт, К/Д	8/8	8/8	32	16/16	32/32
9	Число транзисторов, млн	3,1	5,5	2,8	3,6	6,88
10	Мощность, Вт	13	20	8	10	30

11	Год поставки	1993	1995	1993	1994	1995

Таблица П 7.1.(Продолжение)

№ п/п	Характеристика	DEC		Mips Tech		Sun	Hewlett-Packard	
		ALPHA		Mips		Ultra	PA	
		21064	21164	R8000	R10000	Sparc	7200	8000
1	Частота, МГц	133, 275	266/300	75	200	167	99/140	200
2	Разрядность, бит	64	64	64	64	64	64	64
3	Производительность SPECint92/SPECfp92	74/126 (150)	350/500	108/310	300/600	275/305	175/250 (140)	360/550
4	Число АЛУ, Ц/П	1/1	2/2	2/2	2/2	2/3	1/1	2/2
5	Число ступеней, Ц/П	7	7/9	5/4	5	6/9	5	
6	Число регистров, Ц/П	32/32	32/32	32/32	64/64	144/32	32/32	
7	Число операций за такт	2	4	4	4	4	2	4
8	Объем КЭШ, Кбайт, К/Д	8/8	8/8	16/16	32/32	16/16	1 Мбайт	
9	Число транзисторов, млн	1,75	9,3	3,4	6	3,8	1,3	
10	Мощность, Вт	27	50 (300)		30	30	29	

11	Год поставки	1992	1995	1994	1995	1995	1994	1996

ПРИЛОЖЕНИЕ 8

ОСОБЕННОСТИ АРХИТЕКТУРЫ МИКРОПРОЦЕССОРА P6 ФИРМЫ INTEL

Промышленное производство МП P6 и компьютеров на его основе началось в 1995 году. Основные характеристики этого МП даны в приложении 7. В настоящем приложении рассмотрены некоторые особенности его архитектуры.

Структурная схема МП P6 представлена на рис. П 8.1.

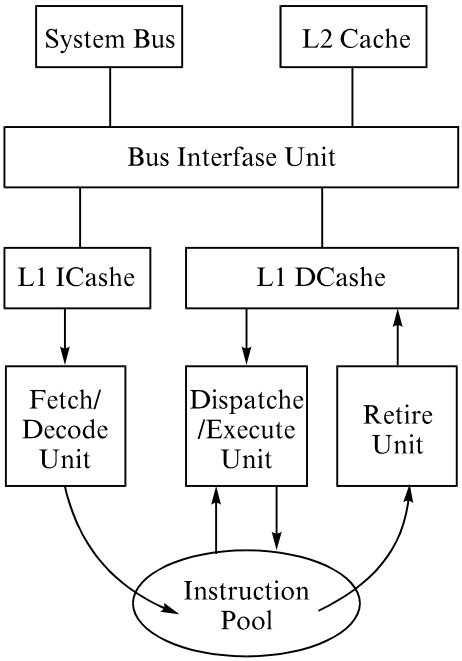


Рис. П 8.1. Структурная схема МР P6

КЭШ команд (L1 ICashe) и КЭШ данных (L1 DCashe) расположены внутри кристалла P6, а КЭШ второго уровня (внешний) объемом 256 Кбайт выполнен на отдельном кристалле. Оба кристалла (МР P6 и L2 Cashe) конструктивно расположены на единой подложке и взаимодействуют через независимую систему шин.

Основу МП P6 составляют три устройства: устройство выборки и декодирования команд — Fetch/Decode Unit (FD); устройство диспетчирования и выполнения команд — Dispatche/Execute Unit (DE); устройство удаления выполненных команд — Retire Unit (R). Все эти устройства функционируют независимо и обмениваются результатами своей работы только через буфер команд — Instruction Pool (IP). Рассмотрим поочередно работу этих устройств.

Устройство выборки и декодирования команд FD выполняет следующие функции:

1. Команды вводятся в буфер команд IP устройством FD, а удаляются из него — устройством R. Вместе с тем для полноценной работы устройства DE необходимо, чтобы в буфере команд IP находилось 20...30 команд программы, среди которых по статистике в среднем находится 4...5 команд условных переходов. Таким образом, первой функцией устройства FD является построение наиболее вероятной трассы выполнения программы путем динамического прогнозирования последовательных ветвлений. Предсказание каждого осуществляется с помощью большого буфера истории переходов BTB (Branch Target Buffer), содержащего 512 входов. Правильность предсказания обеспечивается более, чем в 90% случаев. В начальный момент, когда BTB еще не заполнен, используется аппаратно реализованный алгоритм статического предсказания.

2. На последующих ступенях конвейера FD производится замена исходных команд системы i80x86 на универсальные операции ***uops*** (universal operations). Каждая ***uop*** имеет два логических адреса для операндов и один — для результата. Большинство команд i80x86 преобразуются в одиночную ***uop***, некоторые — в две, три и более ***uop***. Сложные команды реализуются в виде микрокода из ***uop***.

3. Выстроенные в очередь ***uops*** посылаются в блок переименования регистров, Register Alias Table Unit (RAT). Дело в том, что небольшое число регистров общего назначения в системе команд i80x86 (всего 8) ограничивает параллелизм, внося ложные зависимости по данным.

Чтобы устранить ложные зависимости, в P6 используется 40 физических регистров, которые на время обработки заменяют

имена логических регистров, указанных в программе. Замена производится автоматически при каждом обращении на запись в данный регистр. Соответствие между номерами физических и логических регистров хранится в RAT. При каждом обращении к логическому регистру на чтение, производится обращение к RAT.

На выходе из устройства FD каждая ***иор*** снабжается статусной информацией (номер команды, номер ***иор***, готовность операндов и др.) и передается в буфер команд IP, который выполнен в виде контекстно адресуемой памяти. Пропускная способность устройства FD — 3 команды за один такт. Все команды обрабатываются в порядке их расположения в программе (in order). Для работы устройства FD используется 6 первых ступеней конвейера.

Структура диспетчирования и выполнения DE представлена на рис. П 8.2.

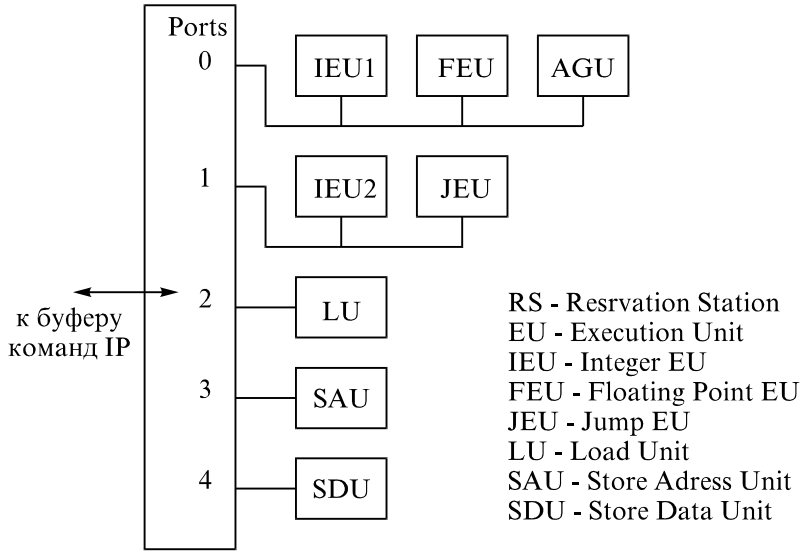


Рис. П 8.2. Структура устройства DE диспетчирования и выполнения команд

Блок диспетчирования RS в каждом такте, выбирает из IP готовые для исполнения ***иорs*** и запускает их на исполнение параллельно по каждому из 5 портов. Естественно, в каждом такте по одному порту можно запустить только одну ***иор***.

Функции устройства DE следующие:

1. Выбор готовых для исполнения в данном такте ***иops***. Алгоритм выбора основан на том, что каждая ***иop*** содержит биты состояния, облегчающие готовность операндов. Если оба операнда готовы, то ***иop*** назначается на соответствующий порт. Назначение является простой операцией, если в данном такте готова только одна ***иop***. Если готовых операций несколько, то необходимо оптимальное планирование. Идеальным было бы выбирать такие ***иops***, которые уменьшали длину информационного графа отрезка программы. Поскольку такое планирование требует слишком много времени, то в P6 используется значительно более простой алгоритм типа "первый пришел — первый ушел" (FIFO).

На рис П 8.3 представлен пример запуска и выполнения ***иops*** исполнительными блоками устройства DE. На рисунке затемненные команды можно выполнять параллельно, а светлые — только последовательно.

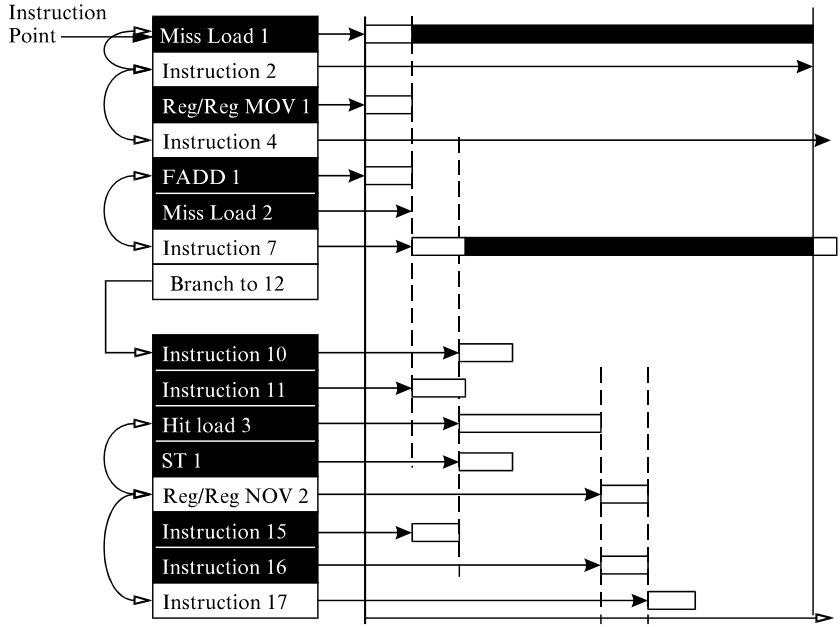


Рис. П 8.3. Диаграмма выполнения отрезка программы в устройстве DE

В первом такте запускается три ***иops***. Первая команда Load должна загружать регистр из L1 DCashe, но вследствие промаха ее выполнение затягивается на много тактов. Две других ***иops*** выпол-

няются за один такт. Во втором такте запускаются три следующие ***uops***, первая из них также соответствует промаху КЭШ. Из диаграммы видно, что выполняются четыре команды и стартует пятая (вторая Load с промахом КЭШ) перед окончанием второго такта, и все эти команды выполняются внутри задержки , вызванной первым промахом памяти. В этом случае в обычном суперскалярном МП после промаха выполнение последующих команд приостанавливается до появления данных из памяти.

Диаграмма на рис. П 8.3 показывает, что

- команды в DE выполняются в порядке их готовности, а не в порядке их размещения в программе (out of order), и это соответствует принципу функционирования процессоров типа Data Flow;
- выполнение команд является условным, т. е. результаты записываются во временные (физические) РОН, а не в логические РОН или память;
- каждый исполнительный блок (целочисленные и плавающие АЛУ, блоки обращения к памяти и др.) содержит несколько ступеней конвейера;
- документация по P6 содержит таблицы, в которых указано: на какое число и каких ***uops*** разбивается каждая команда 80x86; какой порт требуется для исполнения каждой ***uop*** и сколько тактов занимает это исполнение. Эта информация позволяет подсчитать время выполнения программы;
- в пиковом режиме устройство DE может запускать 5 ***uops***, но в длительном режиме — в среднем 3.

2. После выполнения ***uop*** в устройстве DE она возвращается в буфер команд IP, причем в ней один из битов состояния устанавливается в состояние “выполнена”. Кроме того, выявляются все команды, которые используют полученный результат в качестве операнда, и в этих командах устанавливается бит состояния “операнд готов”. Такое изменение состояния переводит некоторые команды в состояние готовности, и они могут запускаться на параллельное исполнение в следующем такте.

3. Когда в DE выполняется команда перехода, то результат ее выполнения сравнивается с предсказанным. Если они совпадают, то команды, которые находятся в программе после данной команды перехода и уже выполнены условно считаются выполненными правильно и их результаты могут быть переданы в память или логический регистр.

Если же результат выполнения команды не совпал с предсказанным, то блок JEU изменит состояние всех команд, находящихся в программе после данной команды перехода таким образом, чтобы они были удалены из IP, а устройство FD обеспечит выборку новой последовательности команд по новому адресу перехода.

Устройство удаления выполненных команд R работает следующим образом:

- просматривается буфер команд IP. Кандидатами на удаление являются все ***uops*** с признаком "выполнено";
- среди кандидатов на удаление выбираются те ***uops***, адреса которых составляют непрерывную последовательность по отношению к последней, ранее удаленной ***uop***, т. е. устанавливается исходная последовательность ***uops***, принятая для устройства FD (in order). За один такт может удаляться до трех смежных ***uops***;
- удаляемые одиночные ***uop*** или группы ***uops*** заменяются на исходные команды 80x86, и результаты записываются по соответствующим логическим адресам, т. е. окончательно. При этом не возникает вопроса о правильности результатов удаленных команд, так как выполненные команды переходов удаляются ранее следующих за ними обычных команд.

При удалении команд вносятся соответствующие изменения в буфер команд IP и другие таблицы. Например, если использование некоторого физического регистра прекращается при удалении команды, то он вычеркивается из таблицы RAT.

В заключение перечислим основные алгоритмы, аппаратно реализованные в P6.

1. Устройство выборки и декодирования FD:

- алгоритм предсказания направления переходов в нескольких последовательных командах условных переходов;
- механизм замены команд МП i80x86 соответствующим набором ***uops***;
- алгоритм переименования логических РОН на физические.

2. Устройство диспетчирования и выполнения ED:

- алгоритм выявления и распределения "готовых" команд по портам исполнительных устройств;
- алгоритм выполнения ***uops*** в исполнительных устройствах;
- алгоритм изменения статуса команд в буфере команд IP после выполнения очередной порции команд
- алгоритм отмены результатов выполненных команд и набора новых команд после выявления неверно предсказанного перехода.

3. Устройство удаления команд R:

- механизм выявления команд для удаления из IP;
- механизм преобразования ***uops*** в команды МП i80x86 и запись результатов по безусловным адресам;
- механизм внесения изменений в буфер команд и другие таблицы после удаления одной или нескольких команд.

Все эти алгоритмы, называемые в совокупности разработчиками Dinamic Execution, обеспечивают неблокируемость конвейера P6. Кроме того, конвейер P6 имеет средства для образования многопроцессорных систем.

В настоящее время МП P6 получил название Pentium Pro. Некоторые характеристики этого семейства представлены в табл. П 8.1 и П 8.2.

Таблица П 8.1.

Технические характеристики семейства процессоров Pentium Pro

Тактовая частота, МГц	150	166	180	200	200*
Быстродействие шины, МГц	60	66	60	66	-

КЭШ-память второго уровня, Кбайт	256	512	256	256	512
Технология, мкм	0,6	0,35	0,35	0,35	0,35
Энергопотребление, Вт					
стандартное	23,2	23,4	25,3	28,1	-
максимальное	29,2	29,4	31,7	35,0	-
Производительность**					
тест SPECint92	276,3	327,1	327,4	366,0	-
тест SPECfp92	220,0	261,3	254,6	283,2	-
Начало серийного производства	IV кв. 1995	I кв. 1996	IV кв. 1995	IV кв. 1995	II кв. 1996

* Технические характеристики процессора 200 Мгц/512 Кбайт пока отсутствуют.

** Тесты SPECint92 и SPECfp92 в настоящее время являются всемирно признанным стандартом для сравнения производительности вычислительных систем соответственно для целочисленной арифметики и операций с плавающей запятой при интенсивной работе 32-разрядных приложений. Показатели Pentium/133 по этим тестам — 190,9 и 120,6.

Источник: Intel

Таблица П 8.1.

Параметры вариантов КЭШ-памяти второго уровня

Объем, Кбайт	256	512
Число транзисторов, млн шт.	15,5	31,0
Технология, мкм	0,60	0,35

Источник: Intel

ЛИТЕРАТУРА

1. Шпаковский Г.И. Архитектура параллельных ЭВМ: Учеб. пособие для вузов. — Мн.: Университетское, 1989. — 192 с.
2. Хокни З., Джессхоуп К. Параллельные ЭВМ. — М.: Радио и связь, 1986. — 389 с.
3. Поспелов Д.А. Введение в теорию вычислительных систем. — М.: Сов.радио, 1972. — 280 с.
4. Бабаян Б.А., Бочаров А.В., Волин В.С. и др. Многопроцессорные ЭВМ и методы их проектирования. — М.: Высш. шк., 1990. — 143 с.
5. Головкин Б.А. Параллельные вычислительные системы. — М.: Наука, 1980. — 518 с.
6. Смирнов А.Д. Архитектура вычислительных систем: Учеб. пособие для вузов. — М.: Наука, 1990. — 320 с.
7. Прангишвили И.В., Виленкин С.Я., Медведев И.Л. Параллельные вычислительные системы с общим управлением. — М.: Энергоатомиздат, 1983. — 311 с.
8. Каляев А.В. Многопроцессорные системы с программируемой архитектурой. — М.: Радио и связь, 1984. — 283 с.
9. Векторизация программ: теория, методы, реализация. Сб. статей: Пер. с англ. и нем. — М.: Мир, 1991. — 275 с.
10. Вальковский В.А. Распараллеливание алгоритмов и программ. Структурный подход. — М.: Радио и связь, 1989. —176 с.
11. Тыугу Э.Х. Концептуальное программирование. — М.: Наука, 1984. — 255 с.
12. Шоу А. Логическое проектирование операционных систем: Пер. с англ. — М.: Мир, 1981. — 360 с.
13. Дейтел Г. Введение в операционные системы. — М.: Наука, 1984. — 255 с.
14. Буза М.К., Зимянин Л.Ф. Секционная модель параллельный вычислений // Программирование. 1990. № 4. С. 54-62.

15. Харп Г., Мэй Д., Уэйман Р., и др. Транспьютеры: архитектура и программное обеспечение / Пер. с англ. А.А. Агараняна. — М.: Радио и связь, 1993. — 303 с.

16. Бахтеяров С.Д., Дудников Е.Е., Евсеев М.Ю. Транспьютерная технология. — М.: Наука, 1990. — С. 3-93.

17. Краснов С.А. Транспьютеры, транспьютерные вычислительные системы и Оккам. В сб.: Вычислительные процессы и системы / Под ред. Г.И. Марчука. Вып.7. — М.: Наука, 1990. — С. 3-93.

18. Додонов А.Г., Кузнецова М.Г., Горбачик Е.С. Введение в теорию живучести вычислительных систем. — Киев: Наук. думка, 1990. — 184 с.

19. Коуги П.М. Архитектура конвейерных ЭВМ / Пер. с англ. — М.: Радио и связь, 1985. — 360 с.

20. Pentium processor User's Manual, v. 1, Intel Order Number 241428, 1993.

21. Головкин Б.А. Расчет характеристик и планирование параллельных вычислительных процессов. — М.: Радио и связь, 1983. — 272 с.

22. Шпаковский Г.И. Метод планирования трасс и архитектура ЭВМ со сверхдлинной командой // ЗРЭ. — 1991. — N 11. —С. 10-27.

23. Кузюрин Н.Н., Фрумкин М.А. Параллельные вычисления: теория и алгоритмы // Программирование. — 1991. — N 2. — С. 3-19.

24. Hardenberg H.L. Характеристики процессоров. — Dr. Dobbs Jr., January 1994.

ПРЕДМЕТНЫЙ УКАЗАТЕЛЬ

ОГЛАВЛЕНИЕ

Учебное издание

Шпаковский Геннадий Иванович

ОРГАНИЗАЦИЯ ПАРАЛЛЕЛЬНЫХ ЭВМ И СУПЕРСКАЛЯРНЫХ ПРОЦЕССОРОВ

Ответственный за выпуск *Г.А.Пушня*
Редактор *Е.В.Дмитриенко*
Художественное оформление *А.И.Кохнюка*
Технический редактор *А.А.Финский*

Подписано к печати __.__.__. Формат 60×84 1/16
Бумага тип. № __ Гарнитура Таймс Усл. печ. л. __,_ Уч.
-изд. л. __,__ Тираж 500 экз. Заказ № __.

Белгосуниверситет. 220050, Минск, пр-кт Ф.Скорины, 4.

Отпечатано на ризографе ИИВЦ Белгосуниверситета,
220050, Минск, пр-кт Ф.Скорины, 4.