

**БИБЛИОТЕЧКА
ПРОГРАММИСТА**

В. Ф. ЖИРОВ

Математическое обеспечение и проектирование структур ЭВМ



БИБЛИОТЕЧКА
ПРОГРАММИСТА

В. Ф. ЖИРОВ

МАТЕМАТИЧЕСКОЕ
ОБЕСПЕЧЕНИЕ
И ПРОЕКТИРОВАНИЕ
СТРУКТУР ЭВМ

Под редакцией
Л. Н. КОРОЛЕВА



МОСКВА «НАУКА»
ГЛАВНАЯ РЕДАКЦИЯ
ФИЗИКО-МАТЕМАТИЧЕСКОЙ ЛИТЕРАТУРЫ
1979

22.18

Ж 73

УДК 519.6

Математическое обеспечение и проектирование структур ЭВМ. Жиров В. Ф./Под ред. Королева Л. П.— Наука. Главная редакция физико-математической литературы, М., 1979.

В книге рассматриваются принципы интегрального проектирования модульного математического обеспечения. Анализируются основные операции обработки строк, символов, списков, деревьев с позиций возможной аппаратной реализации. Далее обсуждаются модель операционной системы и общие структурные методы увеличения производительности ЭВМ, приводятся примеры реализации методов в структуре мощных машин.

Ж $\frac{20204-083}{053(02)-79}$ 62-79. 1702070000

© Главная редакция
физико-математической
литературы
издательства «Наука», 1979

ОГЛАВЛЕНИЕ

Предисловие	5
Введение	7
Глава 1. Особенности типов машинных данных и операций над ними	14
1.1. Автокоды	14
1.1.1. Понятие о машинном языке	14
1.1.2. Принципы автокода	19
1.1.3. Методы трансляции с автокода	22
1.2. Макрокоманды	24
1.2.1. Специализированный макрогенератор	25
1.2.2. Универсальный макрогенератор	30
1.3. Операции обработки строк	42
1.4. Операции обработки списков	52
1.5. Операции обработки деревьев	59
1.6. Операции обработки массивов чисел	61
Глава 2. Операционные системы	69
2.1. Общие положения	69
2.2. Режимы эксплуатации мультипрограммных вычислительных систем	71
2.2.1. Пакетная обработка	71
2.2.2. Режим разделения времени	73
2.2.3. Режим реального масштаба времени	75
2.3. Программно-аппаратные средства операционных систем	77
2.3.1. Прерывание программ	77
2.3.2. Синхронизация процессов	79
2.3.3. Распределение и защита памяти	85
2.4. База данных	90
2.4.1. Сетевая модель данных	93
2.4.2. Принципы реляционной модели данных	102
2.4.3. Процессоры баз данных	107

Глава 3. Особенности структур мощных ЭВМ	111
3.1. Распределенная обработка	111
3.2. Совмещенная обработка команд	112
3.2.1. Совмещенное выполнение микроопераций	114
3.2.2. Заполнение и очистка К-структуры	117
3.2.3. Сохранение последовательности команд	119
3.3. Независимые функциональные блоки	120
3.3.1. Независимые модули оперативной памяти	120
3.3.2. Функциональные блоки арифметического устройства	121
3.4. Параллельные регистры команд	127
Заключение	146
Список сокращений	149
Литература	152

ПРЕДИСЛОВИЕ

Стремительные темпы развития вычислительной техники предъявляют исключительно высокие требования к уровню подготовки специалистов, связанных как с разработкой, так и с эксплуатацией систем обработки и хранения информации, создаваемых в нашей стране в соответствии с «Основными направлениями развития народного хозяйства в СССР на 1976—1980 годы».

В этих условиях приобретает решающее значение не столько накопление совокупности фактов, сколько приобретение навыков в применении основных методов проектирования ЭВМ, среди которых важное место занимают методы, опирающиеся на взаимосвязь математического обеспечения и структуры ЭВМ.

Настоящая книга написана на основе курса лекций, читавшегося студентам старших курсов, специализирующихся по математическому обеспечению ЭВМ. В основу выбора материала книги были положены следующие соображения. Прежде всего учитывалось, что лекционные курсы по математическому обеспечению и аппаратуре ЭВМ не всегда уделяют достаточное внимание многим актуальным вопросам. Поэтому в книге подробно рассматриваются операции над строками, списками, деревьями и массивами в контексте специализированных языков, при этом не уточняются аппаратные регистры для таких операций. Предполагается, что такое уточнение выходит за

рамки проблем связи программ и аппаратуры, для которых существенно, что делает операция, а не какие регистры для этого применяются. Далее подчеркивается, какие действия выполняются в операциях часто, так как именно они требуют аппаратной поддержки.

Общие методы построения структур конкретизируются примерами мощных ЭВМ, в которых предельно используются возможности электроники. Возрастающие требования прикладных программ заставляют искать способы дальнейшего увеличения производительности ЭВМ в автоматическом преобразовании программ пользователя в соответствии со структурой машины.

В книгу вошли также те вопросы взаимосвязи программ и аппаратуры, которые не получили должного освещения в технической литературе, что привело к определенной неоднородности материала: во всех необходимых случаях приводятся ссылки на литературу, которая может оказать помощь заинтересованному читателю.

Работа над книгой была бы невозможна без внимания и поддержки многих сотрудников ИТМ и ВТ АН СССР, которым автор приносит искреннюю благодарность.

В. Ф. Жиров

ВВЕДЕНИЕ

Современные системы обработки информации представляют собой сложные промышленные комплексы. Они состоят из большого количества ЭВМ, распределенных по значительной территории, объединенных каналами связи, имеющими огромную память и высокую скорость выполнения операций [41]. Одной из таких громадных вычислительных систем является сеть ARPA. Сеть ARPA [13] объединяет вычислительные центры нескольких университетов, научно-исследовательских институтов и фирм, расположенных на территории США, Англии, Европы и Гавайских островов. В нашей стране создается сеть вычислительных центров, в которую будут объединены несколько десятков тысяч ЭВМ.

Проектирование подобных систем опирается на ряд общих принципов, которые дают возможность на начальных этапах работы не учитывать многие детали. Одним из таких принципов служит единый интегральный подход к модульному проектированию системы, включающему как аппаратуру, так и математическое обеспечение (МО). Следствием такого подхода является применение стандартов на компоненты аппаратуры и МО.

По современным оценкам большую часть стоимости вычислительной системы [13] составляет стоимость МО, поэтому технология программирования МО является объектом исследования многих научных коллективов. Результатами этих исследований стали технические приемы проектирования программ. Одним из таких приемов [11, 14] является программирование *сверху — вниз*, при котором общее проектирование

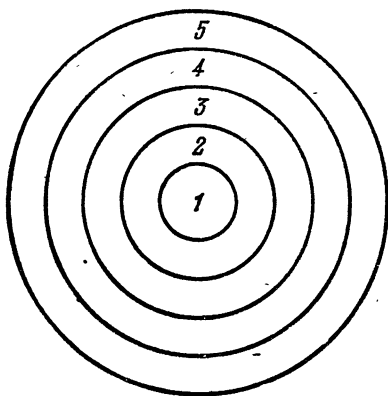


Рис. 1. Иерархия модулей вычислительной системы.

программы сводится к последовательности шагов, на каждом из которых модуль разбивается на вложенные более мелкие модули.

Модулям МО служат подпрограммы. В общем случае система специальным образом организованных подпрограмм и сейчас остается базисом МО [152]. Исторически развитие этого метода шло от многих подпрограмм, изобретаемых отдельными

пользователями, к библиотеке стандартных программ [58] и далее к языкам программирования.

Модули образуют определенные иерархические уровни, используемые при проектировании вычислительных систем. Конкретная иерархия зависит от выбранного критерия. Например, рассматривая характер использования средств обработки, можно вычислительную систему представить в виде концентрических слоев, как показано на рис. 1. Самым внутренним слоем служит аппаратура 1, затем следуют программы, определяемые спецификой аппаратуры, автокод 2 и операционные системы (ОС) 3, системы программирования 4 и пакеты прикладных программ (ПП) 5.

Конкретная иерархия, упрощающая представление системы, отвечает одному из критериев упорядочения абстракций.

Систему с хорошей архитектурой удастся полностью описать с помощью незначительного количества критериев.

Разработка программных модулей выполняется с применением следующих действий:

функционального разбиения, с заданием ожидаемого входа и требуемого выхода;

операционного объединения по данным одинаковой структуры, доступным только через обращение к модулю;

композиции конструкций по правилам структурированного программирования.

Последовательное применение метода сверху — вниз при проектировании модулей представляет реализацию определенных абстракций. Модули верхнего уровня по отношению к модулям более низкого уровня выступают как пользователи, которые, не зная внутренней организации подпрограмм, строят из них свои программы.

На первый план в декомпозиции системы на модули выдвигается описание межмодульных связей или интерфейса между модулями. Точное описание интерфейса, с одной стороны, дает возможность широко использовать модули в различных ситуациях, а, с другой стороны, допускает изменения внутренней организации модулей без модификации всей системы.

Необыкновенная гибкость программ приводит к непрерывной эволюции МО. Это объясняется и внешней простотой внесения изменений и, главным образом, возникновением новых знаний в процессе проектирования.

Декомпозиция системы на модули позволяет учесть эволюцию МО и, например, собрать в модули те места, в которых с большей вероятностью могут быть проведены изменения, и тем самым сократить область новых проверок при внесении изменений.

Методика разработки МО предусматривает несколько этапов, которые во многом совпадают с этапами разработки пользователем прикладных программ:

- 1) точная постановка проблемы,
- 2) выбор алгоритмов и выражение их в терминах структур данных и операций над структурами,
- 3) выбор языка программирования,
- 4) спецификация структуры программы,
- 5) кодирование,
- 6) отладка и тестирование,
- 7) переопределение предыдущих этапов по результатам отладки,
- 8) сопровождение.

Возврат к предыдущим этапам, указанный в п. 7, может в худшем случае иметь максимальный диапазон, когда потребуется изменить постановку задачи,

в лучшем случае — это исправление ошибок кодирования. Проектирование по своей природе — глубоко итеративный процесс, и возвраты принципиально необходимы. Даже для заключительного этапа — сопровождения — может потребоваться возврат к предыдущим этапам, как следствие технического старения или обнаружения ошибок.

Начальные этапы проектирования МО связаны с формулировкой недвусмысленных требований к программам, обычно в терминах ожидаемого входа и требуемого выхода. Степень формализации описания входа и выхода определяет точность спецификации МО и сокращает диапазон возвратов при обнаружении ошибок.

С этой целью функции модулей задаются в математических терминах и связываются со структурами данных. Данные приобретают тот смысл, который заложен в функциях, выполняемых над ними.

Одним из радикальных средств сокращения ошибок при декомпозиции программы служит введение ограничений на набор форм управления и широкое распространение подпрограмм. Совокупность средств такого рода предлагается методикой структурированного программирования, которое упрощает проектирование сложных программных систем.

Большие программы часто оказываются слишком сложными для понимания, и сложность рассматривается структурированным программированием как корневая проблема МО. Структурированное программирование предлагает простые способы уменьшения сложности:

- сокращение количества или полное исключение операторов безусловной передачи управления [14],
- широкое использование подпрограмм,
- обязательный комментарий,
- строгое оформление документации на всех этапах проектирования.

Примеры элементарных операторов управления структурированного программирования показаны на рис. 2. Выбор записи операторов определяется конкретным языком программирования. С этой точки зрения в развитии языков можно выделить два направления:

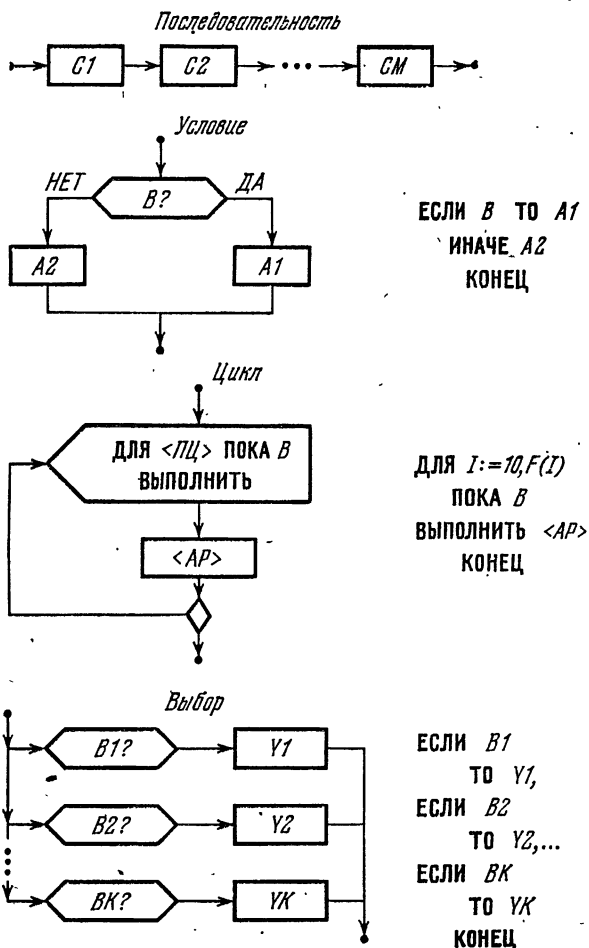


Рис. 2. Примеры элементарных управляющих конструкций структурированного программирования. ПЦ—параметр цикла, AP—тело цикла.

- 1) модификация традиционных языков,
- 2) разработка новых языков.

Структурированные программы в большинстве случаев состоят из модулей, имеющих один вход и один выход, поэтому запись таких программ становится легко читаемой и распечатка текста программ, подготовленная по определенным правилам, может быть принята за основу документации на систему.

Правила оформления документации таковы, что распечатка является полным, точным и легко читаемым документом [51].

Сложность современной вычислительной системы и процесс разделения труда привел к образованию специализированных групп разработчиков, занимающихся отдельными проблемами, изучение которых само по себе представляет ряд научных направлений.

В книге делается попытка конспективно объединить вопросы интегрального проектирования модульного МО и структур мощных ЭВМ и показать на отдельных фрагментах программ и структур необходимость концептуального единства проекта. При таком подходе на начальном этапе проектирования не фиксируется граница между программами и аппаратурой, программы описываются с позиций возможной аппаратной реализации, а аппаратура — с точки зрения предполагаемых программ.

Книга содержит три главы. Первая глава книги начинается с рассмотрения дескрипторов, образующих основу структуры современной ЭВМ [2]. Далее утверждается, что машинный язык занимает промежуточное положение в иерархии языков, он не может в точности совпадать с каким-либо языком высокого уровня и не может оставаться в рамках простых аппаратных микрокоманд.

Затем рассматривается символическая запись команд, которая представляет первый шаг по пути повышения уровня языка, что требует реализация дополнительных операций, основной из которых служит поиск по таблице символов.

Семантика машинных операций во многом определяется типами обрабатываемых данных. Операции над строками символов, списками, деревьями и массивами чисел в настоящее время представляют наибольший

интерес. Рассматриваемые специализированные языки, ориентированные на эти типы данных, построены на основе ограниченного набора функций. Аппаратная поддержка этих основных функций ускорит в целом выполнение программ обработки данных. Максимальное количество операций приходится выполнять при посимвольной обработке строк: одна операция на каждый символ; более эффективна обработка строк как деревьев.

Во второй главе кратко рассматриваются операционные системы. Требования к операционным системам (ОС) во многом зависят от режима эксплуатации мощной многопрограммной вычислительной системы. ОС решает задачу динамического согласования запросов многих программ и имеющихся ресурсов. Сложность получения эффективного решения сочетается с определенной простотой взаимодействия слабо связанных процессов. Ввиду большого объема публикаций по ОС читателю, интересующемуся деталями, следует обратиться к литературе, приведенной в конце книги.

В третьей главе анализируется взаимодействие МО и структуры с позиций возможностей аппаратуры. Основу анализа составляет конвейерная структура (К-структура). Характеристиками К-структуры являются: постоянная пропускная способность и замкнутость уровней. Замкнутость определяет тот набор уровней, на котором реализуется совмещенная обработка команд. Постоянная пропускная способность поддерживается параллельной работой функциональных блоков. Уровень регистра команд играет особую роль в К-структуре, так как он запоминает последовательность команд при параллельной работе аппаратуры.

В заключительных разделах книги анализируется максимальный уровень быстродействия, который может быть достигнут в настоящее время в рамках существующих структур.

ГЛАВА 1

ОСОБЕННОСТИ ТИПОВ МАШИННЫХ ДАННЫХ И ОПЕРАЦИЙ НАД НИМИ

1.1. Автокоды

1.1.1. Понятие о машинном языке. Основное назначение машинного языка состоит в описании операций над регистрами и ячейками памяти. Как содержимое, так и адреса регистров и ячеек памяти служат операндами в операторах машинного языка.

Среди типичных операторов машинного языка современной вычислительной системы важное место занимает группа операторов, производящих арифметические действия. Оператор такой группы содержит информацию о требуемой арифметической операции, о местах расположения операндов и результата. Операнды представляются числами, которые могут быть заданы непосредственно в команде, либо в регистре или ячейке памяти. Количество адресов, записанных в одной команде, определяет адресность команды. Практическое применение нашли команды в диапазоне от четырех адресов до безадресных команд. Безадресные команды предполагают, что места операндов и результата заранее подразумеваются, поэтому в команде достаточно иметь только код операции, но предварительно команды с адресами должны загрузить те регистры и ячейки памяти, над которыми будет выполняться безадресная операция.

Так как в настоящее время большая часть машин, производимых в мире, поставляется фирмой ИБМ (IBM — International Business Machinery), то широкое

распространение получила система команд, в которой для большинства команд указывается адрес регистра первого операнда, адрес ячейки памяти или адрес регистра второго операнда, регистр первого операнда предполагается местом результата. В этой системе команд самая короткая команда использует адреса регистров для первого и второго операндов. Так как в начальный момент исполнения программы вся информация располагается в ячейках памяти, то для операций над регистрами необходимо предварительно их загрузить. Время, потраченное на команды загрузки, будет оправдано, если затем над регистрами будет выполнено несколько операций без обращений к памяти.

Выбор формы записи операторов машинного языка во многом определяется краткостью записи алгоритмов или информативностью командного бита. Например, исследования, проводимые фирмой IBM для IBM/360, показали, что для широкого класса задач система команд IBM/360 обладает, в среднем, наилучшей информативностью [3].

Систему команд можно считать формальным описанием структуры процессора. Основным элементом структуры служит регистр команд, на котором дешифруются команды, вызываемые из памяти. По сигналам регистра команд запускаются блоки процессора, непосредственно выполняющие команду.

Системы команд разных типов машин существенно различны, но элементарные *составляющие* операции во многих машинах похожи. Отличия в них связаны чаще всего с требованиями скорости выполнения операции, приводящими к разному числу регистров в структуре. Поэтому система команд не является точным описанием структуры. Одну и ту же систему команд могут иметь процессоры разного быстродействия.

В дальнейшем целесообразно рассматривать элементарные операции, выполняемые на регистрах, независимо то того, в какой форме представлен код этих регистров в системе команд.

Аппаратуру процессора можно рассматривать как набор регистров, над которыми выполняются операции. Поэтому операции загрузки и разгрузки регистров яв-

ляются основными элементарными операциями. В этих операциях осуществляется пересылка информации из сравнительно медленной оперативной памяти в более быструю память на регистрах. Сложные операции базируются на этих операциях, например, операция сложения заключается в выборке двух операндов и загрузке их в регистры, затем содержимое регистров передается через аппаратуру суммирования, и результат записывается в регистр. Аппаратура суммирования реализует одну из логических функций, все множество которых задается машинным языком. Таким образом, структуру процессора можно рассматривать так же, как набор логических функций, представленных связями между регистрами.

В операциях обмена информацией между памятью и регистрами указывается как номер регистра, так и адрес ячейки памяти. Адрес ячейки памяти формируется как номер элемента вектора, входящего в совокупность векторов, описывающих память [2]. Каждому вектору ставится в соответствие ячейка памяти, содержащая необходимую информацию для обращения к вектору и называемая *дескриптором*. В свою очередь совокупность подряд расположенных дескрипторов можно считать вектором, которому поставлен в соответствие свой дескриптор. Дескриптор — это описатель вектора, который содержит следующую информацию:

- 1) адрес начала вектора в памяти,
- 2) тип элемента вектора,
- 3) длина вектора или количество элементов вектора,
- 4) дополнительные признаки.

Для формирования адреса элемента вектора в операциях считывания и записи информации дескриптор используется следующим образом:

$$A = \text{ТФ}(\text{БАЗА} + \text{ИНДЕКС} + \text{СМЕЩЕНИЕ}), \\ \text{ИНДЕКС} + \text{СМЕЩЕНИЕ} \leq \text{ДЛИНА}$$

где A — адрес элемента вектора в памяти, БАЗА — адрес начала вектора в памяти, ИНДЕКС — относительный номер элемента вектора, СМЕЩЕНИЕ — адрес, размещаемый непосредственно в операции, ДЛИНА — длина вектора, ТФ — функция согласования

размеров элементов вектора и ячейки памяти; например, если размер ячейки памяти равен 64 разрядам, а размер элемента вектора равен байту, то ТФ представляет целочисленное деление на 8.

Учитывая многократность обращений к вектору, БАЗУ, ИНДЕКС и ДЛИНУ целесообразно разместить на соответствующих регистрах процессора, а СМЕЩЕНИЕ как адрес в операции находится в момент выполнения обращения на регистре команд. Если сумма ИНДЕКСА и СМЕЩЕНИЯ больше ДЛИНЫ вектора, то обращение к памяти блокируется и вызывается программа обработки особой ситуации. Так как размер ячейки памяти заранее известен, то функция ТФ формируется аппаратно по полю ТИПА в дескрипторе. Практически вся информация для дескрипторов нескольких векторов помещается на регистрах в мощных процессорах.

Одним из стандартных дескрипторов служит дескриптор вектора команд. Дескриптор команд содержит следующую информацию:

1) база программы, образуемой этим вектором команд;

2) тип команд;

3) длина программы;

4) признаки.

При обращении к команде по дескриптору возможны два режима: последовательная выборка команд и передача управления. ИНДЕКСОМ для команд служит счетчик команд; при последовательной выборке команд СМЕЩЕНИЕ равно нулю. В начальный момент выполнения программы на счетчик команд передается СМЕЩЕНИЕ для первой выполняемой команды, и по этому адресу выполняется обращение к памяти. Из памяти команда выбирается на регистр команд и выполняется, т. е. интерпретируется, аппаратно. Далее, если команда не была командой передачи управления, то к содержимому счетчика команд добавляется длина выполненной команды, и выбирается следующая команда. Если команда была командой передачи управления, то на счетчик команд передается СМЕЩЕНИЕ из поля команды, по которому выбирается новая команда внутри того же командного вектора.

Передача управления может предполагать вызов другой программы, т. е. переход на другой вектор команд и смену дескриптора. Так как программы однозначно идентифицируются своими именами, то используется вектор символических имен тех программ, которые могут быть вызваны данной программой. Этому вектору соответствует дескриптор, называемый дескриптором области связи. Передача управления со сменой дескриптора выполняется как обращение к элементу области связи.

Это обращение может иметь две формы: первая форма, соответствующая самому первому обращению к программе, называется *первым знакомством*, вторую форму имеют все последующие обращения. При первом знакомстве элемент вектора области связи содержит символическое имя и признак прерывания. По признаку прерывания происходит обращение к операционной системе, которая отыскивает вызываемую программу по символическому имени и выделяет для этой программы место в памяти, заменяя символическое имя на дескриптор вызываемой программы. Этот дескриптор загружается в соответствующий программе элемент вектора области связи, в котором содержится следующая информация:

- 1) командный дескриптор вызываемой программы,
- 2) дескриптор области связи вызываемой программы,

3) **СМЕЩЕНИЕ** для первой команды вызываемой программы.

По возврату из прерывания снова повторяется команда передачи управления, по которой происходит перезагрузка дескрипторов команд и области связи для вызываемой программы. Если после выполнения второй программы снова будет вызвана первая программа, в которой произойдет повторный вызов второй программы, то теперь уже прерывания не будет.

Другим стандартным типом дескриптора можно считать тип данных. В конкретных ситуациях тип данных дополнительно уточняется, например, вещественный, целый, строковый и т. д. Если при обращении к элементу вектора данных уже выполнена процедура первого знакомства, то дескриптор содержит следующую информацию:

- 1) адрес начала вектора данных,
- 2) тип,
- 3) длина вектора данных,
- 4) признаки.

При обращении к элементу данных адрес формируется так же, как для команд. Если необходимо перейти к новому вектору данных, то сначала происходит обращение к элементу вектора области связи.

По аналогии с использованием номеров регистров, в которых располагаются операнды и результат, в командах могут использоваться номера регистров, в которых расположены соответствующие дескрипторы, тогда загрузка этих дескрипторов может быть выполнена по некоторому дескриптору, описывающему вектор дескрипторов.

Исторически машинный язык использовался в начальный период разработки МО для первичных программ ввода и вывода. Затем на машинном языке писался транслятор с подмножества автокода, на котором производилась реализация основных блоков ОС, затем снова расширялся автокод и т. д. В настоящее время при создании автокода новой машины используются средства МО уже работающей машины, на которой строится и программная модель выполнения команд новой машины, называемая интерпретатором. Такой подход дает возможность вести отладку МО проектируемой машины на существующей машине, называемой инструментальной. Интерпретатор новой машины используется при проведении начальной отладки модулей нового МО, комплексная отладка затруднена обычно тем, что мощность новой машины значительно больше мощности старой, но проведение отладки МО Иллиак на B6500 говорит о безусловной оправданности такого подхода [125].

1.1.2. Принципы автокода. Под автокодом понимается символическая запись операторов машинного языка. Основное преимущество символической записи — это легкость внесения изменений, так как установление соответствия между символической записью и числовой является задачей ассемблера, транслятора текста на автокоде. Автокод при максимальной эффективности может обладать и достаточными удобствами.

Именно на автокоде написаны основные коммерческие ОС и трансляторы с языков высокого уровня. Положительными характеристиками автокода коммерческого применения считаются следующие:

- многообразие форм данных (двоничные, восьмеричные, десятичные, шестнадцатеричные, символические и битовые);

- гибкость распределения памяти;

- универсальность эквивалентных представлений символов;

- возможность использования комментариев;

- полнота диагностики и средств отладки;

- сегментация программ;

- простота использования операторов расширения языка;

- удобства и скорость связи с общим системным архивом.

При разработке МО новой машины можно воспользоваться методом метаавтокода [5], который объединяет в себе некоторые общие возможности автокодов, и, воспринимая описание конкретного машинного языка и правила конкретного автокода, функционирует как автокод новой машины. Одно из существенных ограничений такого автокода — это фиксированный входной синтаксис. Метаавтокод во многом напоминает макросы, описываемые ниже. В тех случаях, когда система команд новой машины продолжает модифицироваться, метаавтокод дает удобные средства обработки системы команд.

Главный принцип автокода — трансляция «один в один», т. е. перевод каждого символического объекта в соответствующий объект машинного языка, — справедлив только для ядра автокода. Основная тенденция развития автокода состоит в расширении его средствами языка высокого уровня, причем введение новых автокодных операторов, интерпретируемых транслятором, сопровождается частичной потерей эффективности.

Первым шагом в повышении уровня автокода можно считать наличие *псевдокоманд*, для которых нет соответствующих машинных команд. Синтаксис псевдокоманд обычно полностью аналогичен синтаксису операторов ядра автокода. Псевдокоманды, называемые

командами управления транслятором, сообщают об эквивалентности символических адресов, указывают идентификаторы, общие для подпрограмм, управляют распределением памяти, расширяют систему команд новыми командами. При программировании на автокоде пользователь сам управляет распределением памяти, и группа псевдокоманд требует дополнительного внимания.

В автокод могут быть постепенно введены условные операторы, операторы цикла, функции, подпрограммы, подобные операторам языка высокого уровня, что неминуемо изменяет синтаксис автокода. Например, Фортран возник как постепенное расширение автокода IBM 704, называемое программой символического ассемблирования — SAP (Symbolic Assembly Program), которая была заменена программой формальной трансляции и ассемблирования — FAP (Fortran Assembly Program).

Повышение уровня автокода опирается на опыт использования языков высокого уровня, таких, как алгол, кобол, ПЛ/1, и непосредственно связано с модификацией структуры машины в направлении введения аппаратуры входа в подпрограммы, регистров дескрипторов, тегов, функциональных блоков операций и т. п. Такое расширение приводит к усложнению структуры машины, и в общем к увеличению времени выполнения отдельных операций. Но, так как такие операции заменяют целые подпрограммы, ранее строившиеся из элементарных операций, то время выполнения одной сложной операции оказывается меньше суммарного времени выполнения элементарных операций. Повышение уровня автокода связано также с соответствующим преобразованием МО, с тем чтобы в полной мере воспользоваться возможностями аппаратуры. Например, введение дескрипторов с типами параллельности требует соответствующих подпрограмм ОС.

Увеличение требований к надежности программ также требует учета непосредственно на уровне автокода. Аппаратура проверяет соответствие ДЛИНЫ в дескрипторе ИНДЕКСУ и согласование ТИПОВ как в дескрипторе, так и в теге. Эти действия осуществляются в момент выполнения программы и требуют от

программиста анализа ситуаций конкретного шага счета. Целесообразно перенести большую часть проверок на этап трансляции, расширив язык конкретными ТИПАМИ. Такого рода методы предложены в языках: паскаль, эвклид, альфард, клу, модула [11]. Осуществление проверок на этапе трансляции выполняется за счет получения от программиста избыточной информации, которая сжимается транслятором и в минимальном виде анализируется во время счета.

1.1.3. Методы трансляции с автокода. Основной задачей трансляции является распределение ячеек памяти для объектов программы с заменой символических адресов объектов на адреса ячеек памяти. Трансляторы с автокода работают по двухпроходной и однопроходной схемам. Так как двухпроходная схема несколько проще, то она рассматривается в первую очередь.

Предположим для простоты, что оператор автокода имеет следующий вид:

⟨метка⟩ ⟨код операции⟩

⟨символический адрес⟩.

Далее пусть автокодный оператор переводится в одну ячейку памяти, и существует счетчик операторов. Первому оператору соответствует нулевое значение счетчика. Трансляция начинается с просмотра текста слева направо. При первом просмотре текста значение счетчика увеличивается на единицу при переходе от одного оператора к другому. Если оператор автокода начинается с метки, то в таблицу, называемую таблицей символов, помещаются метка и текущее значение счетчика. При втором просмотре выполняется полная замена символической записи на двоичную или генерация объектного кода. Символический код операции заменяется с помощью таблицы операций на код операции. Символический адрес сравнивается с метками, хранящимися в таблице символов, и в случаях совпадения заменяется на значение счетчика, соответствующее этой метке, при несовпадении выдается сообщение об ошибке: «неопределенный адрес».

В однопроходной схеме трансляции все действия должны быть выполнены за один просмотр текста.

В тех ситуациях, в которых символический адрес встречается раньше, чем соответствующая ему метка, необходима специальная техника. В момент появления такого символического адреса его нельзя заменить на значение счетчика, поэтому он помещается в строку T1 таблицы символов с адресом того места программы, которое должно быть занято значением счетчика — П1. Если в дальнейшем просмотре текста опять встретится тот же символический адрес, то в текущую строку таблицы символов T2 помещается адрес П2 того места программы, которое должно быть занято значением счетчика, и кроме того, адрес T1 строки таблицы символов, в которой находится этот же символический адрес. При повторении ситуации в текущую строку таблицы символов помещается адрес П3 программы и адрес T2 и т. д., до тех пор, пока не встретится метка, совпадающая с этим символическим адресом. Тогда используются ссылки T_i для получения всех адресов $П_j$, по которым подставляется в текст программы значение счетчика, соответствующее метке.

Если длина команды определяется кодом операции, то программу не придется модифицировать, в противном случае, когда длина команды определяется типом оператора автокода, имеющим метку, необходима перекомпоновка программы.

В рассмотренных схемах широко используется операция поиска по таблице символов, от времени поиска зависит время трансляции. Общепринятым способом ускорения поиска служит введение функции расстановки (перемешивания) [56]. Функция расстановки применяется к символическому адресу или к метке как к числу, и в результате получается адрес строки таблицы символов. Известны и исследованы различные функции расстановки, поэтому аппаратная реализация таких функций может быть выбрана с учетом как программных требований, так и возможностей аппаратуры. Обычные методы быстрого деления на константу дают удовлетворительные результаты [64]. В большинстве практических случаев достаточно использовать операцию сложения по mod 2. Конкретные примеры использования функций расстановки приведены в нижеследующих разделах.

1.2. Макрокоманды

Основным способом повышения уровня автокода служит введение макрокоманд. Макрокоманда в простейшем случае — это *большая команда*, определение которой вводится программистом и имя которой размещается в поле кода операции автокода. Определение, называемое макроопределением, часто представляет группу команд. Процесс замены макроимени на соответствующее макроопределение называют макрогенерацией.

Макрокоманды и макроопределения объединены одним понятием — *макроста*. Макросы отличаются от подпрограмм тем, что все макрокоманды заменяются на макроопределения с соответствующими модификациями до выполнения программы, тогда как подпрограмма заменяет команду вызова во время выполнения. При использовании макросов экономится время вызова подпрограмм, но проигрывается память.

Общее для подпрограмм и макросов то, что решается задача многократного выполнения одних и тех же действий без переписывания программистом одних и тех же частей текста.

Первичная цель использования макросов — сократить длину текста. Но в результате текст выполняющейся программы отличается от того текста, который написал программист, в то время как диагностические сообщения относятся к выполняемому тексту, что может существенно усложнить отладку.

Исторически подпрограммы делились на замкнутые и открытые. Открытые подпрограммы заменяют вызов подпрограммы на тело во время трансляции, а не в момент выполнения, как замкнутые. Макросы можно рассматривать как дальнейшее развитие открытых подпрограмм.

Программу преобразования входного текста, содержащего макросы, называют макрогенератором. Результат работы макрогенератора есть макрорасширение. Выходной текст макрогенератора, таким образом, состоит из непреобразованных частей входного текста и макрорасширений. Если выходной текст макрогенератора рассматривается как входной для некоторого транслятора, то синтаксис выходного текста определя-

ется этим транслятором. Макрогенератор, ограниченный синтаксисом выходного текста, называется специализированным.

1.2.1. Специализированный макрогенератор. Известным примером специализированного макрогенератора является макроассемблер [96]. Макроассемблер — это макрогенератор, специализированный под синтаксис автокода или языка ассемблера. Входной текст макрогенератора состоит из последовательности команд, соответствующих автокоду, и макрокоманд. Каждая строка текста имеет вид, обычный для автокода:

⟨метка⟩ ⟨операция⟩ ⟨операнды⟩.

В поле операции помещается либо символический код автокода, в частности код машинной операции, либо имя макрокоманды. В дальнейшем используются средства макроассемблера ИБМ Системы 360, за некоторыми исключениями: вместо амперсанда для символа параметра используется процент, вместо команд Системы 360 используется одноадресная система команд.

Макроопределение во входном тексте ограничивается специальными скобками: **MACRO** и **MEND**.

MACRO

⟨тело макроопределения⟩

MEND

Строка после слова **MACRO** называется *прототипом* и содержит имя макроса и параметры, следующие строки макроопределения называются *модельными* предложениями. Основное назначение модельных предложений — заменить макрокоманду во входном тексте. Перед каждым параметром в модельном предложении ставится символ параметра — процент (%). В примерах используется одноадресная система команд.

Если необходимо записать сложение двух чисел в виде одной строки, то можно ввести следующее макроопределение:

MACRO	}	макроопределение
СУММА %X, %Y, %Z		
СЧИТАТЬ %X		
СЛОЖИТЬ %Y		
ЗАПИСАТЬ %Z		
MEND		
СУММА A, B, C — макрокоманда.		

Подстановка макроопределения создает макрорасширение:

СЧИТАТЬ A
СЛОЖИТЬ B
ЗАПИСАТЬ C

В этом примере соответствие между формальными и фактическими параметрами устанавливается позиционно, слева направо по порядку.

Параметры могут располагаться в любом из трех полей: метки, операции и операндов, например:

MACRO	}	макро- определе- ние		
ФУНКЦИЯ %X, %Ф, %М, %Y				
СЧИТАТЬ %X				
%М %Ф %Y				
СЛОЖИТЬ %X				
УСЛОВНЫЙ-ПЕРЕХОД %М				
ЗАПИСАТЬ %X				
MEND				
ФУНКЦИЯ A1, УМНОЖИТЬ,			}	макро- команда
ЦИКЛ, A2,				

макрорасширение —

СЧИТАТЬ A1
ЦИКЛ УМНОЖИТЬ A2
СЛОЖИТЬ A1
УСЛОВНЫЙ-ПЕРЕХОД ЦИКЛ
ЗАПИСАТЬ A1

Кроме позиционной передачи параметров, существует передача параметров по ключу. При передаче

параметров по ключу каждый параметр записывается в прототипе в виде пары, разделенной знаком равенства:

⟨имя параметра⟩ =
= ⟨начальное значение параметра⟩

В макрокоманде с передачей параметров по ключу можно указывать не все параметры, а только те, которые в данном случае имеют значения, отличающиеся от начальных. Например, если входной текст:

```
MACRO
СДВИГ    %ОПЕРАНД = А,
                                     %КОЛИЧ = 1,
                                     %ЦИКЛ = М1,
                                     %РЕЗУЛЬТ = В
СЧИТАТЬ %ОПЕРАНД
%ЦИКЛ СД    %КОЛИЧ
. УСЛОВНЫЙ-ПЕРЕХОД %ЦИКЛ
ЗАПИСАТЬ    %РЕЗУЛЬТ
MEND
СДВИГ      РЕЗУЛЬТ = С,
                                     КОЛИЧ = 2
```

то макрорасширение:

```
СЧИТАТЬ А
М1 СД      2
УСЛОВНЫЙ-ПЕРЕХОД М1
ЗАПИСАТЬ С
```

Макроассемблер имеет большой набор встроенных системных функций и переменных, используемых только в период макрогенерации и не передаваемых в выходной текст. Системные функции присваивания — SET, безусловной передачи управления — AGO, условной передачи управления — AIF обеспечивают необходимый минимум сервисных средств для наиболее часто используемых случаев. Присваивание учитывает описание типа переменных, таких как:

SETA — арифметический,
 SETB — булевский,
 SETC — символьный.

Переменные указанных типов описываются как локальные и глобальные:

1) локальные

LCLA — арифметического типа,
 LCLB — булевского типа,
 LCLC — символьного типа.

2) глобальные

GBLA — арифметического типа,
 GBLB — булевского типа,
 GBLC — символьного типа.

Локальные переменные существуют только во время обработки того макроопределения, в котором они описаны, и их начальные значения — нули. В примере суммирования чисел, кроме системных операторов и переменных периода макрогенерации, используются и метки периода макрогенерации в виде набора символов, начинающегося с точки:

	MACRO		} макроопределение
	СУММА	% X, % Y	
	LCLA	% СЧЕТ	
	LCLB	% УСЛОВ	
МЕТКА	% СЧЕТ SETA	% СЧЕТ + 1	
	% УСЛОВ SETB	(% СЧЕТ GT	
		% X)	
	AIF	% УСЛОВ. КОНЕЦ	
	СЛОЖИТЬ	% Y % СЧЕТ	
	AGO .МЕТКА		
	.КОНЕЦ ANOP		
	MEND		
	СУММА 10, A —	макрокоманда	

В этом примере используются: описания локальных переменных — LCLA, LCLB; операторы присваивания — SETA, SETB, для операций сложения $\%СЧЕТ + 1$ и отношения $\%СЧЕТ > \%X$; оператор условной передачи управления — AIF на КОНЕЦ, если $\%УСЛОВ$ — истина; оператор безусловной передачи управления — AGO на МЕТКА; пустой оператор — ANOP. Если макрокоманда:

СУММА 10, A

то макрорасширение

СЛОЖИТЬ A1

СЛОЖИТЬ A2

СЛОЖИТЬ A10

В макроассемблере могут использоваться следующие операции отношения:

EQ =

NE \neq

LT $<$

LE \leq

GT $>$

GE \geq

и логические операции над булевыми переменными:

AND \wedge

OR \vee

NOT \neg

Отдельную группу системных функций образуют атрибуты. Атрибут есть функция, которая применяется к формальному параметру. Значением функции служат тип параметра, количество символов в параметре, размер области памяти и т. д. Например, существуют следующие атрибуты:

T — тип данных,

K — количество символов,

I — количество десятичных цифр.

L — размер области памяти,

Аргумент функции отделяется апострофом, например: T' % X. Цель введения атрибутов — увеличить степень изменяемости текста относительно той, которую обеспечивает механизм передачи параметров, эти функции могут быть полезными при написании программ трансляции. Например, T' % X может иметь следующие значения:

B — двоичный тип,

C — символьный тип,

D — тип «длинного» числа с плавающей запятой и т. д.

Возможности специализированного макрогенератора широко используются в операционной системе ИБМ 360. Операционная система поддерживает общую библиотеку системных макроопределений, доступную всем пользователям. Компонентами библиотек являются средства связи между программами пользователя и системой, подсистема ввода/вывода, управление файлами и т. д. Существенное упрощение работы пользователя с файлами достигается за счет применения правил умолчания, базирующихся на передаче параметров по ключу.

Реализация макроассемблера строится на основе двухпроходной схемы трансляции с использованием стека, в котором запоминается текущее состояние программы при каждом макровывозе (макрокоманде) [19].

Макроассемблер представляет неоднородное объединение различных программных средств. Поэтому при отображении этих средств на структуру ЭВМ возникают проблемы, общие для больших программ, а именно проблемы аппаратной поддержки связей между подпрограммами.

1.2.2. Универсальный макрогенератор. Известным примером универсального макрогенератора (УМГ) служит GPM (General Purpose Macrogenerator) [140]. УМГ Стречи обладает почти минимальным набором управляющих символов. Основа УМГ — это строка, которая помещается между § и ;. Такая строка пред-

ставляет макровывоз. Вложенные макровывозы строятся в виде

§...§...; §...; ...;

Если после § стоит DEF, то строка является макроопределением, и вызывается системная подпрограмма DEF, которая загружает текст, расположенный между DEF и ;, в таблицу макроопределений (ТМ). Таким образом, макроопределение имеет вид

§ DEF, <имя>, <тело>;

В макровывозе после § запятые отделяют фактические параметры, и DEF считается нулевым параметром. Количество фактических параметров может быть больше количества формальных параметров.

Формальные параметры записываются в теле в виде

$\sim i$

где \sim признак формального параметра, i — номер параметра.

Фактические параметры пронумерованы в макровывозе слева направо, по запятым. Если после § не стоит имя системной подпрограммы, то такое имя должно быть ранее помещено в таблицу макроопределений с помощью DEF. Общий вид макровывоза следующий:

§ ИМЯ, P_1 , P_2 , ..., P_N

где ИМЯ — имя макроса, P_i — параметр, $i = (1 \div N)$.

Замена макровывоза на макроопределение называется вычислением, результатом которого служит макрорасширение. Для вычисления макровывоза сначала необходимо вычислить все фактические параметры. При вычислении макрогенератор просматривает текст слева направо.

В УМГ символ § требует вычисления символического выражения; когда необходимо отменить действие § или любого служебного символа, то применяются угловые скобки <.>. Если тело макроопределения заключено в угловые скобки, то при загрузке в таблицу макроопределений одна пара самых внеш-

них угловых скобок снимается. Угловые скобки регулируют последовательность вычислений.

Вычисление макровывоза начинается с вычисления значений всех фактических параметров.

Пример 1.

§ DEF, PI, 3.14159265359; — макроопределение,
§ PI; — макровывоз,
3.14159265359 — макрорасширение.

Пример 2.

§ DEF, C, ~2~1~3~1; — макроопределение,
§ C, A, П, Р; — макровывоз,
ПАРА — макрорасширение.

Пример 3.

§ DEF, A, (§ C, A, П, Р; ШЮТ); — макроопределение,
§ A; — макровывоз,
ПАРАШЮТ — макрорасширение.

Пример 4.

МОЙ (§) A (<); — входной текст,
МОЙ § A; — первый этап макрорасширения,
МОЙ ПАРАШЮТ — второй этап макрорасширения.

Необходимым условием записи произвольного алгоритма является наличие операторов условной передачи управления. В УМГ условные операторы могут быть реализованы с помощью макроопределений внутри макровывозов, причем имя внутреннего макроопределения должно при одних значениях фактических параметров совпадать с ранее введенным именем, а при других — нет. Если имена совпадают, то, так как таблица макроопределений имеет стековую (магазинную) организацию, то в вычислениях используется тело, введенное внутренним макроопределением, в противном случае выбирается внешнее макроопределение.

Далее рассматривается пример получения следующей цифры по любой заданной, т. е. $n + 1$ для n , где n — десятичная цифра [137]:

\S DEF, SUC, $\langle \S$ 1, 2, 3, 4, 5, 6, 7, 8, 9, 0 \S DEF,
 1, $\langle \sim \rangle \sim 1$; \rangle ; — макроопределение,
 \S SUC, 3; — макровывоз,
 4 — макрорасширение.

Действительно, подставив в макроопределение значение фактического параметра вместо формального, имеем:

\S 1, 2, 3, 4, 5, 6, 7, 8, 9, 0 \S DEF, 1, $\langle \sim \rangle 3$; ;

второй шаг макрорасширения —

\S 1, 2, 3, 4, 5, 6, 7, 8, 9, 0 \S DEF, 1, ~ 3 ; ;

затем вычисляется значение фактического параметра \S DEF, 1, ~ 3 ; для макровывоза \S 1, 2, ...; , в результате которого в таблице макроопределений появляется строка: 1 есть ~ 3 . Окончательно результат вычисления макровывоза \S 1, 2, ...; равен 4.

Далее пусть необходимо найти $m + 1$ для m , где m — двузначное число. Решение задачи состоит в сравнении цифры младшего разряда числа с 9. Если цифры совпадают, то вместо цифры младшего разряда берется 0, а цифра старшего разряда подставляется как фактический параметр в макросе SUC; если цифра младшего разряда не совпадает с 9, то старшая цифра остается без изменений, а младшая цифра становится фактическим параметром макроса SUC.

Таким образом, макроопределение можно записать следующим образом:

\S DEF, SUCCESOR, $\langle \S \sim 2, \S$ DEF, $\sim 2, \sim 1$
 \langle, \S SUC, $\rangle \sim 2 \langle ; \rangle$;
 \S DEF, 9, $\langle \S$ SUC, $\rangle \sim 1 \langle ;, 0 \rangle ; ; \rangle$;

а макровывоз:

\S SUCCESOR, M1, M2;

где M1, M2 — цифры двухзначного числа. Сначала подставляются фактические параметры в макроопределение:

\S M2, \S DEF, M2, M1 \langle, \S SUC, \rangle M2 $\langle ; \rangle$;
 \S DEF, 9, $\langle \S$ SUC, \rangle M1 $\langle ;, 0 \rangle ; ;$

При вычислении фактических параметров заполняется таблица макроопределений:

$$\begin{array}{l} 1. M2 - \text{пмя} \mid M1. \text{\$SUC. } M2; \\ 2. 9 - \text{пмя} \mid \text{\$SUC, } M1; \quad , 0 \end{array}$$

Если $M2 \neq 9$, то результат следующий:

$\text{\$ SUC, } M1; \quad , 0$ и затем $M1 + 1, 0$ если

$M2 = 9$, то получается:

$M1, \text{\$ SUC, } M2;$ и далее $M1, M2 + 1$

Принцип реализации УМГ состоит в применении единственного стека, ячейки которого связываются в списки разных типов. УМГ сохраняет в стеке все промежуточные результаты, а на выход выдается окончательно сформированная строка.

В процессе обработки входного текста УМГ проматривают символы слева направо, и если не встречает предупреждающих символов

$| < | > | \$ | , | ; | \sim |$

то передает их на выход. Если же встречаются предупреждающие символы, то они вместе с другими символами загружаются в стек. Далее выполняется обработка стека и входной строки так, что, по мере завершения частичных вычислений подстрок, эти подстроки выбираются из стека на выход. Таблица макроопределений также формируется в стеке в виде ячеек стека, объединяемых одним списком, который называется Е-списком.

Режим сканирования УМГ входной строки определяется счетчиком угловых скобок Q . В начальный момент $Q = 1$, затем для каждой открывающей угловой скобки Q увеличивается, а для закрывающей — уменьшается. Когда $Q = 2$, то все входные символы как бы заключены в скобки, и предупреждающие символы не опознаются. При всех увеличениях Q от 2 и выше выполняется копирование входных строк на выход. Если $Q = 1$ и встретилась открывающая скобка, то Q присваивается значение 2 без копирования этого символа, таким образом осуществляется снятие одного слоя скобок. В общем случае источником сим-

волов для УМГ может служить либо входной поток, либо стек, для разделения которых используется служебная ячейка S . Если $S \neq 0$, то в ней находится адрес ячейки стека, из которой должен быть взят символ; если $S = 0$, то сканируется входной поток.

Если при сканировании встречается обычный непредупреждающий символ, то пункт выдачи его указывается в служебной ячейке H : если $H \neq 0$, то — в стек, если $H = 0$, то — на выход.

Главный предупреждающий символ \S сообщает о начале макровывоза, но переход на замену макроса не осуществляется до появления ; для этого \S . Так как ; может быть отделена от своего \S произвольным количеством вложенных макровывозов, то вводится F -список.

Ячейки F -списка формируются при появлении \S , для которых еще нет ; . При каждом новом \S F -список наращивается. Когда встречается ; , элемент из F -списка передается в R -список с необходимыми изменениями. Когда все элементы F -списка перешли в R -список, то строки стека преобразуются и переносятся на выход. Таким образом, завершение макровывоза сопровождается удалением последнего элемента R -списка вместе с таблицей параметров, поэтому в элементах R -списка хранится адрес для возобновления сканирования — AC_i . Полный формат элементов R -списка следующий:

l	P_i	AC_i	таблица параметров	w
-----	-------	--------	--------------------	-----

где l — длина от левого символа l до конечного маркера w , P_i — адрес предыдущего элемента R -списка, AC_i — адрес ячейки стека, которая сканировалась, когда встретилась ; .

Так как элементы F -списка преобразуются в элементы R -списка, то элементы F -списка содержат такое же количество ячеек

H_i	F_i	0
-------	-------	---

где F_i — адрес предыдущей ячейки F -списка, H_i — значение H , когда встретился \S , т. е. для $H \neq 0$ — адрес

ячейки стека для незаполненной строки. В ячейку Н помещается адрес начала незаполненной строки стека при перемещении символа из входной строки в стек, по окончании просмотра строки в Н заносится 0.

Текущий адрес стека хранится в ячейке S, поэтому при появлении конца строки разность $(S - H)$ показывает длину строки, которая помещается по адресу Н, строка — это последовательность непредупреждающих символов.

УМГ Стречи преобразует текст только в том случае, если в нем есть предупреждающие символы и он не применим для преобразования произвольно заданного текста.

Преобразование произвольного текста выполняется с помощью шаблонов. Одной из первых работ в этом направлении можно считать создание макрогенератора ЛИМП (LIMP — Language Independent matching Macro Processor) [148]. Основным качеством ЛИМПа является максимальная простота реализации. Для этого сначала вводятся в систему все макроопределения, а затем преобразуемый текст. Первая строка макрораспределения всегда содержит шаблон. Шаблон состоит из обычных символов и параметров, обозначаемых звездочкой *. Текст сравнивается с шаблоном с точностью до одного символа. Все символы шаблона, за исключением параметров, должны полностью совпадать с символами строки, тогда как * (звездочке) может соответствовать произвольный символ.

Параметры пронумерованы в шаблоне справа налево по порядку. Их номера используются в теле макроопределения для указания заменяемых мест. Те символы, которые совпадают со звездочками, получают номера в соответствии с номерами звездочек. Макроопределение заканчивается символом MACEND. Например,

$* = * + *$

СЧИТАТЬ %2

СЛОЖИТЬ %3

ЗАПИСАТЬ %1

MACEND

макроопределение

ЕСЛИ (*), *, *

СЧИТАТЬ %1

ПЕРЕХОД-ЕСЛИ-МЕНЬШЕ-НУЛЯ L %2

ПЕРЕХОД-ЕСЛИ-РАВНО-НУЛЮ L %3

макроопределение

ПЕРЕХОД-ЕСЛИ-БОЛЬШЕ-НУЛЯ L %4

MACEND

* STOP

L %1 ОСТАНОВ

макроопределение

MACEND

X = A + B

ЕСЛИ (X), 2, 7, 8

7 STOP

Результаты работы ЛИМПа:

СЧИТАТЬ A

СЛОЖИТЬ B

ЗАПИСАТЬ X

СЧИТАТЬ X

ПЕРЕХОД-ЕСЛИ-МЕНЬШЕ-НУЛЯ L₂

ПЕРЕХОД-ЕСЛИ-РАВНО-НУЛЮ L₇

ПЕРЕХОД-ЕСЛИ-БОЛЬШЕ-НУЛЯ L₈

L₇ ОСТАНОВ

В этом примере фиксируются следующие совпадения:

1) * = * + *

X = A + B

2) ЕСЛИ (*), *, *, *

ЕСЛИ (X) 2, 7, 8

3) * STOP

7 STOP

Дальнейшее развитие методов макрогенерации шло по пути объединения поиска по шаблону с поиском предупреждающих символов [8]. В МЛ/1 (ML/1 — Macrolanguage One) макроопределения задаются в форме

MCDEF <распределенное имя>
AS <тело макроопределения>.

Например,
MCDEF MOVE TO; AS <
СЧИТАТЬ %A1.
ЗАПИСАТЬ %A2. > ;

В первой строке макроопределения размещается шаблон или распределенное имя, которое состоит из строк символов и промежутков между ними.

При сравнении с шаблоном строки должны точно совпадать, тогда как промежуткам между строками могут соответствовать произвольные наборы символов. Тело макроопределения размещается после AS в угловых скобках. В теле специальным образом отмечаются заменяемые строки, например, сочетание процента, буквы A, *выражения* и точки рассматривается как аргумент, номер которого равен значению *выражения*, т. е. %A1. обозначает первый аргумент. Если входной текст:

MOVE A TO B;

тогда, используя предыдущее макроопределение, получаем, что значение первого параметра есть — A, а второго — B.

Результатом работы МЛ/1 служит макрорасширение:

СЧИТАТЬ A
ЗАПИСАТЬ B

МЛ/1 обладает большим набором встроенных функций, выполняемых в период макрогенерации, среди которых основными являются операции безусловных и условных переходов и операция присваивания:

MCGO — перейти на
IF — если
UNLESS — до тех пор пока не...
MCSET — присвоить.

Например:
MCDEF MOVE TO; AS <

MCGO L1 UNLESS %A1. = 0
 ОЧИСТИТЬ-СУММАТОР MCGO L2;
 %L1. СЧИТАТЬ %A1.
 %L2. ЗАПИСАТЬ %A2. >;

Из этого макроопределения в выходной текст могут попасть только следующие строки: ОЧИСТИТЬ-СУММАТОР, СЧИТАТЬ, ЗАПИСАТЬ, остальные символы служат для модификации текста.

В том случае, если первый аргумент равен нулю, то можно опустить считывание, так как достаточно очистить сумматор и записать его содержимое в ячейку по адресу A2. Например, макровывоз

MOVE 0 TO B5;

макрорасширение

ОЧИСТИТЬ-СУММАТОР
 ЗАПИСАТЬ В

но, если макровывоз

MOVE A TO B;

то макрорасширение

СЧИТАТЬ A
 ЗАПИСАТЬ B.

В использованном макроопределении имеются метки периода макрогенерации L_i , которые вставлены в оператор MCGO. Символы % и точка (.) запрещают перенос меток в выходной текст.

Одним из основных методов МЛ/1 служит представление шаблона в виде совокупности разделителей D_i и аргументов A_i . Например,

MCDEF	:	MOVE	:	TO	:	:	:
D_0	A_0	D_1	A_1	D_2	A_2	D_3	A_3

Такой подход дает возможность рассматривать различные значения разделителей так же, как рассматриваются значения аргументов. Например,

RET A = B + C,
 RET A = B - C.

В этих двух выражениях знаки $+$ и $-$ можно считать различными значениями разделителя, для этого введем макроопределение следующим образом:

MCDEF RET = OPT + OR - ALL; AS $\langle \dots \rangle$;

где фраза

OPT $\langle D_i \rangle$ OR $\langle D_i \rangle$ OR $\langle D_i \rangle$ ALL

указывает возможные значения разделителя D_i .

Если количество возможных значений разделителей заранее неизвестно, как например в выражении с переменным числом слагаемых

LET A = B + C - D + F - G;

LET X = Y;

LET Y = X + Y + C + D;

то группа разделителей, повторяющихся неизвестное число раз, объединяется одной меткой, например

MCDEF-LET = N1 OPT + N1 OR - ALL;

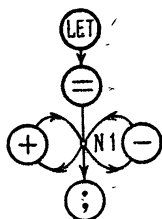


Рис. 3.

В МЛ/1 широко используется операция присваивания, например:

MCSET T5 = 5;

Специальный символ зарезервирован в системе для обозначения переменных, имеющих значение только внутри данного макроса — T_i . Кроме того, выделяются $T1$ и $T2$, $T1$ указывает количество аргументов текущего макроса, а $T2$ — количество макросов, выполненных к текущему моменту.

Пусть входной текст:

MCDEF LET = N1OPT + N1OR - ALL;

AS < СЧИТАТЬ %A2. ;

MCSET T1 = 3;

%L4. MCGO L2 IF %T1 - 1. = +;

MCGO L6 IF %T4 - 1. = -;

%L6. ВЫЧЕСТЬ %AT1. ;

MCGO L3;

%L2. СЛОЖИТЬ %AT1. ;

%L3. MCSET T1 = T1 + 1. ;

MCGO L4;

%L5. ЗАПИСАТЬ %A1.>;

LET A = B - C + D; — преобразуемый
текст.

Тогда макрорасширение есть:

СЧИТАТЬ B

ВЫЧЕСТЬ C

СЛОЖИТЬ D

ЗАПИСАТЬ A

МЛ/1 реализован на основе дескриптивного языка, описывающего промежуточную абстрактную машину. Этот язык представляет набор макросов на машинно-зависимом языке. Общее количество команд дескриптивного языка составляет около двух тысяч, для сравнения следует заметить, что УМГ Стречи занимает около двух сотен ячеек. МЛ/1 обладает хорошей мобильностью, так как макросы дескриптивного языка реализуются достаточно просто для широкого класса машин.

При отображении методов универсальных макрогенераторов в структуру ЭВМ важное место занимает минимальное количество встроенных функций и наличие единственного стека. Все еще остается проблемой сохранение текущего состояния стека для генерации диагностических сообщений, соответствующих строкам входного текста.

1.3. Операции обработки строк

Решение задачи обработки строк можно рассматривать как реализацию совокупности операций над символами. Такая позиция в общем предполагает отображение этих операций на специализированный символьный процессор.

Исторически языки обработки строк возникли одновременно с языками цифровой обработки, например, одновременно с языком фортран. Как те, так и другие, были развитием автокодов и шагом в сторону пользователя. Фортран сделал существенный шаг вперед тем, что в нем впервые был введен синтаксис, отличный от синтаксиса машинных команд. Поэтому первичный анализ языка обработки строк можно провести, оценивая те черты синтаксиса, которые сближают операторы языка с записью машинных команд.

Первым нецифровым языком считается ипл (IPL), разработанный под руководством Ньюэла (Newell) в 1957 г. для работ по искусственному интеллекту, а именно, для имитации человеческого мышления и доказательства теорем. Основной недостаток ипл был в его сходстве с автокодом машин тех времен, возникающий же тогда фортран имел успех именно потому, что сделал шаг в направлении математических обозначений числовых вычислений, развиваемых и исследованных в течение столетий, тогда как методы математической обработки строк и соответствующие обозначения начали появляться только с появлением ЭВМ.

Частично задача обработки символов была решена и в фортране, при введении операторов ввода/вывода, которые согласуют символьное представление информации на внешних устройствах с числовым представлением в процессоре.

В общем понятие языка высокого уровня достаточно относительно. По мере того как аппаратура приближается к языку высокого уровня недавнего прошлого, язык высокого уровня настоящего момента претерпевает существенные изменения понятий. Ограничения аппаратуры всегда будут определять тот язык, который ей соответствует как автокод, хотя в прошлом он бы мог служить языком высокого уровня,

и прогресс теоретических исследований по программированию всегда будет отрывать язык высокого уровня настоящего момента от аппаратуры.

Существующие ЦВМ в общем не приспособлены для обработки символов. Так, умножение двух чисел, каждое из которых состоит из восьми цифр, выполняется примерно за 1 мкс на ЦВМ средней мощности, в то же время считается, что человеку нужно около 20 с для такого умножения. С другой стороны, поиск слова из восьми символов в тексте из 1000 символов требует на той же ЦВМ примерно 2 мкс, а человек делает это за те же 20 с. В общем, можно сказать, что разработка ЦВМ для обработки чисел примерно в 1000 раз выгоднее, чем для обработки символов.

Одним из первых языков, разработанных с целью «механической трансляции» программ, был введенный в 1958 г. язык Comit [157]. Тем не менее до сих пор нельзя сказать, что есть практически устоявшиеся операции над строками, как, например, привычно для арифметики. Достаточно часто в операциях обработки строк встречаются следующие операции:

- 1) конкатенация, т. е. объединение двух строк в одну;
- 2) идентификация подстрок, т. е. разбиение строк на именуемые подстроки;
- 3) поиск по шаблону;
- 4) преобразование идентифицируемых подстрок.

Комит важен именно потому, что в нем впервые был введен единообразный механизм поиска по шаблону с замещением совпавших подстрок. Операция замещения или подстановки вообще является фундаментальной составляющей операцией при построении алгоритмов [34], и выполнение любого алгоритма можно приближенно считать серией последовательных выполнений операции подстановки.

Основной единицей языка комит служит *конституента*. В качестве разделителя конституент был принят знак +, вместо пробела использовался знак —, и вообще любой символ, отличный от буквы, имел синтаксическое значение. Для хранения строк в комите было принято 128 так называемых *полок* и одна рабочая область, куда необходимо переместить содержимое одной из полок перед обработкой. Всего в про-

грамме должно быть не более 129 отличающихся строк.

Для указания литеральных символов использовалась * (звездочка), т. е. * может запретить рассматривать символы как синтаксические знаки. Например, строка «44 веселых чижа» в комите должна быть представлена так:

* 4 * 4 + . — + веселых + — + чижа +

где + отделяет конститuentы, а — эквивалентен пробелу.

Комит-программа состоит из последовательности правил, каждое из которых имеет пять полей:

- 1) левая часть или шаблон;
- 2) знак равенства;
- 3) правая часть или строка замещения;
- 4) символ управления, представленный двумя наклонными чертами;
- 5) указатель следующего оператора, обычно в виде go to <метка>.

Метка может быть помещена перед любой строкой. Левая часть представляет шаблон, который разыскивается в строке рабочей области. В случае совпадения шаблона с подстрокой правая часть используется для замены.

Шаблон задается в виде набора литеральных, т. е. представляющих самих себя, и нелитеральных символов. В общем случае нелитеральные строки представляют «все, что угодно», а литеральные строки должны совпадать с точностью до символа. В левой части правила могут быть использованы следующие символы:

\$, \$n, <буква>, <число>,

где \$ обозначает произвольную подстроку, \$n обозначает подстроку из конститuent, <буква> представляет литеральную подстроку, <число> указывает номер конститuentы в левой части, используемой в данном месте шаблона.

Например, в рабочей области находится строка

A + B + C + A + B

и к ней применяется шаблон

\$ + C + A + \$

тогда первому символу \$ соответствует $A + B$, а последнему — B . Пусть задано правило:

$$\$ \div C + A + \$ = 2 + 1 + 3$$

где числа в правой части указывают номер конститuenty, поэтому устанавливается следующее соответствие:

1. \$ $A + B$
2. C C
3. A A
4. \$ B

и результат операции есть

$$C + A + B + A$$

Пусть, далее, в рабочей области находится строка

$$P + R + O + G + R + A + M + M + E + R$$

и есть правило

$$P + \$1 + \$ + 2 + \$1 + \$ = 2 + 5 + Z + O + R$$

тогда устанавливается следующее соответствие между левой частью правила и рабочей областью:

1. P P
2. \$1 R
3. \$ $O + G$
4. 2 R
5. \$1 A
6. \$ $M + M + E + R$

где в четвертой строке цифра 2 указывает, что надо взять значение второй конститuenty левой части. Результат работы программы

$$R + A + Z + O + R$$

Хотя последняя версия комита — комит II (Commit II) — реализована на машине IBM System/360 и 370, в настоящее время этот язык практически не используется.

Следующий шаг вперед в обработке строк был сделан языком снобол (SNOBOL) [96]. В нем вместе с опе-

рацией поиска по шаблону был явно введен оператор присваивания в виде

$\langle \text{имя} \rangle = \langle \text{значение} \rangle$

Снобол за все время своего существования не получил широкого распространения, и прежде всего потому, что ЦВМ, существовавшие в момент его появления, были плохо приспособлены для выполнения операций обработки символов. В первой реализации на IBM7090 символы выделялись с помощью медленных операций сдвига, и такая реализация оказалась совершенно не эффективной.

Основным элементом языка снобол служит строка. Строка, взятая в кавычки, представляет самое себя, т. е. литерал. Типичная операция над строкой — это выяснение ее структуры с помощью шаблона, который может содержать несколько альтернатив. В языке не используется описание типа, собственно понятие типа реализуется шаблоном. В сноболе существуют ограниченные средства работы с числами, позволяющие найти номер символа, число символов в строке, преобразовать символы в число и т. п.

Идентификаторы переменных начинаются с буквы. Знаки операций разделяются пробелами, причем, если операнды арифметических операций представляют строки только из цифр, то выполнение операции сопровождается вычислением значений. Например, присвоить значение

$X = 5$

$Y = 14 + 16 - 10$

$S = \text{'ПРИМЕР'}$

Оператор должен размещаться на одной строке. Перенос разрешается только в тех местах строки, в которых стоит пробел, и следующая строка должна начинаться со знака +.

Знак операции конкатенации обозначается пробелом, например:

$A1 = \text{'ФУТ'}$

$A2 = \text{'БОЛ'}$

$A3 = A1 A2,$

результат конкатенации есть ФУТБОЛ.

Если в выражении встречаются арифметическая операция и конкатенация, то конкатенация выполняется в последнюю очередь

ВАГОН = 5

ПОЕЗД = 'СТРЕЛА'

ОТЪЕЗД = ПОЕЗД ВАГОН

это выражение эквивалентно следующему:

ОТЪЕЗД = 'СТРЕЛА' 5

далее, если применить операцию сложения

ОСТАНОВ = ПОЕЗД ВАГОН + 10

то результат будет:

'СТРЕЛА 15'

Поиск по шаблону задается в виде:

⟨объект⟩ ⟨шаблон⟩ = ⟨строка⟩

это означает, что к строке, называемой объектом, применяется шаблон, и в случае совпадения с подстрокой подстрока заменяется на строку. Пусть, например:

ПРОФ = ПРОГРАММИСТ

ПРОФ 'ОГРАММ' = 'ИБОР'

тогда результат есть:

'ПРИБОРИСТ'

Сам объект может быть литеральной строкой, например, 'СТРЕЛА 5' и операция поиска по шаблону может быть представлена:

'СТРЕЛА 5' ПОЕЗД ВАГОН + 10

Здесь сначала выполняется сложение, затем конкатенация и, наконец, поиск по шаблону 'СТРЕЛА 15' в объекте 'СТРЕЛА 5'.

Если необходимо получить объект как конкатенацию строк, то операнды заключаются в скобки:

ЧАСЫ = 19

МИНУТЫ = 15

(ЧАСЫ МИНУТЫ) 21.00 = 'ВРЕМЯ'

если бы шаблон формировался конкатенацией, то скобки бы не потребовались.

Например,

КОМАНДА = 'СПАРТАК'

СТАДИОН = 'ТА'

ИГРОК = 'К'

КОМАНДА СТАДИОН ИГРОК = 'ЖА'

тогда результат равен

'СПАРЖА'

При поиске по шаблону можно задавать альтернативы поиска, т. е. последовательный поиск по нескольким шаблонам, такие шаблоны разделяются вертикальной чертой. Строки, разделенные вертикальными чертами, образуют новую строку, которая может быть идентифицирована именем. Например,

СИНТЕЗ = 'АЛГОРИТМЫ'|'СТРУКТУРА'

ФРАЗА = 'ПРОГРАММА И СТРУКТУРА
АДЕКВАТНЫ'

ФРАЗА СИНТЕЗ = 'ДАННЫЕ'

тогда результат равен

'ПРОГРАММА И ДАННЫЕ АДЕКВАТНЫ'

Операндами операции конкатенации могут быть шаблоны, каждый из которых представлен несколькими альтернативами, например

БАЗА = 'ФИКСИР'|'ПЛАВ'

МАС = 'ДВОИЧН'|'ДЕСЯТ'

ТИП = БАЗА МАС

ПЕРЕМ = 'X-ПЛАВДЕСЯТ'

ПЕРЕМ ТИП = 'КОНТРОЛЬ'

итак результат есть

'X-КОНТРОЛЬ'

В этом примере ТИП принимает следующие значения:

'ФИКСИРДВОИЧН', 'ФИКСИРДЕСЯТ',

'ПЛАВДВОИЧН', 'ПЛАВДЕСЯТ'

При поиске по шаблону не всегда точно известно, с каким из шаблонов произошло совпадение, поэтому в языке можно использовать переменную, которая в случае совпадения принимает значение совпавшего шаблона, например

БАЗА = ('ДВОИЧН'|'ДЕСЯТ') ОСН

тогда, если БАЗА используется в качестве шаблона, то при совпадении с 'ДЕСЯТ' переменная ОСН получит значение 'ДЕСЯТ', либо другая альтернатива. Вместо идентификатора переменной можно использовать идентификатор функции, тогда в случае совпадения такая функция выполняется, например

БАЗА = 'ДВОИЧН'|('ДЕСЯТ'.OUTPUT)

и в случае совпадения с 'ДЕСЯТ' на печать будет выдано ДЕСЯТ.

Операторы передачи управления в языке снобол обозначаются зарезервированными символами: F — для лжи, S — для истины, помещаемыми в конце строки, например, F(M) обозначает переход на оператор с меткой M в случае неудачи (ложь) при поиске по шаблону.

Пусть, например, из КРАСКА необходимо исключить ЦВЕТ:

ЦВЕТ = 'КРАСН'|'ЗЕЛЕН'
'СИНИЙ'

... КРАСКА = ...

ЯРК КРАСКА ЦВЕТ =
: S(ЯРК) F(ТУСКЛ)

ТУСКЛ ...

в этом примере к тексту будет столько раз применяться шаблон, сколько будет совпадений с 'КРАСН', 'ЗЕЛЕН', 'СИНИЙ'.

Общий вид оператора снобола следующий:

⟨метка⟩ ⟨объект⟩ ⟨шаблон⟩ = ⟨строка⟩

Последовательность применения шаблона к объекту описывается алгоритмом «протаскивания иглы через бусину», который поясняется на конкретном примере. Вначале вводятся определения:

1) курсор — указатель на сканируемый символ входной строки, обозначается \uparrow ;

2) игла — указатель на применяемую часть шаблона, которая получается в результате конкатенации альтернатив, обозначается \rightarrow ;

3) бусина — часть шаблона, уже совпавшая с символами входной строки.

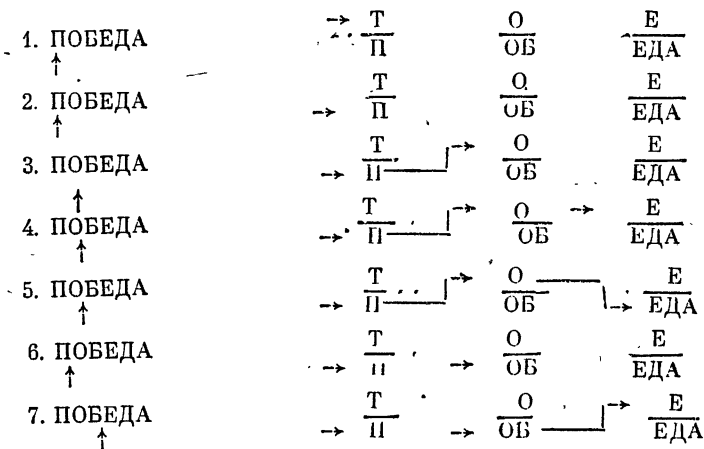
Пусть шаблон есть:—

ПРИМ = ('Т'|'П') ('О'|'ОБ') ('Е'|'ЕДА')

и объект в операции поиска по шаблону

ПОБЕДА

тогда шаги поиска можно представить в следующем виде:



На первом шаге сравнивается П с Т, но так как они не совпадают, то на втором шаге игла переходит на П, и здесь возникает совпадение. При совпадении игла пронзает бусину, и курсор передвигается на О. Поэтому на третьем шаге возникает совпадение, и игла пронзает бусину, а курсор передвигается. На четвертом и на пятом шагах игла упирается в Е и ЕДА, но так как курсор показывает на Б, то — нет совпадения, и ввиду отсутствия вариантов перебора для иглы предыдущее совпадение признается неправильным, и

делается попытка отыскать новое совпадение, вернув курсор. На шестом шаге курсор показывает на О, а игла перемещается на бусину ОБ. Так как на шестом шаге игла пронзает бусину, то курсор вновь передвигается вперед. На седьмом шаге игла показывает на Е, и так как возникает совпадение, то игла, пронзив набор альтернатив ПОБЕ, заканчивает сравнение.

Большое количество возможных вариантов при выполнении поиска по шаблону потребовало введения в язык специальных средств оптимизации. Такими средствами являются, например, сравнение только по первым символам строк, сравнение только атомов, частичное сравнение для обнаружения несовпадения и т. д. В программе вместо полного сравнения можно сначала задать сравнение с укороченным шаблоном, и только в случаях совпадения для совпавших строк задать полный шаблон. В языке снобол встроен большой набор системных функций, ускоряющих поиск, например, функция BREAK имеет аргумент, который представляет шаблон, совпадающий со всеми символами, кроме указанных в аргументе. Так, BREAK () выбирает строку до первого пробела. Функция SPAN выбирает, начиная с курсора, наидлиннейшую строку, которая содержит хотя бы один из символов, содержащийся в аргументе, так, SPAN () будет выбирать все подряд идущие пробелы независимо от количества.

Часто используется функция LEN, значение которой есть шаблон, который подходит к любой строке заданной длины. Длина задается аргументом LEN.

Для того чтобы ввести новую подпрограмму, в сноболе используется встроенная функция DEFINE, например

```
DEFINE (F(X, Y), L1, L2, 'ВХОД')
```

где F — имя функций, X, Y — формальные параметры, L1, L2 — локальные переменные, 'ВХОД' — метка точки входа.

Строки тела подпрограммы размещаются между символами DEFINE и END.

Язык снобол прошел длительный путь развития от первых вариантов до снобола-4 (SNOBOL-4), который реализован более чем на десяти типах ЦВМ.

Обработка строк символов в принципе ориентирована на один символ, поэтому эффективность ее ниже цифровой обработки, хотя бы из-за того, что длина символа меньше длины числа в 6—8 раз.

Аппаратура памяти ЦВМ обычно состоит из ячеек фиксированной длины в несколько десятков разрядов, поэтому поиск конкретного символа требует выбора ячейки, а затем выбора символа из ячейки. Строки символов имеют переменную длину, тогда как длина аппаратных регистров фиксирована. Поэтому требуется дополнительная аппаратура, обрабатывающая строку по частям. Трудности решения перечисленных проблем приводят к тому, что современные ЦВМ, даже очень мощные, выполняют последовательную обработку строк символ за символом [75].

Основным элементом структуры ЦВМ, обрабатывающей строки, служит аппаратура непрерывной выборки слов из оперативной памяти, соединенная с устройством обработки символов и устройством непрерывной записи слов, составленных из строк, в память. В настоящее время в связи с требованиями практики обработка строк становится экономически выгодной, так как большую часть стоимости аппаратуры составляет память, и плотная упаковка символов в одной ячейке, дающая возможность экономить память, сочетается с одновременной обработкой нескольких символов в процессоре с совмещенным выполнением операций.

1.4. Операции обработки списков

Размещение строк символов в памяти, представляющей вектор ячеек фиксированной длины, связано со значительными потерями времени на перекомпоновку массивов. Эти потери существенно сокращаются переходом на списочную память.

В списочной памяти соседние элементы не обязательно располагать в ячейках с последовательными адресами. Адрес следующего элемента списка хранится в предыдущем, что дает возможность упростить операции по распределению памяти. При этом достаточно исправить адрес, хранящийся в ячейке, чтобы получить новую структуру памяти, т. е. структура памяти хранится в ней самой.

В 1957 г. группа под руководством Ньюэлла (Newell Alen) на IBM 650 реализовала ипл-V (IPL-V), который получил в то время широкое распространение: IBM650, 704, 709, 7090, 1620, Univac 1105, 1107, CDC 1604, GE 20, BURROUGHS 220, PHILCO 2000, AN/FSQ-32.

Несмотря на то что ипл-V был пятым вариантом языка, в котором разработаны все основные операции обработки списков, по синтаксису он был очень близок к автокоду машин того времени, что оказалось существенным недостатком и привело к постепенному исчезновению этого языка.

В 1959 г. сотрудниками Массачусетского технологического института, работающими над проблемами искусственного интеллекта, под руководством Мак-Карти (McCarthy John) [128] разработан язык лисп-1,5 (LISP-List processing), в котором был существенно упрощен синтаксис, введена непрерывная горизонтальная скобочная запись и последовательно проведена концепция нахождения значения по имени. В лиспе *найти значение* означает остановиться в последовательном прохождении по косвенным ссылкам.

Команды и данные представлены в лиспе в виде символьных строк, рассматриваемых как единое целое и называемый атомами. Так как атом по своей сути такой объект, который нельзя разделить, то внутреннее представление атома может быть ячейкой фиксированной длины. Все операции в лисп-системе производятся над словами, за исключением связей с пользователем, когда используется символьное представление атома, называемое внешним, т. е. в случаях ввода или вывода информации.

Атомы разделяются ограничителями; обычно пробелами. Списки образуют атомы, разделенные пробелами и взятые в скобки. Элементом списка может быть и атом, и список. Список всегда начинается с открывающей скобки и заканчивается закрывающей. Например, атомы

AB A BY 123 C5 HHO0336 ,

списки

(234) (BA CD) (A(BC)H) (BA(BC YB)) ((MA)) ().

Списки и атомы называются S-выражениями. Следует заметить, что слово *список* используется как для списка в S-выражениях, так и для списка ячеек памяти, образующего структуру памяти.

Язык лисп имеет почти минимальный набор встроенных функций, на основе которого могут быть сформированы произвольные сложные функции. Функция CAR (Contents of the Address Register) имеет значение, равное значению первого элемента списка, представляющего значение аргумента. Функция CDR (Contents of the Decrement Register) имеет значением то, что остается после удаления первого элемента из списка, представляющего значение аргумента. Таким образом, функции CAR и CDR рассматривают свои аргументы как имена и применяются к их значениям.

Простейшим случаем значения является само S-выражение, т. е. то, как оно записано. Такое значение вырабатывается функцией QUOTE. Если необходимо нейтрализовать действие функции QUOTE, то применяется функция EVAL, которая требует вычислить значение аргумента. Например,

$$\begin{aligned}(\text{CAR} - (\text{QUOTE} - (A - B))) &\rightarrow A \\(\text{CDR} - (\text{QUOTE} - (A - B))) &\rightarrow (B)\end{aligned}$$

где \rightarrow обозначает «значение есть».

Объединение списков и атомов для построения новых списков выполняется встроенной функцией CONS, которая образует список из значения своего второго аргумента, а оно, в свою очередь, должно быть списком, если подставить в это значение в качестве первого элемента значение ее первого аргумента. Например,

$$\begin{aligned}(\text{CONS} - (\text{QUOTE} - A) - (\text{QUOTE} - (C - D))) &\rightarrow \\&\rightarrow (A - (C - D)).\end{aligned}$$

Формирование подпрограмм в лиспе осуществляется с помощью двух встроенных функций SEXPR и LAMBDA. Функция LAMBDA указывает связь имени функции, представленного атомом, с телом. Функция LAMBDA перечисляет формальные параметры в теле функции. Общая форма введения новых подпрограмм следующая:

$$(\text{SEXPR} - f - (\text{LAMBDA} - (X - Y - \dots - Z) - e))$$

где f — атом, соответствующий имени введенной функции; X, Y, \dots, Z формальные параметры; e — выражение, содержащее эти параметры.

Оператор обращения к функции записывается в виде

$$(f - W - T - \dots - V)$$

где f — атом, W, T, \dots, V — фактические параметры, значения которых подставляются в выражение вместо формальных.

Функцию LAMBDA называют также *определяющим выражением* функции. Если в определяющее выражение функции f входит обращение к f или к другим функциям, в определяющих выражениях которых содержится обращение к f , то функция f называется рекурсивной.

Необходимым элементом правильно записанной рекурсивной функции, а также всякого рода циклов и ветвлений, служит присутствие условной встроенной функции COND. Общая форма записи функции COND следующая:

$$(COND - (p_1 - e_1) - (p_2 - e_2) - \dots - (p_N - e_N))$$

где p_i — предикаты, e_i — выражение, $i = (1 \div N)$.

Предикаты — это логические функции, принимающие значения встроенных констант Т или NIL (исти́на или ло́жь). Атомы Т и NIL зарезервированы в языке для этих целей. Значение функции COND равно значению того выражения e_k , для которого p_k имеет значение Т, а все p_j для $j = 1 \div (k - 1)$ имеют значение NIL.

Примерами встроенных предикатов могут служить функции: АТОМ, EQ. Функция АТОМ имеет значение Т, если ее аргумент имеет значение атом. Функция EQ имеет значение Т, если значения двух ее аргументов суть совпадающие атомы. В противных случаях эти функции имеют значения NIL.

Например,

$$\begin{aligned} (ATOM - (QUOTE - A)) &\rightarrow T \\ (EQ - (QUOTE - A) (CAR - \\ &\quad (QUOTE - (A - B)))) &\rightarrow T, \end{aligned}$$

$(\text{NULL} \rightarrow \text{T}) \rightarrow \text{NIL}$
 $(\text{NULL} \rightarrow ()) \rightarrow \text{T}$
 $(\text{ATOM} \rightarrow ()) \rightarrow \text{T}$
 $(\text{ATOM} \rightarrow \text{NIL}) \rightarrow \text{T}$

Следует заметить, что пустой список есть атом NIL.

Пусть требуется записать определяющее выражение функции, которая узнает, не является ли X элементом списка Y, если все элементы списка — атомы:

$(\text{SEXPR} \rightarrow \text{MEMBER} \rightarrow (\text{LAMBDA} \rightarrow (\text{X Y}) \rightarrow$
 $(\text{COND} \rightarrow ((\text{NULL} \rightarrow \text{Y}) \rightarrow \text{NIL}) \rightarrow$
 $((\text{EQ} \rightarrow \text{X} \rightarrow (\text{CAR} \rightarrow \text{Y})) \rightarrow \text{T}) \rightarrow$
 $(\text{T} \rightarrow (\text{MEMBER} \rightarrow \text{X} \rightarrow (\text{CDR} \rightarrow \text{Y}))))).$

Применение только аппарата рекурсивных функций не всегда удобно, поэтому в лиспе имеются и традиционные средства программирования. В язык лисп введена функция PROG, которая задает локальные переменные, имеющие значение NIL в начальный момент, и список, состоящий из последовательно выполняющихся операторов. Эти операторы включают оператор присваивания — $(\text{SETQ} \rightarrow \text{X} \rightarrow \text{Y})$, оператор безусловной передачи управления на метку — $(\text{GO} \rightarrow \text{M})$, где M — атом, оператор выхода из PROG — $(\text{RETURN} \rightarrow \text{Z})$, при выполнении которого функция получает значения Z. Общая форма записи функции PROG следующая:

$(\text{PROG} \rightarrow (\text{X1} \rightarrow \text{X2} \rightarrow \dots \rightarrow \text{XN}) \rightarrow e)$

где X1, X2, ..., XN — локальные или связанные переменные, e — выражение.

В e могут использоваться и ранее введенные функции CAR, CDR, CONS, COND, ATOM, EQ и т. д. Например, введем функцию, вычисляющую значение полинома:

$(\text{SEXPR} \rightarrow \text{P} \rightarrow (\text{LAMBDA} \rightarrow (\text{A0} \rightarrow \text{D} \rightarrow \text{X} \rightarrow \text{N}) \rightarrow$
 $\rightarrow (\text{PROG} \rightarrow (\text{S} \rightarrow \text{C} \rightarrow \text{A1} \rightarrow \text{Y}) \rightarrow (\text{SETQ} \rightarrow \text{Y} \rightarrow \text{N}) \rightarrow$
 $\rightarrow (\text{SETQ} \rightarrow \text{C} \rightarrow 0) (\text{SETQ} \rightarrow \text{A1} \rightarrow \text{A0}) \rightarrow$
 $\rightarrow (\text{SETQ} \rightarrow \text{S} \rightarrow 0) \rightarrow \text{M} \rightarrow (\text{COND} \rightarrow ((\text{EQ} \rightarrow \text{C} \rightarrow \text{N}) \rightarrow$
 $\rightarrow (\text{RETURN} \rightarrow \text{S}))) \rightarrow (\text{SETQ} \rightarrow \text{A1} \rightarrow (\text{PLUS} \rightarrow \text{A1} \rightarrow \text{D})) \rightarrow$
 $\rightarrow (\text{SETQ} \rightarrow \text{S} \rightarrow (\text{PLUS} \rightarrow \text{S} \rightarrow \text{A1})) \rightarrow (\text{SETQ} \rightarrow \text{S} \rightarrow$

$$\begin{aligned} & - (\text{TIMES } - S - X)) - (\text{SETQ } - C - \\ & - (\text{PLUS } - C - 1)) - (\text{GO } - M))) \end{aligned}$$

В этом примере атомы, состоящие из цифр, представляют константу, значение которой есть значение числа, образованного этими цифрами, функция PLUS суммирует, а TIMES умножает значения своих аргументов.

Одним из недостатков лиспа является монотонность скобочной записи, существует метаязык для лиспа, который имеет другой синтаксис.

Например, запись уже известных функций на метаязыке становится более структурированной:

$$\text{car}[(A - B)] = A$$

$$\text{cdr}[(A - B)] = (B)$$

$$\text{car}[\text{cons}[x; y]] = x$$

$$\text{label}[ff; \lambda[[x]; [\text{atom}[x] \rightarrow x; T \rightarrow ff[\text{car}[x]]]]]$$

где *ff* — имя функции; λ — LAMBDA-функция; \rightarrow связывает пару $(p_i \rightarrow e_i)$ в выражении COND; *label* — аналог SEXPR.

Язык лисп имеет минимально необходимый набор встроенных функций, на базе которых строятся более сложные функции. Работа этих функций ведется над данными, имеющими фиксированное внутреннее представление. Списки ячеек памяти построены из элементов, использующих внутреннее представление, центральным звеном которого служит таблица свойств атомов или *p*-список (*property*). Атом в символьном виде размещается в массиве внешних представлений, и каждому атому ставится в соответствие с помощью функции расстановки адрес ячейки массива *p*-списка. Ячейка *p*-списка содержит код типа атома, адрес начала символического представления атома и адрес-ссылку на элемент списка, связанного с данным атомом. Ячейка *p*-списка аналогична дескриптору.

В языке лисп используется незначительное количество типов атомов:

- 1) имя встроенной функции;
- 2) имя встроенной функции с аргументами специального вида;
- 3) константа;
- 4) число;

5) имя функции, заданной определяющим выражением;

6) имя функции, заданной определяющим выражением с аргументами специального вида;

7) все остальные атомы.

Часто встречающимися операциями в листе является анализ типов атомов и получение значения по адресу, хранящемуся в ячейке. Характер таких операций требует завершения предыдущего обращения перед выполнением следующего. Совмещение операций подобного типа может быть реализовано аппаратурой параллельного выполнения косвенных обращений.

В начале работы системы память программы представлена списком свободных ячеек. Затем, после некоторого периода работы программы, списки можно поделить на две группы:

1) рабочие списки, к элементам которых можно обратиться через некоторую начальную ячейку — голову списка;

2) недоступные списки, которые образовались в результате реформирования рабочих списков и которые не нужны для дальнейшей работы программы.

Поэтому в системе лист обязательно присутствует программа реорганизации памяти, которая преобразует ненужные списки в список свободных ячеек. Такая программа называется программой сборки мусора.

Выполнение программы сборки мусора состоит из нескольких этапов:

на первом этапе просматриваются рабочие списки, и их ячейки отмечаются специальным маркерным битом;

на втором этапе, при последовательном сканировании памяти, все немаркированные ячейки соединяются в список свободных ячеек; одновременно все списки уплотняются.

Важное значение имеет момент включения в работу программы сборки мусора. Обычно это происходит в момент полного заполнения памяти. Программа пользователя останавливается, и в течение длительного времени выполняется сборка мусора.

При работе программы в виртуальной памяти, имеющей бесконечные размеры, нельзя слишком долго

обрабатывать списки без реорганизации памяти, так как возникает эффект дробления памяти, когда ячейки рабочих списков распределяются равномерно среди большой массы ячеек недоступных списков, что приводит к увеличению потока обменов с внешней памятью и к снижению производительности системы. Поэтому программа сборки мусора в виртуальной памяти включается периодически через короткие интервалы времени для переработки буферных массивов, включающих целое число листов. Буферный массив исключается из работы программы, и роль синхронизатора при прерываниях играет аппаратура защиты листов.

Существенный прогресс в алгоритмах сборки мусора был достигнут введением параллельной сборки [104, 132, 147]. Такие алгоритмы могут использовать параллельные специализированные процессоры и давать значительный эффект в режиме реального масштаба времени, когда диапазон времен ответов не должен превышать нескольких секунд, в то время как при последовательном алгоритме сборки мусора длительность возможных задержек колеблется от 6 секунд до 9 минут по данным Уодлера [147].

1.5. Операции обработки деревьев

Дерево — это такая структура данных, которая соответствует элементам конечного множества, среди которых фиксируется один элемент, называемый корнем и остальные элементы образуют попарно непересекающиеся подмножества. С корнем связывается подмножество вершин, называемых первым уровнем дерева. Каждая вершина первого уровня может быть корнем для некоторого подмножества. Это подмножество образует второй уровень и т. д.

Строку, содержащую сбалансированные скобки, можно представить в виде дерева. В структурированной таким образом строке операция поиска по шаблону подстроки заменяется операцией сравнения деревьев. Деревья кодируются при вводе с помощью симметричных списков, поэтому время поиска по структурированным строкам значительно меньше, чем, например, время поиска в языке снобол.

Операции над деревьями появились в законченном виде в языке амбит (Agronomy May Be Ignored Totally) [89] и привлекли всеобщее внимание.

Основная операция языка амбит очень похожа на поиск по специальному шаблону, строка шаблона содержит указатель и сбалансированные скобки. Указатель обозначается Δ <имя>.

При поиске по специальному шаблону к строке данных примеряется левая часть правила записи так, чтобы совпали указатели, скобки, литералы и т. п., промежутки между совпавшими частями ставятся в соответствие параметрам. Затем по результату совпадения осуществляется замена на правую часть.

Например, пусть правило

$$\Delta p(A * A) \rightarrow \Delta p(A \uparrow 2)$$

и строка данных

$$E\Delta Q((ALPHA + \Delta p((M + 1) * (M + 1))) = 6.0)$$

тогда результат преобразования следующий:

$$E\Delta Q((ALPHA + \Delta p((M + 1) \uparrow 2)) = 6.0)$$

Для следующего примера заменяются на a все *неуказатели*, называемые сегментами, и обозначаются в правиле: e — элемент, s — строка.

Пусть правило

$$(e1(s1 \Delta a s2)) \rightarrow (e1 \Delta a(s1 s2)),$$

и строка

$$(a(aa \Delta a(aaa))aa)$$

тогда применение левой части правила к строке создает следующее дерево, слева записана строка, справа — левая часть правила:

(левый контекст
a	e1
((
aa	s1
Δ	Δ
a	a
(aaa)	s2
)aa)	правый контекст

и по правой части правила сформируется результат $((a\Delta a(aa(aaa)))aa)$.

Во внешнем представлении, для пользователя, строка данных со сбалансированными скобками есть линейная последовательность символов. Во внутреннем представлении строка данных представляет симметричный список, в котором связи соединяют сегменты и скобки, кроме того, каждый сегмент правой части правила имеет однонаправленную ссылку на соответствующий сегмент левой части, и указатель из левой части имеет ссылку на указатель правой части.

Во время выполнения каждый сегмент имеет также однонаправленную ссылку на строку данных. Амбит не получил широкого распространения, хотя он удобен и эффективен при обработке символов.

1.6. Операции обработки массивов чисел

Операции обработки массивов чисел в традиционных алгоритмических языках представлены выборкой элемента массива по индексу. Незначительно улучшены эти средства в языке алгол-68. Операции в алголе-68 дают возможность выделить из массива подмассив и расширить список операций над массивами с помощью конструкции *ор*. Определение операции, введенное конструкцией *ор*, допускает использование одинаковых идентификаторов для разных операций, например, $+$ можно использовать как для операции сложения матриц, так и для операции сложения векторов. Для идентификации операций в языке используется не только идентификатор, но типы операндов.

Сложная структура массива как операнда операции требует соответствующего контроля правильности выполнения операции по атрибутам описания массива. Важное значение при проверке правильности играет задание граничных пар для диапазона изменения индекса:

$$A[l_1 : u_1, l_2 : u_2, \dots, l_n : u_n]$$

где l_i — минимальное значение индекса по i -й координате, u_i — соответственно, максимальное значение.

При выделении подмассива из массива происходит соответствующая модификация граничных пар, так можно сузить диапазон изменения индекса: $l_i \leq v_i$, $t_i \leq u_i$,

$$A[l_1 : u_1, l_2 : u_2, \dots, v_i : u_i, \dots, l_n : u_n].$$

В крайнем случае индекс может иметь одно постоянное значение: x_k ,

$$A[l_1 : u_1, l_2 : u_2, \dots, x_k, \dots, l_n : u_n].$$

Но даже в таком мощном языке, как алгол-68, отсутствуют *встроенные* операции над векторами или матрицами, типа сложения и умножения. Чрезвычайная гибкость конструкции *ор* в общем приводит к широкому использованию процедур, в то же время с позиции анализа отображения операций на структуру необходима фиксация небольшого количества встроенных операций, которые были бы удобны в большинстве случаев.

Разнообразный набор встроенных операций содержит язык апл (APL) [114]. С помощью конструкций языка апл можно кратко и наглядно записать многие численные алгоритмы обработки массивов.

Принципом, положенным в основу введения операций над массивами, является простота обозначений, что играет важную роль при большом количестве операций.

Все операции поделены на бинарные и унарные, одни и те же идентификаторы операций имеют разный смысл в зависимости от того, является ли операция бинарной или унарной. К простейшим операциям апл относятся операции над скалярными величинами и над векторами.

В дальнейшем обсуждение операций проводится на примерах, записанных так, как если бы пользователь обращался к системе апл с терминала. Апл практически используется только в диалоговой форме. В примерах справа записывается вопрос пользователя к системе, а слева и ниже система выдает ответ. Если пользователь запишет имя объекта, то система ответит значением.

Рассмотрим запись сложения двух векторов.

12 11 13 9 7 8 + 3 4 5 вопрос пользователя
ответ системы

Следующий пример определяет сложение вектора со скаляром так, что скаляр рассматривается как вектор с одинаковыми компонентами.

$$\begin{array}{r} 978 + 5 \\ 141213 \end{array}$$

В большинстве случаев операциям над скалярными величинами соответствуют аналогичные операции над массивами, так, если известные операции над скалярами обозначаются

$= x$	обратный знак x
$!x$	факториал $x!$
$ x$	абсолютная величина $ x $
\div	обратная величина $1/x$
$*x$	экспонента e^x
$0x$	натуральный логарифм $\ln x$
$x^{.5}$	\sqrt{x}
$\sim x$	логическое отрицание $\neg x$

то к векторам те же операции применены по-компонентно, например:

1 2 6 24 ! 1 2 3 4 вычислить факториал
ответ системы

В языке есть возможность просто получить любой отрезок натурального ряда, для этого используется символ \vdash :

$$\begin{array}{cccccc} & & & & & 5 \\ 1 & 2 & 3 & 4 & 5 & \\ & & & & & 2 \\ 1 & 2 & & & & \end{array}$$

Для того чтобы узнать размерность уже существующего массива, используется унарная операция ρ :

Таб 2

3	1	7
7	10	4
6	9	1
1	6	7

9 TAB 2

4 3

Трехмерные массивы выдаются в виде совокупности матриц

ТАБ0

1 4 7
2 5 8
3 6 9
7 6 5
10 13 16
11 14 17
12 15 18
5 6 8

ρ ТАБ0

2 4 3

Для превращения таблицы любой размерности в вектор используется запятая. Например,

3 1 7 7 10 4 6 9 1 1 6 7 , ТАБ2

12 ρ , ТАБ2

В апл элементами вектора могут быть не только числа, но и символы. Например, присвоить литеральную строку символов

$A \leftarrow \text{'helloy'}$

ρA

6

Можно использовать операцию «зеркального отражения», например:

Φ 1 2 3 4
4 3 2 1

Имеется группа операций вырезания подмассивов из массивов, например, взять из вектора первые n элементов, где $n = 4$ и т. д.

$C \leftarrow \lfloor 5$
 $D \leftarrow \lfloor 5$

0 1 1 1 1
 0 0 1 1 1
 0 0 0 1 1
 0 0 0 0 1
 0 0 0 0 0

$C \circ \leq D$

(Выполняется сравнение D с каждой координатой C, результат булевский.)

$C \circ \leq D$

1 1 1 1 1
 0 1 1 1 1
 0 0 1 1 1
 0 0 0 1 1
 0 0 0 0 1

Операции «зеркального отображения» применяются к матрицам: Φ создает «зеркальный» порядок столбцов, Φ создает «зеркальный» порядок строк. Для формирования таблиц применяется бинарная операция ι , например, сформировать трехмерный массив из 24 элементов натурального ряда:

$A \leftarrow 2\ 3\ 4\ \iota\ 24$

результаты образуют две таблицы по 3 строки и 4 столбца, или,

$B \leftarrow 2\ 3\ 4\ \Phi\ \iota\ 24$

последовательность, «зеркально» перевернутая, структурируется в виде двух таблиц. Далее:

8 5 3 9

@

8 5 3 9 -1 -4 0 0

$v \leftarrow 8\ 5\ 3\ 9\ -1\ -4$

$4 \uparrow v$

$0 \uparrow v$

(т. е. «нет элементов»)

$8 \uparrow v$

При обработке матриц выполняются операции, подобные операции скалярного умножения, состоящего из двух операций: умножения и сложения.

В апл имеется целый спектр модификаций таких операций, например:

$A \leftarrow \iota 5$

$B \leftarrow 0.01 \times 1\ 2\ 5$

0.01 0.02 0.05

B

$A \circ \times B$ (B умножается на каждую координату A)

0.01 0.02 0.05
0.02 0.04 0.1
0.03 0.06 0.15
0.04 0.08 0.2
0.05 0.1 0.25

$A \circ + B$ (к B добавляется каждая координата A)

1.01 1.02 1.05
2.01 2.02 2.05
3.01 3.02 3.05
4.01 4.02 4.05
5.01 5.02 5.05

Бинарная операция ρ формирует массив, размерность которого определена первым операндом, из последовательности чисел второго операнда

$M \leftarrow 2 \ 4 \ \rho \ 2 \ 5 \ 6 \ 1 \ 4 \ 2 \ 9 \ 3$

M

2 5 6 1
4 2 9 3

Если необходимо из второй строки взять второй и четвертый столбцы, то

$M[2; 2 \ 4]$

2 3

Далее, взять из первой и второй строки второй столбец

$M[1 \ 2; 2]$

5 2

или взять первый столбец:

$M[; 1]$

2 4

или взять все элементы:

$M[;]$

2 5 6 1
4 2 9 3

Аналогичные операции выполняются над строкой символов

	$X \leftarrow 'ABCDEFGH IJK'$
	$X_i 'SAFE'$ (найти номера указанных символов в строке X).
3 1 6 5	$X [3 \ 1 \ 6 \ 5]$ (индексация)
SAFE	$X [2 \ 5 \ p \ 3 \ 1 \ 8 \ 9 \ 4 \ 2 \ 10 \ 6 \ 7 \ 5]$
CANID	
BJFGE	$X [4 \ p \ 3]$
CCCC	$X [4] \leftarrow '?'$
	X
ABC ? EFGH IJK	

При разработке расширения языка апл была сделана попытка ввести обобщенные операции над массивами, которые в качестве элемента содержали бы подмассив. Этот аппарат так и не был реализован, несмотря на внешнюю привлекательность. Это объясняется большими потерями при реализации подобных структур на ЭВМ, память которых обычно представляет вектор. Поэтому целесообразно реализовать операции над массивами, подобные тем, которые предложил Абрамс [63] когда каждому массиву ставится в соответствие дескриптор, содержащий следующую информацию:

- размерность,
- вектор, задающий длину массива по каждой координате,
- начальный номер первого элемента массива,
- вектор приращений адресов элементов по каждой координате,
- начальный адрес массива.

Подобные же характеристики массивов использованы при реализации алгола-60 [44] в операции адрес элемента массива (АЭЛ) рабочей программы и в паспортах массивов в языке алгол-68 [10]. Правила выборки элементов массива, собранные в дескрипторе,

дают возможность выполнять операции сжатия, расширения, циклического сдвига, реверса, транспонирования, обращения и т. д. только над дескрипторами, формируя новые дескрипторы с соответствующим вектором приращений и начальным номером, не производя реальных перемещений элементов для нового массива.

«Откладывая на потом» окончательное выполнение операций, можно существенно сократить количество перемещений элементов, осуществляя только в случае необходимости перемещения по дескриптору, вычисленному по суперпозиции операций [29, 30, 49].

ГЛАВА 2

ОПЕРАЦИОННЫЕ СИСТЕМЫ

В настоящее время общепризнан тот факт, что подпрограммы операционной системы, выполняющие функции непосредственного сопряжения с аппаратурой, постепенно заменяются соответствующими аппаратными блоками. Этот процесс объясняется необходимостью разрешить конфликт между все увеличивающейся сложностью, объемом математического обеспечения и обязательной высокой эффективностью ОС.

ОС — наиболее подвижная компонента МО. За период технического старения ЭВМ для нее разрабатывается сначала первый вариант ОС для быстрого введения ЭВМ в эксплуатацию, затем второй вариант — с добавлением определенных сервисных программ, и, наконец, далее возникает целая серия версий ОС, разрабатываемых квалифицированными пользователями.

Появление новых внешних устройств, изменение характеристик и состава пользователей, появление новых методов программирования — все это приводит к изменениям прежде всего в ОС.

Основной тезис этой главы заключается в том, что аппаратное ускорение ОС как средства управления состоит во введении регистров, образующих окружение, в котором должны выполняться менее привилегированные программы.

2.1. Общие положения

Операционная система представляет первичный интерфейс между пользователем и вычислительной системой. ОС содержит программы, которые восприни-

мают сигналы пользователя и управляют аппаратурой. ОС — это программное продолжение устройства управления ЭВМ. Предполагается, что в любой ЭВМ можно выделить следующие функциональные (аппаратные) блоки: устройство управления (УУ), арифметическое устройство (АУ), оперативное запоминающее устройство (ОЗУ), внешнее запоминающее устройство (ВЗУ), устройства ввода — вывода (УВВ). Даже тот пользователь, который пишет свои программы на языке, наиболее близком к аппаратуре — на автокоде, использует подпрограммы ОС и прежде всего подпрограммы ввода — вывода.

Для пользователя имеет существенное значение способ взаимодействия с системой: перфокарты (п/к), перфолента (п/л), терминал, телетайп, телефон, дисплей и т. п. Предельным случаем непосредственного взаимодействия пользователя с ЭВМ служит его работа за инженерным пультом, где пользователь может управлять любым устройством, помимо ОС. В других ситуациях права пользователя гораздо меньше, чем у ОС.

ОС обслуживает пользователя, помогает ему, защищает его от других пользователей, сообщает об ошибках и т. д. В сложных вычислительных системах соображения физической безопасности установки требуют полного отдаления пользователя от установки и наличия определенной иерархии среди персонала обслуживания, так что большинство операторов рассматриваются ОС как специализированные пользователи. В то же время пользователь, сидящий за терминалом, может взаимодействовать с программой, моделирующей режим работы с инженерного пульта, но в рамках прав, предоставленных ОС.

Вычислительная система состоит из разнородных устройств, существенно отличающихся скоростями обработки информации. ОС, как программное продолжение устройства управления, учитывает пропускные способности различных участков общего тракта обработки информации в системе.

На несколько порядков отличаются скорости электронных и электромеханических процессов, и одной из главных задач ОС является согласование требований пользователя с высокой эффективностью использования

устройств. Таким образом, ОС — это программа решения задачи оптимального управления *медленными* процессами со *слабой* связью при ограниченных ресурсах с заранее заданными критериями по режимам использования.

2.2. Режимы эксплуатации мультипрограммных вычислительных систем

2.2.1. Пакетная обработка. В режиме пакетной обработки ставится цель увеличения общей производительности системы, и для этого в ЭВМ загружается сразу несколько программ пользователей — пакет. Эффективность пакетной обработки оценивается по максимальному количеству задач, которое удастся решить за достаточно большой промежуток времени (недели, месяцы). Способом увеличения эффективности является максимальная загрузка отдельных параллельно работающих устройств системы.

В пакетном режиме вводимые программы сначала накапливаются в буферной памяти на ленте или дисках, называемой *областью ввода*, затем ОС оценивает требования задач на новые ресурсы и имеющиеся свободные ресурсы и вырабатывает график пуска задач, который должен обеспечить максимальную загрузку устройств. ОС пускает задачи на выполнение, и результаты работы задач накапливаются в буфере, называемом *областью вывода*. Наличие областей ввода и вывода дает возможность лучше совместить работу электромеханических и электронных устройств. Самым дорогостоящим устройством считается центральный процесс (ЦП), поэтому программ в области ввода должно быть столько, чтобы всегда нашлись программы, требующие работы ЦП. Для сглаживания скоростей устройств между медленными и быстрыми устройствами вставляются буферные устройства со *средними* скоростями, например, между оперативной памятью (ОП) и печатающим устройством вставляется буферная магнитная лента.

Так как требования задач меняются по мере выполнения, то ОС должна в идеальном случае непрерывно решать задачу распределения ресурсов. Чем чаще это делается, тем больше времени ЦП расходу-

ется на обслуживание. Поэтому обычно подпрограммы ОС вызываются на ЦП только в случае появления определенных событий: окончания счета на ЦП, окончания обмена, запросов обменов из задач и т. п.

В пакетном режиме время пребывания задачи в системе увеличивается, другими словами, время счета для каждой отдельной задачи в пакетном режиме больше, чем если бы задача решалась отдельно, но суммарное время решения всех задач меньше, чем сумма последовательного решения каждой отдельной задачи. Показатели пакетного режима зависят от состава смеси задач, т. е. от того, какие конкретные задачи находятся одновременно в ЭВМ.

Для этих целей задачи можно условно разделить на *счетные* и *обменные*. Счетные задачи имеют большой объем вычислений с малым количеством вводимых и выводимых данных, т. е. счетные задачи — это те, которые, в основном, загружают ЦП. Обменные задачи имеют малое время обработки, но большой объем ввода и вывода, т. е. загружают внешние устройства. Смесь из счетных и обменных задач будет удовлетворительно загружать и центральные, и периферийные устройства.

Если в смеси присутствуют задачи главным образом обменного типа, то ЦП оказывается незагруженным. В такой системе можно установить менее мощный и более дешевый процессор.

Другая ситуация характеризуется полной загрузкой ЦП, а внешние устройства работают только часть своего времени, и можно сократить парк внешних устройств.

На практике область ввода дает возможность подбирать смесь задач, учитывая их априорные характеристики, и корректировать выбранную смесь в зависимости от данных статистики, собираемой при выполнении.

Необходимость в пакетном режиме иметь для счета на ЦП сразу несколько задач увеличивает затраты оперативной памяти (ОП). Чтобы эти затраты не были слишком большими, задачи хранятся в ОП в *сжатом* виде, т. е. в ОП помещаются не все задачи целиком, а только те их части, которые нужны в данный интервал счета, остальные части задач размещаются на внешнем запоминающем устройстве (ВЗУ). Так как по

мере выполнения задачи. так или иначе используют все свои подпрограммы и данные, то для сжатых задач организуется обмен между ОП и ВЗУ. Если во время обмена по одной задаче на ЦП всегда выполняется другая задача, то система работает максимально эффективно. Если же ЦП приходится долго ждать окончания обменов для продолжения счета, то задачи сжаты слишком сильно и слишком маленькие их части размещены в ОП. В таком случае надо либо увеличить емкость ОП, либо сократить количество задач в смеси и тем самым уменьшить степень сжатия.

Так как время выполнения ОС на ЦП должно быть минимально возможным, то алгоритмы управления в ОС не могут быть очень сложными, и в трудных редко встречающихся ситуациях ОС обращается за помощью к оператору, который указывает, что нужно делать. Оператор может перезапустить систему, заново ввести пакет, может продолжить счет по контрольным точкам, отключить неисправное устройство и включить исправное, поставить другие пакеты магнитных дисков или бобины магнитной ленты и т. п. Оператор входит в контур управления, и поэтому от его действий во многом зависит производительность системы. В частности, оператор производит первичную сортировку задач и упрощает подбор смеси для ОС.

Территориально удаленные внешние устройства присоединяются к ЭВМ через линии связи. Если пакет поступает в ЭВМ по линиям связи, то такой режим называется дистанционной пакетной обработкой. В этом режиме время передачи по линии связи добавляется ко времени ввода, и это заставляет увеличивать емкость областей ввода и вывода. Сама линия связи может рассматриваться как сложное и дорогостоящее внешнее устройство, и так как необходимо увеличивать загрузку дорогих устройств, то линия связи должна использоваться мультиплексно.

2.2.2. Режим разделения времени. Диалоговое взаимодействие группы пользователей с вычислительной системой определяет режим разделения времени (РРВ). В РРВ пользователь терминала получает ответы на *не очень сложные* запросы в течение нескольких секунд. Цель РРВ состоит в достижении максимальной скорости одновременного счета задач всех пользователей.

В результате у всех пользователей создается иллюзия единоличного владения вычислительной системой. С увеличением числа пользователей время ответа на запросы увеличивается, это эквивалентно для пользователя переходу на более медленную машину, действительно на один запрос приходится P/N мощности, где P — общая мощность системы, а N — количество пользователей с простыми запросами. Аналогичная ситуация возникает, если пользователь генерирует сложный запрос, эквивалентный M простым запросам, так что на запрос будет приходиться $P/(M \times N)$ мощности.

В диалоговом режиме пользователю удобно вводить, редактировать и отлаживать. При пуске на счет пользователь уже психологически подготовлен к такому периоду ожидания, который предполагается сложностью его счета.

Стоимость обслуживания в РРВ выше, чем в пакетном режиме, так как устройства могут простаивать, главное, чтобы задачи считались почти одновременно.

Пользователь может всегда выдать такой запрос на счет ЦП, который не удастся выполнить за несколько секунд, поэтому порядок работ пользователей в РРВ регулируется приоритетами. Приоритеты имеют тенденцию увеличиваться при увеличении времени пребывания пользователя в системе. В начале обычно всем задачам пользователей выделяется минимальный квант времени ЦП, в течение которого все задачи выполняются последовательно, затем тем задачам, которые не выполнялись, выделяется больший квант времени и уменьшается приоритет. Для задач с одинаковым квантом приоритет может быть больше у задачи, дольше находящейся в системе. Если задача при вводе требует большой временной интервал работы ЦП, то такой задаче присваивается низкий приоритет. Те задачи, например, которые вводятся в систему РРВ в пакете, образуют отдельную группу, называемую *фоновой*. На ЦП всегда пускается одна из имеющихся в распоряжении фоновых задач, если нет ни одной задачи с терминала, нуждающейся в ЦП.

Непрерывная смена задач на ЦП в РРВ, вызванная необходимостью осуществить «одновременный» счет задач всех пользователей, приводит к высокому темпу

обмена с ВЗУ. Если в режиме пакетной обработки задача, выполняющаяся на ЦП, как правило, не прерывается до тех пор, пока она сама не запросит такой обмен с ВЗУ, окончания которого она должна ожидать, то в РРВ задача, отработав фиксированный квант времени на ЦП, прерывается для того, чтобы обеспечить равномерный счет по всем задачам.

Задачи пользователей с терминала обычно не могут разместиться целиком в ОП, так как они используют библиотеки, трансляторы, обслуживающие программы и т. п., что приводит к высокому темпу обмена с ВЗУ и заставляет учитывать характеристики конкретного ВЗУ. В настоящее время наибольшее распространение получили магнитные диски (МД), для которых время обращения почти в 10^4 раз больше времени обращения к ОП, поэтому в РРВ ОС накапливает очередь запросов к МД, учитывает угол поворота МД и движение головки. Несмотря на принимаемые меры, системы РРВ характеризуются низким коэффициентом загрузки ЦП, порядка 10—30%. Сокращение потока обмена между ОП и ВЗУ можно получить введением *массового* оперативного запоминающего устройства (МЗУ), у которого емкость и время обращения примерно в 10 раз больше, чем у ОП. Обмены между ОП и МЗУ всегда выполняются такой группой слов, чтобы среднее время обращения к одному слову МЗУ приблизительно равнялось времени обращения к ОП. Обычно в группе содержится 8—16 слов. Тем не менее резкое увеличение количества и сложности запросов пользователей, например, в середине рабочего дня, в конце месяца или года, несомненно увеличивает время ответа системы и иногда заставляет ОС послать сообщение непривилегированным пользователям об отключении от системы на некоторый длительный интервал времени.

2.2.3. Режим реального масштаба времени. В режиме реального масштаба времени (РМВ) пользователю необходимо получить решение задачи не позже, чем за заданный интервал времени, и ОС должна соответствующим образом распределить ресурсы. Конкретная система РМВ работает обычно в ограниченном классе задач, для которого задан диапазон времен ответов $T_{\text{отв}}$. Для пакетного режима, характеризующе-

гося максимальной загрузкой устройств, $T_{\text{отв}}$ может составлять несколько дней и имеет тенденцию расти.

Вычислительные системы можно разделить на группы в зависимости от величины времени ответа $T_{\text{отв}}$:

- 1) системы пакетной обработки — 1—2 дня;
- 2) коммерческие системы запрос — ответ (продажа билетов, торговля и т. д.) — 0,5—1 минут;
- 3) интерактивные человеко-машинные системы (диалог, поиск и т. д.) — 0,5—5 секунд;
- 4) промышленные системы управления, в которых ВС включена в контур управления — 5—500 миллисекунд.

Операционная система реального времени (ОСРВ), подобно другим ОС, поддерживает общее управление системой: планирует выполнение программ, устанавливает приоритеты работ, обслуживает прерывания, связывается с оператором системы и выполняет задачи учета и статистики. Конкретные параметры ОСРВ существенно определяются размерами системы и целями проектирования. Опыт использования существующих коммерческих ОС общего назначения показывает, что они по многим параметрам не удовлетворяют требованиям реального времени: дефрагментированы, очень сложны, и, в целом, неэффективны по времени ответа.

Несмотря на все перечисленные недостатки ОС общего назначения и принимая во внимание, что главное требование системы РМВ — это высокая надежность, так как в случае сбоев и отказов время ответа может стать непредсказуемо большим, единственно правильный путь построения систем РМВ — это использование ОС, поставляемых разработчиком ВС.

Любая попытка пользователя сделать свою собственную ОС или спроектировать нестандартную ОС для конкретной установки приводит к потере совместимости со стандартной системой общего назначения и к невозможности отыскать все ошибки ОС в одиночку, ввиду ее сложности, и, следовательно, к плохой и ненадежной работе системы.

На практике для вырожденных случаев примитивных ЭВМ пользователь иногда пишет простую управляющую программу, которая устойчиво работает в весьма ограниченных режимах, ввиду непреодолимых трудностей полной отладки.

Программы пользователя реального времени должны обладать мобильностью по отношению к специализированным терминальным устройствам. Это требование к ОСРВ крайне важно, так как в системах реального времени применяются разнообразные внешние устройства, и если меняется тип внешнего устройства, то программы пользователя меняться не должны, если только они не моделируют работу этого внешнего устройства.

2.3. Программно-аппаратные средства операционных систем

2.3.1. Прерывание программ. Аппаратура прерывания программ ориентирована на взаимодействие слабо связанных процессов ОС. Она является частью блоков, выполняющих общие функции вызова подпрограмм и возврата из них с сохранением и восстановлением состояния вызывающей программы. Специфика программ обработки прерываний состоит в их привилегированности. Учитывая конкретные особенности слабо связанных процессов ОС, затраты аппаратуры можно ограничить регистрами прерывания и маски, блоками выполнения команды установки и снятия блокировки прерывания и команды синхронного считывания (TEST & SET) [26].

Аппаратура такого объема не всегда удовлетворяет требованиям прикладных программ, в которых могут существовать *сильно связанные* процессы и для которых требуется дополнительная аппаратная поддержка.

Прерывание программ дает возможность осуществить синхронизацию процессов через общую ячейку памяти, роль которой играет регистр прерывания. Над этой общей ячейкой может выполняться группа специальных операций: операция прерывания, операция считывания и операция записи.

Операция прерывания состоит в том, что если хотя бы один разряд регистра прерывания находится в единице и для него соответствующий разряд регистра маски находится тоже в единице, т. е. маскирует его, и блокировка прерывания не установлена, то между командами работающей программы вставляется аппаратно команда вызова программы обработки прерыва-

ния с сохранением состояния предыдущей программы. Программа обработки прерывания выполняет над регистром прерывания операцию считывания, которая формирует номер самого приоритетного разряда из всех находящихся в единице и маскированных разрядов регистра прерывания. Номер этого разряда определяет программу, обслуживающую данное прерывание. В обслуживающей программе выполняется операция записи нуля в разряд регистра прерывания, номер которого равен номеру обслуживающей программы.

Операции прерывания, считывания и записи выполняются последовательно друг за другом. В операции прерывания устанавливается блокировка прерывания, что дает возможность однозначно выбрать обслуживающую программу и сохранить состояние прерванной программы. Когда выбрана обслуживающая программа и погашен разряд регистра прерывания, то обычно сбрасывается и блокировка прерывания, за исключением некоторых случаев, в которых реакция на прерывание должна занимать минимально возможное время. В этих случаях короткие программы реакции на прерывание выполняются в режиме с блокировкой прерываний. Установка блокировки прерываний приводит к отклонению процессора от источника внешних сигналов, поэтому для вычисления времени ответа в режиме реального времени необходимо время максимального интервала установки блокировки прерывания добавить к времени всех программ обслуживания прерывания.

Всякая смена программ на процессоре связана с сохранением и восстановлением состояния программ, что снижает производительность процессора, поэтому чем реже появляются прерывания, тем меньше потери в производительности. Потери в производительности тем больше, чем больше мощность процессора, так как достижение высокой скорости выполнения операций основано на К-структуре (конвейер), рассматриваемой в третьей главе. Чем больше уровней в К-структуре, тем больше потери при прерывании. Прерывание по своей природе не связано с прерываемой программой, поэтому при вызове программы обработки прерывания объем сменяемой информации будет максимальным по сравнению с вызовами программ, обменивающихся параметрами.

Прерывание дает возможность мультиплексно использовать процессор для нескольких программ, поэтому в тех случаях, когда необходимо получить максимальную скорость выполнения одной программы, прерывания должны обслуживаться менее мощными периферийными процессорами. В быстродействующих центральных процессорах прерывания должны быть только в редких случаях.

2.3.2. Синхронизация процессов. Базисом логики взаимодействия процессов ОС служат семафоры [15]. Семафоры представляют общие переменные s взаимодействия процессов, над которыми выполняются операции P и V .

P -операция закрывает семафор, т. е. S уменьшается на единицу, если $S \geq 1$. Если $S = 0$, то P -операция не может завершиться, и процесс переходит в состояние ожидания или «зависает» на P -операции. V -операция «открывает семафор», т. е. увеличивает S на единицу.

Синхронизация процессов строится на разделении пространства развития процессов на две области. В первой области, называемой *областью цикла*, процесс развивается циклически независимо от других процессов, во второй области, называемой *критической областью*, процесс может быть только весьма ограниченное время и всегда только один. Для этого с каждой критической областью связываются семафоры. В каждый данный момент времени в критической области может находиться только один процесс. Критические области — это участки синхронизации процессов.

Прохождением процессов через критические области управляют семафоры. Если критическая область открыта, то процесс, входящий в нее, закрывает семафор, и никакой другой процесс уже не может войти в критическую область. Когда процесс покидает область, то он открывает семафор. Те процессы, которые находились в стадии выполнения P -операции над семафором S , теперь, так как семафор открыт, разыгрывают между собой приоритет, и один из них входит в критическую область. Логика такого взаимодействия процессов дает возможность сосредоточить внимание только на критической области и не учитывать взаимодействие процессов вне ее.

Аппаратная реализация операций Р и V опирается на команду синхронного считывания. — CC (Test & Set). Синхронное считывание состоит в выполнении двух неразрывных обращений к ОП. Если два процессора пытаются одновременно выполнить операции синхронного считывания, то эти операции выполняются последовательно. Операция синхронного считывания состоит в выполнении считывания старого содержимого ячейки и записи в него нового значения. Запись — это способ маркирования ячейки, маркер сообщает, что к ячейке было обращение по считыванию. Обычно после команды синхронного считывания выполняется команда условной передачи управления, которая передает управление по нулю на продолжение процесса, а по единице — на перевод процесса в состояние ожидания. Из состояния ожидания процесс выводится операцией открытия семафора.

Понятие процесса не отождествляется с процессором, процесс можно считать некоторой областью памяти, в которой запоминается его состояние. Каждому процессу соответствует область памяти, называемая информационным полем процесса — ИПП. В ИПП хранится текущее состояние регистров процессора, если процесс не выполняется на процессоре. Если процесс выполняется на процессоре, то в его ИПП хранится признак выполнения, а текущее состояние регистров процесса соответствует текущему состоянию регистров процессора. В начальном состоянии все процессы ожидают освобождения процессора, затем, когда процессор освобождается, ИПП самого приоритетного процесса переносится на регистры процессора.

Если процесс на процессоре выполняет команду синхронного считывания и получает маркер того, что семафор закрыт, то состояние регистров процессора запоминается в ИПП с сохранением причины ожидания. Команда синхронного считывания является внутренней командой привилегированных подпрограмм ОС, соответствующих операциям Р и V. При выполнении этих подпрограмм должны быть запрещены прерывания с помощью команды установки блокировки прерывания, так как операции Р и V должны представлять непрерываемые участки программ. Блокировка прерывания дает возможность рассматривать последовательность

команд как одну непрерывную команду. Если при этом одновременно команда синхронного считывания запрещает параллельные вычисления, то команды процессов выполняются *строго* последовательно — от начала до конца. Внутри таких *строгих* последовательностей могут находиться команды сохранения состояния процессоров, если они используют общие ячейки памяти. Так как время выполнения таких *строгих* последовательностей заранее известно, то одновременно с установкой блокировки прерывания включаются *часы*, которые задают максимальный интервал, по истечении которого присутствие блокировки прерывания воспринимается как поломка аппаратуры. В такой ситуации часы освобождают процессор и, как правило, пытаются вновь запустить процесс с какого-то предыдущего момента.

Сохранение состояния системы связано со всеми процессами, поэтому такие действия может выполнять только механизм ОС, общий для всех процессов. Обычно ОС периодически сохраняет состояние тех изменений, которые произошли в системе за предыдущий интервал работы без сбоя. Выбор периода сохранения определяется средней величиной интервала между сбоями и требованиями к надежности системы.

Синхронизация процессов с помощью семафоров не всегда удобна. Точки синхронизации могут быть распределены в произвольных местах программ, тогда как хотелось бы иметь команды синхронизации, собранные вместе. С этой целью можно рассмотреть способ синхронизации процессов с помощью подпрограмм: «послать сообщение», «ждать сообщение», «послать ответ», «ждать ответ» (рис. 4). Например, рассмотрим синхронизацию пары процессов. Пусть выполняются два процесса i и j , взаимодействующие между собой. Если процесс i «послал сообщение» в момент t_2 , а процесс j «ждет сообщение» в момент t_3 , то, так как $t_2 < t_3$, процессы будут выполняться без задержек на синхронизацию. Если процесс j «ждет сообщение» в момент t_1 , где $t_1 < t_2$, то процесс j остановится на интервал $t_2 - t_1$, т. е. до момента t_2 , в который процесс i «послал сообщение». Далее, пусть, например, процесс i в момент t_4 «ждет ответ», а процесс j «послал ответ» в момент t_5 , где $t_5 > t_4$, тогда процесс i остановится на

интервал $t_5 - t_4$. Если бы процесс i стал «ждать» ответ» в момент t_6 , где $t_6 > t_5$, то никакой задержки нет.

Дальнейшее расширение понятий синхронизации процессов связано с модификациями входа в подпрограмму и передачи параметров. Если вызов программы

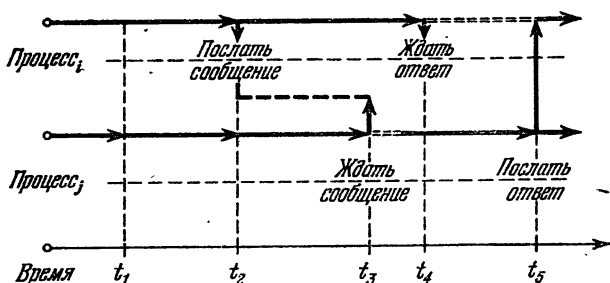


Рис. 4. Пример передачи сообщений между процессами.

требует синхронизации по передаваемым параметрам, то эта синхронизация осуществляется внутри вызываемой подпрограммы. Для этих целей обычно служат три типа связей подпрограмм:

1) *процедурная связь*, когда оператор вызова заменяется на выполняемую программу, которая исчезает после выполнения;

2) *корутинная связь*, когда оператор вызова передает управление на программу, которая, начав выполняться, может снова вернуть управление вызывающей программе, следующее обращение от вызывающей программы к вызываемой продолжит выполнение вызываемой с момента последнего возврата;

3) *процессная связь*, когда операторы вызова программы есть подпрограммы обмена сообщениями, которые либо передают сообщение и разрешают продолжить работу вызывающей программы, либо задерживают продолжение вызывающей программы до ответного сообщения от вызываемой программы.

В первых двух типах точки связи фиксированы для последовательного выполнения команд, в третьем типе команды могут выполняться параллельно. В зависимости от трех типов связи формируются правила взаимодействия модулей, состоящих из программ. Модули

при своем взаимодействии обмениваются параметрами, которые могут иметь любой из трех типов. Модули работают совершенно независимо, за исключением опроса состояния готовности параметров. Готовность параметров устанавливается подпрограммами, обслуживающими каждый из типов. Если параметр передается по процедурной или корутинной связи, то вызывающий модуль будет ждать готовности параметра, полученной после полного или частичного выполнения вызванной программы. При процессной связи готовность параметра устанавливается сообщением от вызванной программы.

Таким образом использование переменных в качестве фактических параметров, *закрывает* их в определенных ситуациях от дальнейшего использования. Для того чтобы уменьшить количество закрываемых переменных, они делятся на входные и выходные. Выходные переменные формируются вызываемой программой, и поэтому закрываются. Входные переменные не затрагиваются вызываемой программой, и поэтому могут быть использованы в дальнейшем. При такой передаче параметров возможны взаимоблокировки подпрограмм, вызванных цепочками ожидания готовности. Но такие цепочки могут обнаруживаться на стадии трансляции.

Типичная ОС настоящего времени содержит подпрограммы создания процессов, их объединения и разветвления, и, наконец, прекращения. Такие подпрограммы представляют набор операций над данными, образуемыми системными таблицами, см. раздел 2.3.3.

Особое место в ОС занимает процесс, связанный с отсчетом времени. Этот процесс обязательно периодически захватывает процессоры и тем самым предотвращает узурпацию процессора каким-либо одним процессом, который мог заиклиться в результате ошибки.

Задача распределения процессоров среди процессов является частью задачи распределения ресурсов, которую решает ОС. Ресурсы составляют не только аппаратные устройства, но и программы, семафоры, таблицы и т. д. Неправильное распределение ресурсов может привести к взаимоблокировке процессов [105]. называемой *тупиком*. Так, например, два процесса А

и В заблокируют друг друга, если процесс А захватил ресурсы Р, процесс В захватил ресурсы Т, и общий объем ресурсов равен $T + P$, если процессу А для продолжения нужны дополнительные ресурсы РА и процессу В ресурсы ТВ, хотя $P + RA < T + P$, $T + TB < T + P$. В тупике может находиться группа процессов, и тупик может существовать в виде непрерывной передачи ресурсов между процессами.

Простейший алгоритм распределения ресурсов рассчитывает выдачу очередной порции ресурсов по запросу с учетом максимального количества ресурсов, которое нужно, чтобы завершился хотя бы один процесс. Если такой процесс найдется, тогда те ресурсы, которые он освобождает в момент окончания, передаются всем остальным процессам, и отыскивается еще какой-либо процесс, который может окончиться с учетом этих освободившихся ресурсов. Ресурсы, которые освобождаются после окончания второго процесса, опять отдаются всем остальным процессам и т. д. Если при этом окажется, что все процессы могут закончиться, то запрос на ресурс удовлетворяется.

Вопросу тупиковых ситуаций посвящено большое количество работ [38], но пока еще нет устоявшихся программных реализаций, которые могли бы быть подержапы аппаратурой.

Одним из подходов к решению задачи тупиковых ситуаций служат попытки обнаружить уже возникший тупик. В общем случае обилие форм тупиковых ситуаций очень усложняет обнаружение этого факта. Если бы ошибки алгоритмов распределения ресурсов приводили к заикливанию процессоров на одной команде, то тогда можно было бы с помощью аппаратуры фиксировать ситуацию, в которой время выполнения команды значительно превышает установленный заранее интервал. Далее можно было бы говорить об аппаратной фиксации неизменного состояния не только регистров, но и ячеек памяти, и, наконец, в общем случае можно было бы потребовать дополнительного периодического ответа от программы, проверяющей наличие тупика. Этот ответ должен изменять состояние регистра, если же состояние регистра неизменно, то это может служить признаком тупиковой ситуации.

В случае систем РМВ накладываются дополнительные ограничения на время обнаружения тупиков. Типичной реакцией на обнаруженную тупиковую ситуацию служит возврат захваченных ресурсов одним из процессов, исключение этого процесса из счета и попытка продолжить счет для оставшихся процессов.

В таких ситуациях облегчает положение наличие контрольных точек у процесса, так как можно осуществить возврат состояния исключенного процесса к предыдущей контрольной точке и продолжить счет при освобождении ресурсов.

2.3.3. Распределение и защита памяти. Параллельное выполнение процессов осуществляется на основе распределения и защиты памяти одного процесса от непредусмотренных действий другого для каждого процесса. Память одного процесса защищается от непредусмотренных действий другого. Защита построена по иерархическому принципу. На самом нижнем уровне иерархии находятся прикладные программы, действия которых определяются теми границами, которые установлены системными программами.

В свою очередь ОС как любая большая программа также имеет иерархическую структуру, на самом верхнем уровне иерархии находятся подпрограммы ядра ОС, которые тщательно отлажены и составляют базис для построения следующих уровней защиты. Среди этих программ важное место занимает программа, которая создает для каждого процесса свою «бесконечную» виртуальную память.

Вообще под виртуальной памятью понимается возможность использовать в программе очень большое адресное пространство, значительно превышающее диапазон адресов оперативной памяти реальной машины, с автоматическим выполнением обменов между оперативной и внешней памятью, на которую отображается адресное пространство пользователя. Обычно большой адрес делится на две части: адрес сегмента и адрес элемента внутри сегмента. В дальнейшем рассматривается сегментированная листовая память [89].

Сегмент — это именуемая логическая единица оперативной памяти, представленная вектором произвольной длины. Лист — это квант обмена между внешней и оперативной физической памятью. Программа вир-

туальной памяти организует таблицы, с помощью которых выполняется отображение сегментированной виртуальной памяти на физическую.

Дальнейшее описание взаимодействия процессов и памяти строится на основе упрощенной модели ОС [91]. В этой модели работа ОС описывается несколькими системными таблицами. Центральное место занимает таблица процессов, номер строки которой соответствует номеру процесса. Процессоры представлены в виде таблицы очередей. Номер строки таблицы очередей идентифицирует процессор. Для каждого процесса вводится специальная таблица, называемая информационным полем процесса, которая содержит: адрес команды, с которой должна быть продолжена программа процесса, содержимое регистров процессора, начальный адрес таблицы сегментов.

Каждый процесс обладает своей собственной виртуальной сегментированной памятью. Виртуальная память каждого процесса представлена таблицей сегментов и таблицами листов. Для таблиц сегментов и листов выделяются виртуальные сегменты с одними и теми же номерами во всех процессах. Остальные номера сегментов внутри каждого процесса распределяются независимо. При работе с общими данными согласование разных номеров сегментов разных процессов, относящихся к одному и тому же общему сегменту, осуществляется через таблицу общих сегментов, размещаемую вместе с таблицами сегментов и листов процессов.

Как показано на рис. 5, строки n , i , 1 таблиц процессов А, Б и В определяют один и тот же общий сегмент, описываемый строкой S_j в таблице общих сегментов. Строка j содержит ссылку на общую таблицу листов ТЛ_j. Для синхронизации обращений процессов к общему сегменту в соответствующих строках таблиц сегментов имеется специальный признак синхронизации. Процессу, имеющему один из указанных сегментов, разрешается обращаться к общему сегменту через строку j таблицы общих сегментов, если признак синхронизации в строке таблицы сегментов процесса и признак синхронизации в строке таблицы общих сегментов одновременно равны единице. В противном случае процесс переходит в состояние ожидания.

В строке таблицы процессов (рис. 6) находится адрес начала информационного поля процесса, в котором хранится адрес начала таблицы сегментов процесса. В строке таблицы процессов хранится также адрес строки таблицы очередей. Каждая строка таблицы очередей соответствует процессору. Если, например, процессы $1, k, m$ запрашивают процессор l , тогда l -я строка таблицы очередей процессоров хранит 1 как номер первого процесса в очереди и m как номер

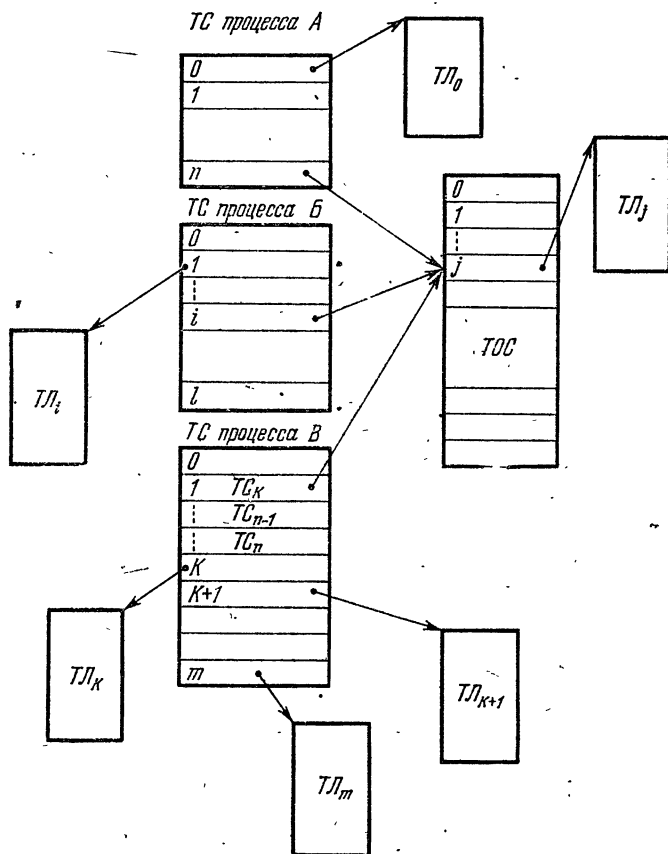


Рис. 5. Таблица сегментов ТС и таблица листов ТЛ для процессов, имеющих общий сегмент S_j в таблице общих сегментов ТОС.

последнего. Общая очередь процессов к одному процессору поддерживается за счет того, что для каждого процесса, стоящего в очереди, в строке таблицы процессов хранится номер процесса, следующего за ним, а номера процессов для начала и конца очереди хранятся в строке таблицы очередей, соответствующей процессору.

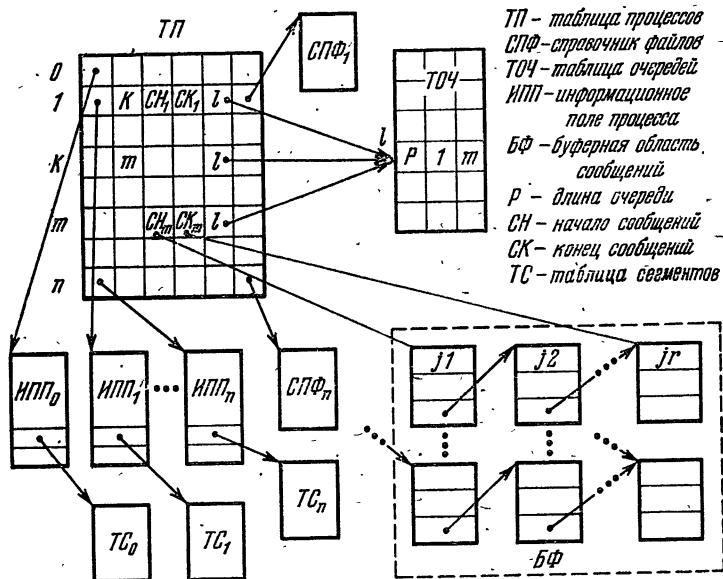


Рис. 6. Организация таблиц модели ОС.

Работа процессов сопровождается обменом сообщениями. Если процесс посылает сообщение другому процессу, то в строке таблицы процессов, соответствующей процессу, которому направляются сообщения, помещается адрес первого и адрес последнего сообщения. В первом сообщении хранится адрес второго и т. д. Все сообщения размещаются в буферной области. Кроме того, в каждом сообщении хранится номер процесса, пославшего сообщение.

При освобождении l -го процессора выбирается первый процесс из очереди, номер которого хранится в

строке таблицы очередей. По этому номеру извлекается строка таблицы процессов, в которой находится адрес начала информационного поля процесса. Из информационного поля перемещается содержимое регистров на процессор, в частности, устанавливается адрес начала таблицы сегментов и адрес первой команды программы процесса, по которому выполняется обращение к памяти с преобразованием виртуального адреса в физический через таблицы сегментов и листов, в которых хранятся признаки разрешения доступа по выполнению, по считыванию, по записи и т. д.

Широко известная аппаратура распределения и защиты памяти содержит ассоциативные регистры преобразования виртуальных адресов в физические, регистры параллельных косвенных обращений и регистры формирования функций расстановки, т. е. всех обычных средств, служащих для ускорения поиска в таблицах, размещенных в линейной памяти. Основой высокой производительности процессоров с виртуальной памятью служат магнитные барабаны с большой емкостью и скоростью обмена, магнитные диски с головкой на каждую дорожку и другие современные внешние устройства.

Строки рассматриваемых таблиц составляют частные случаи реализации дескрипторов, используемых при конкретной обработке динамических массивов [2]. Использование дескрипторов массивов, ячейки которых могут дополнительно содержать минимальную информацию о типе элементов массива, дает возможность создать однородную структуру механизма распределения памяти, рассматриваемой как набор векторов.

Дескрипторы, передаваемые в качестве параметров в программах, определяют возможности доступа к областям памяти. Программа, использующая дескриптор, может его изменить только таким образом, который определен передающей программой. Права доступа к дескриптору передаются программе вместе с дескриптором. Существенно, что программа не может превысить тех прав, которые ей предоставлены, по обращению к дескриптору, и программа может передать дескриптор дальше только в том виде, который задан правами по передаче. Начальный вид основных дескрипторов определяется системными программами [92].

2.4. База данных

Работа современных вычислительных систем основывается на организации хранения большого количества данных. Данными является числовая, символьная, двоичная информация. Данным обычно противопоставляются обрабатывающие программы, но в общем случае один и тот же информационный объект может быть то данными, то программой. Например, текст на входном языке служит данными для транслятора, результатом работы которого является обрабатывающая программа.

Хранение — важная фаза процесса обработки информации. На этой фазе предполагается использование оперативного запоминающего устройства, внешнего запоминающего устройства и устройств ввода/вывода. Если емкость оперативного устройства существенно ограничена и операции хранения в нем достаточно просты, то внешние устройства обладают гораздо большими возможностями и требуют сложных операций хранения реализуемых специальными программами. Эти программы должны учитывать характеристики устройств и конкретные требования пользователей, поэтому они снабжаются большим количеством параметров. Указывая конкретные значения параметров, пользователь приспособливает программы для своих целей. Обычно такие программы организованы в виде библиотеки стандартных программ, общих для всех пользователей. Появление новых более совершенных устройств внешней памяти и увеличение сложности задач, решаемых пользователем, приводит к ситуациям, в которых пользователей не устраивают возможности параметризованных программ. Действительно, невозможно хранить в вычислительной системе программы для всех случаев применений, поэтому пользователю предлагается язык высокого уровня и транслятор.

Использование языков, ориентированных на операции хранения данных, является *первым* принципом баз данных.

Увеличение степени универсальности обработки определяется прогрессом в технологии аппаратуры, который постепенно снижает ее стоимость. Применение трансляторов с языков высокого уровня стало возможным только при наличии дешевых устройств памяти.

Соответственно применение языков обработки данных стало возможным с появлением дешевых устройств внешней памяти прямого доступа. Так же как применение языка алгол-60 скрывает от пользователя многие детали конкретной машины, так и применение языка данных перемещает пользователя на новый более высокий уровень абстракции. Пользователь работает с обобщенным внешним устройством, которое в большей степени учитывает его интересы.

Первыми попытками введения определенных обобщений внешних устройств явилось использование разнообразных параметров программ обмена. Параметры дали возможность работать не с конкретным физическим устройством, а с представителем устройств определенного типа, например, можно говорить об устройстве типа «ленты» или типа «оперативной памяти». Всю память вычислительной системы можно рассматривать как последовательность пронумерованных ячеек или вектор. При обращении к ячейке используется номер элемента вектора. С другой стороны, память можно считать множеством, каждый элемент которого хранит свой номер, тогда обращение к памяти выполняется с помощью сравнения требуемого номера с содержащимся элементом.

Программы с параметрами, организованные в библиотеки стандартных программ, в принципе приводят к дублированию, так как, начиная с некоторого момента, появляются программы, которые мало отличаются друг от друга. Для программ пользователей, ориентированных на обработку больших массивов хранимых данных, дублирование прежде всего было характерно для самих данных, потому что объем данных мог значительно превосходить объем обрабатывающих программ. Структуры данных прямо связывались с логикой программы, поэтому увеличение объема и изменение структур данных приводило к изменению программ.

Централизованное хранение информации, которое дает возможность сократить дублирование как данных, так и программ, и обеспечить независимость данных, т. е. неизменяемость программ при увеличении объема данных, можно считать вторым принципом баз данных. База данных (БД) в принципе универсальна, т. е.

она не содержит информацию, предназначенную для одного какого-либо конкретного применения. БД содержит общую информацию, которая служит для построения информационной модели каждого конкретного применения.

Типичным режимом взаимодействия пользователя с БД является режим разделения времени, хотя существуют и режим пакетной обработки, и режим реального масштаба времени. Принципиально важно обеспечить одновременный доступ нескольких пользователей, так как база данных централизует данные.

Комплекс программ, служащий интерфейсом между пользователем и хранимыми данными, образует *система управления базой данных* (СУБД).

СУБД дает возможность:

- 1) организовать интегральное множество данных, доступное широкому кругу пользователей;
- 2) поддерживать достоверность и непротиворечивость данных;
- 3) обеспечить централизованные средства сохранения тайны;
- 4) продлить неопределенно долго срок существования данных.

Последний пункт подчеркивает чрезвычайно высокие требования к надежности всей системы и, особенно, программ.

Взаимодействие пользователя с СУБД строится на основе некоторой модели или абстракции данных. Модели можно поделить на три группы:

- 1) иерархическая или древовидная,
- 2) сетевая,
- 3) реляционная.

Наиболее широко распространенная абстракция — это представление объектов внешнего мира в виде определенной иерархии. Наиболее общая абстракция — это представление внешнего мира как отношений объектов или реляций.

Сетевая модель занимает промежуточное положение, в ней, с одной стороны, вводится достаточное универсальное представление данных, имеющее широкое применение, а с другой стороны, учитываются возможности современных устройств прямого доступа, именно поэтому следующий раздел посвящен сетевой модели.

2.4.1. Сетевая модель данных. Сложность проблем проектирования и внедрения баз данных потребовала создания специальной международной организации — рабочей группы по банкам данных РГБД КОДАСИЛ, результатами работ которой являются руководящие документы по базам данных.

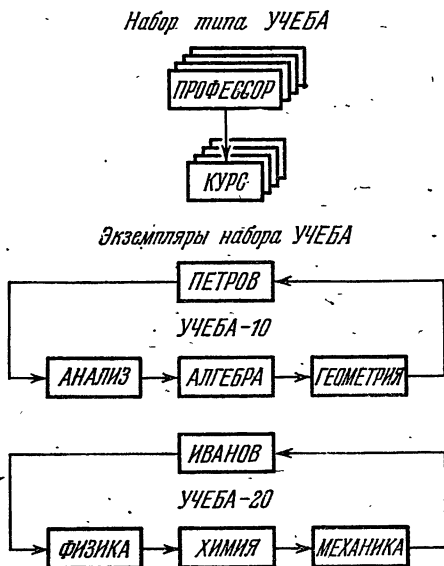


Рис. 7. Пример набора типа УЧЕБА.

Рекомендации РГБД прежде всего относятся к классификации языков. Предлагается рассматривать три отдельных языка:

- 1) язык определения данных (ЯОД);
- 2) язык манипулирования данными (ЯМД);
- 3) включающий язык (ВЯ) обработки, в который входит язык манипулирования данными как подязык.

Описание БД составляется независимо от программ пользователя на ЯОД. Это описание называется *схемой*. Схема транслируется в таблицы, содержание которых используется при выполнении операторов ЯМД. Основу взаимодействия программы с БД составляет рабочая область пользователя (РОП).

Основной структурный объект БД, предлагаемый РГБД, называется *записью*. В общем понятие записи

как элемента файла служило прототипом этого термина. Запись — неделимая единица БД, операторы ЯМД могут обращаться к записи, задавая ее имя. Запись состоит из элементов. Для того, чтобы обратиться к элементу, надо сначала извлечь запись из БД, а затем

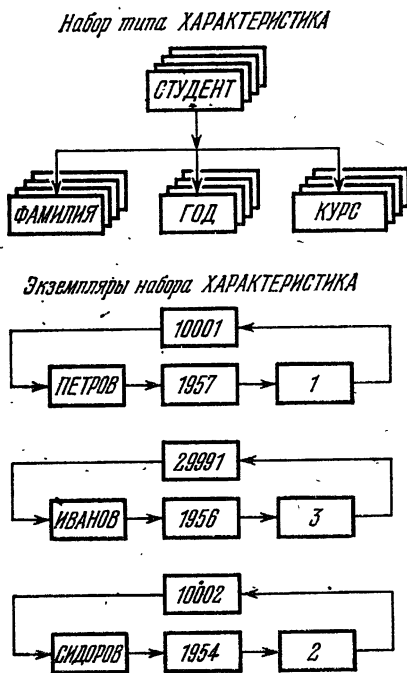


Рис. 8. Пример набора, содержащего несколько разных типов записей.

из записи элемент. Из записей можно строить произвольные структуры, называемые *наборами*. Набор — это упорядоченная совокупность записей (см. рис. 7—10). Наборы дают возможность построить сложные сетевые структуры, опираясь на рекомендации РГБД КОДАСИЛ. Схема БД включает как описание записей, так и наборов.

Сетевая модель данных во многом учитывает требования эффективности. Хотя схема не содержит ссылок на физические устройства или участки носителя,

тем не менее она содержит указания о желательном размещении групп записей. Одно из таких указаний вводится понятием *области* — совокупности записей, которые должны быть размещены вместе. Вообще

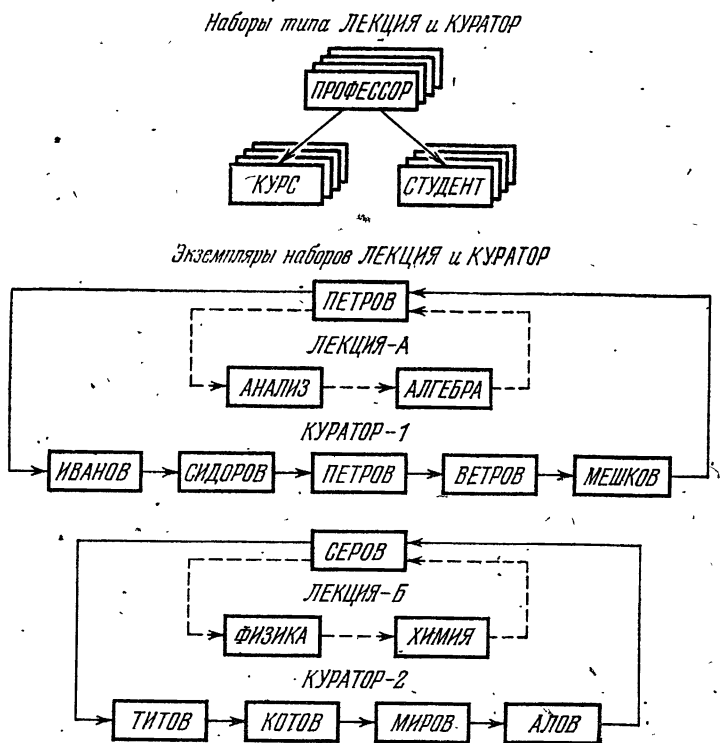


Рис. 9. Пример записи, являющейся владельцем двух наборов.

схема составляется специальным системным программистом — администратором базы данных (АБД), учитывающим противоречивые потребности пользователей и имеющиеся ресурсы.

Администратор базы данных разрабатывает информационную структуру, соответствующую реальным объектам и характеру их изменений.

Для того чтобы учесть специфические интересы конкретного пользователя, составляется описание лишь

той части БД, называемое подсхемой, которая интересует именно его, причем частичное описание может во многом отличаться от полного. Все данные хранятся в базе данных в соответствии со схемой, и при передаче в рабочую область пользователя (РОП) происходит распаковка этих данных так, как это требуется

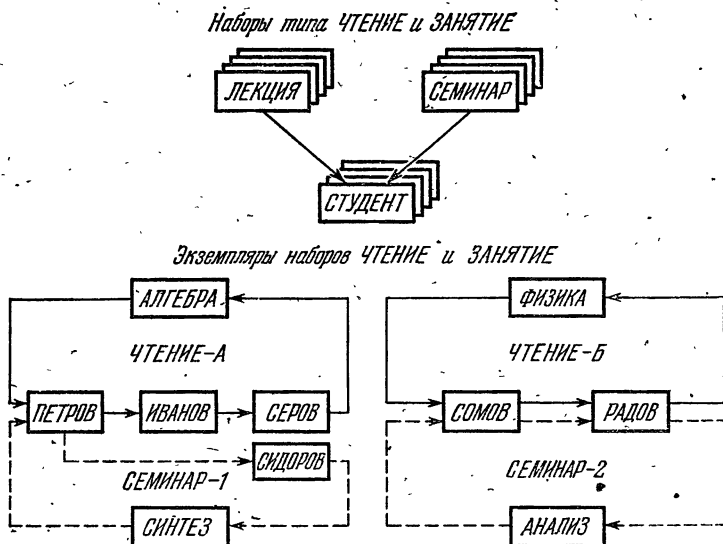


Рис. 10. Пример записи-члена, экземпляры которой входят в два разных набора.

подсхемой. Для пользователя вся база данных представляется такой, какой она описана в подсхеме.

Сетевая модель строится по следующему принципу. Во-первых, часть записей выделяется в отдельную группу, обращение к которой происходит прямо из системы. Такие записи называются сингулярными. Далее записи связываются между собой следующим образом: один экземпляр записи некоторого типа выделяется как экземпляр записи владельца, и произвольное количество экземпляров записей других типов, называемых экземплярами записей членов, объединяется с владельцем в экземпляр набора. Начальный и единственный экземпляр владельца *сингулярного на-*

бора образует систему управления базой данных (СУБД). По запросу пользователя в этом наборе можно найти экземпляр записи-члена, который является владельцем для следующего набора и т. д.

Каждая запись идентифицируется ключом базы данных. Ключ БД — целое число, которое дает возможность отличить одну запись от другой.

Когда в программе выполнится оператор ЗАПОМНИТЬ, то генерируется новый ключ БД, который существует вместе с записью до выполнения оператора ВЫЧЕРКНУТЬ.

Каждому набору в схеме приписывается определенная упорядоченность экземпляров записей-членов:

- 1) в возрастающем (убывающем) порядке значений элементов данных,
- 2) в порядке включения в набор новых экземпляров записей-членов,
- 3) в порядке, заданном СУБД.

Одни и те же записи могут быть членами разных наборов и, следовательно, быть по-разному упорядочены. Средство, избавляющее от полного просмотра всех записей-членов при поиске, называется индексом. Обычно для каждого экземпляра набора создается свой индекс.

Членство записи в наборе может быть:

- 1) автоматическим или ручным,
- 2) обязательным или необязательным.

При автоматическом членстве включение записи в набор требует, чтобы запись вошла во все наборы, в которых она объявлена членом. При обязательном членстве запись нельзя исключить из набора.

Способ выбора экземпляра набора определяется критерием выбора. Критерий выбора задает правила выбора экземпляра набора непосредственно в описании этого набора. Экземпляр набора может задаваться как СИСТЕМНЫЙ, ТЕКУЩИЙ, ПО-КЛЮЧУ, ПО-ФУНКЦИИ-РАССТАНОВКИ. Критерий выбора может состоять в задании непрерывного пути в БД к нужному набору. При описании пути указываются элементы данных, значения которых используются во время поиска.

Для всех наборов СУБД осуществляет поиск только в пределах записей членов текущего набора, выбран-

ного на предыдущем шаге. Текущий набор фиксируется выбором записи-владельца.

В заключительной части раздела приводится программа, печатающая названия КУРСОВ, которые читает ПРОФЕССОР ПЕТРОВ (см. рис. 7). Для этой программы используется подсхема:

ПОДСХЕМА ВУЗ

ОБЛАСТЬ ДИСК

ЗАМОК ДЛЯ ИЗМЕНЕНИЙ ПАРОЛЬ

ЗАПИСЬ ПРОФЕССОР

РЕЖИМ РАЗМЕЩЕНИЯ СИСТЕМА

ОБЛАСТЬ ДИСК

2 ПРОФЕССОР

ТИП СИМВОЛ 40

ЗАПИСЬ КУРС

РЕЖИМ РАЗМЕЩЕНИЯ

ЧЕРЕЗ НАБОР УЧЕБА

ОБЛАСТЬ ДИСК

2 КУРС

ТИП СИМВОЛ 60

НАБОР СОСТАВ

ВЛАДЕЛЕЦ СИСТЕМА

ПОРЯДОК ПОСТОЯННЫЙ

НЕСУЩЕСТВЕННЫЙ

ЧЛЕН НАБОРА ПРОФЕССОР

ВЫБОР НАБОРА ПО ВЛАДЕЛЬЦУ

В СИСТЕМЕ

НАБОР УЧЕБА

ВЛАДЕЛЕЦ ПРОФЕССОР

ПОРЯДОК ПОСТОЯННЫЙ

НЕСУЩЕСТВЕННЫЙ

ЧЛЕН НАБОРА КУРС

ВЫБОР НАБОРА ТЕКУЩИЙ

КОНЕЦ

ПРОГРАММА:

1. НАЧАЛО ПРОГРАММЫ

2. НАЙТИ ЭКЗЕМПЛЯР ЗАПИСИ ПЕТРОВ
С ЗАПИСЬ СТАНОВИТСЯ ТЕКУЩЕЙ
3. ПОЛУЧИТЬ
С ЗАПИСЬ СТАНОВИТСЯ ДОСТУПНОЙ
4. НАПЕЧАТАТЬ ПРОФЕССОР
С ПЕЧАТАЕТСЯ: ПЕТРОВ
5. НАЙТИ ПЕРВЫЙ ЭКЗЕМПЛЯР ЗАПИСИ
КУРС В НАБОРЕ УЧЕБА
6. НАПЕЧАТАТЬ КУРС
С ПЕЧАТАЕТСЯ: АНАЛИЗ
7. НАЙТИ СЛЕДУЮЩИЙ ЭКЗЕМПЛЯР
ЗАПИСИ КУРС
8. ЕСЛИ ЕСТЬ СЛЕДУЮЩИЙ ЭКЗЕМПЛЯР
ТО НАПЕЧАТАТЬ КУРС
И ПЕРЕЙТИ НА СТРОКУ 7
ИНАЧЕ НАПЕЧАТАТЬ КОНЕЦ
РАБОТЫ И ПЕРЕЙТИ НА СТРОКУ 9
9. КОНЕЦ ПРОГРАММЫ

Рассмотренная программа напечатает:

ПЕТРОВ АНАЛИЗ АЛГЕБРА ГЕОМЕТРИЯ

в соответствии с экземпляром набора УЧЕБА-10 на рпс. 7.

Концепция текущего состояния занимает центральное место в сетевой модели. Пользователь при обращении к базе данных выполняет два шага: при первом шаге запись становится текущей записью процесса пользователя, а затем, при втором, — запись становится доступной. Этим двум шагам соответствуют два оператора языка манипулирования данными: НАЙТИ и ПОЛУЧИТЬ, так что любая запись, которая передается пользователю, проходит через состояние текущей записи процесса. Последовательность поиска, в основном, складывается из поиска нужного экземпляра набора и поиска члена внутри этого экземпляра. Если учесть общую упорядоченность записей по ключам базы данных, то можно представить варианты поиска, связанные с последовательным просмотром записей внутри области. При помещении записи в состояние

текущей записи процесса одновременно для всех наборов и для всех областей помещаются текущие записи так же, как для соответствующего типа записи. Подобного рода информация объединяется под общим названием *индикаторов текущего состояния*. Индикаторы текущего состояния содержат:

текущий экземпляр записи процесса,
текущие экземпляры записей всех типов областей,
текущие экземпляры записей всех типов наборов,
текущие экземпляры всех типов записей.

Взаимодействие пользователя с базой данных заключается, в общем, в передаче в рабочую область пользователя индикаторов текущего состояния. Кроме этого имеется ряд специальных регистров базы данных, в которые передается пользователю информация о режиме выполнения его операций. Блок-схема взаимодействия пользователей с базой данных показана на рис. 11.

Программа пользователя связывается через рабочую область с программами СУБД. СУБД, кроме программ выполнения прямых функций поиска, выборки и изменения базы данных, состоит из большой группы программ загрузки, редактирования, выдачи, сбора статистики, реорганизации, сброса и восстановления. С помощью такого рода программ администратор базы данных проектирует, обслуживает, реорганизует и восстанавливает базу данных.

При централизованном хранении важное значение имеет защита информации, которая реализуется программой СУБД:

1) СУБД обеспечивает секретность информации, т. е. защиту от неавторизованного доступа, с обращением только по паролям;

2) СУБД поддерживает целостность базы данных, т. е. защиту от противоречивых и неправильных данных.

Обращение только по паролем выполняется с проверкой соответствия *замков и ключей*. Замки секретности описываются на всех уровнях: схемы, подсхемы, записи, набора, элемента, области. Ключ секретности может быть либо константой, либо аргументом функции, либо значением переменной. Правильные отношения между данными описываются в схеме в виде про-

цедур проверки для указанных объектов. При обращении к данным осуществляется выполнение процедур.

Процесс взаимодействия пользователя, представленный операторами ЯМД, с СУБД сетевой модели состоит из нескольких этапов:

1) пользователь задает значения параметров в рабочей области и затем оператором языка манипулирования данными обращается к СУБД;

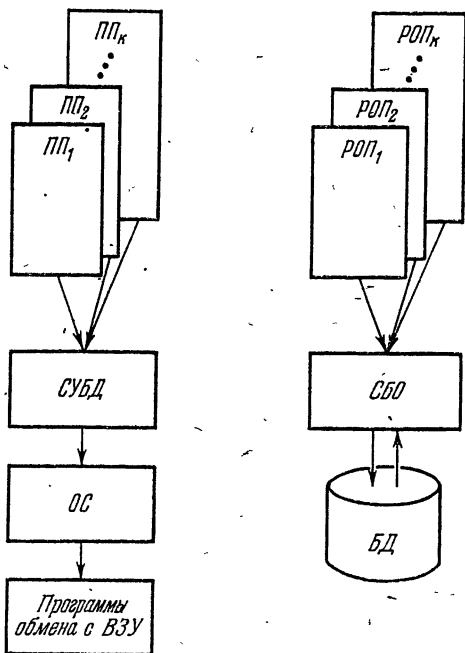


Рис. 11. Блок-схема взаимодействия пользователей с базой данных.

2) СУБД анализирует вызов, значения параметров и индикаторы текущего состояния и обращается к ОС;

3) ОС осуществляет обмены между ВЗУ и системными буферами и возвращает результаты обменов СУБД;

4) СУБД из системных буферов пересылает данные в рабочую область пользователя и формирует индикаторы текущего состояния.

Характер такого взаимодействия предполагает пользователя-программиста, который знает язык манипулирования данными и постоянно работает с базой данных.

Построение модели данных состоит в отображении объектов и отношений между ними в систему хранения на ЭВМ, при этом возникают образы как объектов, так и отношений. В этом смысле сетевая и реляционная модели данных отличаются средствами, с помощью которых пользователь манипулирует с этими образами.

В сетевой модели отношения между объектами представлены в виде наборов, реализуемых указателями. При таком подходе по-разному рассматриваются объекты и отношения между ними. В реляционной модели и объекты, и отношения представлены одним и тем же образом: n -выборками из доменов.

2.4.2. Принципы реляционной модели данных. Требования широкого круга пользователей во многом определяют тенденции развития баз данных. Основой реализации этих тенденций служит развитие реляционной модели данных. Развитие реляционных баз данных сопровождается появлением новых диалоговых систем и распределенных баз данных.

В реляционной модели важную роль играет реализация языка запросов пользователя — ЯЗ.

По сравнению с типичными языками манипулирования данными язык запросов обычно выше уровнем, менее процедурный и ориентирован на непрофессионального пользователя. В то же время уже существуют проекты, представляющие язык манипулирования данными и язык запросов одним подязыком данных.

Увеличению уровня ЯЗ соответствует модель данных более высокого уровня абстракции. Эта модель использует алгебру отношений или реляционное исчисление [84]. Отношение представляется обычно в табличной форме (табл. 1).

Имя таблицы соответствует имени записи, имена столбцов — именам элементов записи, строки таблицы представляют экземпляры записи. Таблица задает *отношение*, имеются операции над таблицами, которые порождают новые таблицы или новые отношения. Совокупность операций образует алгебру отношений.

СТУДЕНТ

НО- МЕР	ФАМИЛИЯ	ГОД- РОЖД	ГОРОД	ПЕРЕЕЗД
1	АНДРЕЕВ	1957	МОСКВА	ТУЛА, КИЕВ
2	ПЕТРОВ	1956	ЛЕНИНГРАД	МОСКВА
3	ИВАНОВ	1956	КИЕВ	МОСКВА
4	СИДОРОВ	1955	МОСКВА	КИЕВ, ОРЕЛ
5	ПЕТРОВ	1957	МОСКВА	ХАРЬКОВ
6	КОТОВ	1954	ОРЕЛ	МОСКВА
7	ИВАНОВ	1958	МОСКВА	ОДЕССА

ТРАНСПОРТ

ФАМИЛИЯ	ГОД-РОЖД	ГОРОД	НОМЕР- БИЛЕТА	ВИД ТРАН- СПОРТА
АНДРЕЕВ	1957	ТУЛА	123576	ПОЕЗД
АНДРЕЕВ	1957	КИЕВ	452399	САМОЛЕТ
ПЕТРОВ	1956	МОСКВА	878992	ПОЕЗД
ИВАНОВ	1956	МОСКВА	137455	САМОЛЕТ
СИДОРОВ	1955	КИЕВ	458854	САМОЛЕТ
СИДОРОВ	1955	ОРЕЛ	919192	ПОЕЗД
ПЕТРОВ	1957	ХАРЬКОВ	534132	ПОЕЗД
КОТОВ	1954	МОСКВА	123456	САМОЛЕТ
ИВАНОВ	1958	ОДЕССА	323435	САМОЛЕТ

В табл. 1 СТУДЕНТ — имя отношения, НОМЕР, ФАМИЛИЯ, ГОД-РОЖД. ГОРОД — имена столбцов, ТРАНСПОРТ — имя другого отношения.

В произвольное отношение R входят множества (столбцы) D_1, D_2, \dots, D_n , не обязательно различные, так, что отношение R задается множеством n выборов, каждая из которых содержит d_1, d_2, \dots, d_n , где $d_i \in D_i$, $i = 1, 2, \dots, n$. Множества D_i называют доменами отношения R , а n -степенью отношения. Столбец D_i или множество столбцов, значения в которых однозначно определяют строки, называется *ключом* отношения или *ключевым доменом*. В каждом отношении должен быть хотя бы один ключ.

Отношение R называется отношением первой нормальной формы, если каждое значение d_i домена D_i неделимо, т. е. не может быть представлено в виде отношения.

Домены отношения, представленного в первой нормальной форме, могут быть связаны функциональными

ми зависимостями, соответствующими их информационному содержанию. Поэтому при выполнении операций по добавлению, вычеркиванию и изменению n -выборок какого-либо отношения программист ограничен не только запрещением дублировать первичный ключ, но и этими функциональными зависимостями.

Домен D_i отношения R функционально зависит от домена D_j , если каждое значение d_i связано только с одним значением d_j , где d_i и d_j принадлежат одной и той же n -выборке.

Отношение R находится в третьей нормальной форме, если все неключевые домены функционально взаимно независимы и функционально зависят только от одного ключевого домена [91]. Если отбросить требование функциональной независимости неключевых доменов, то возникает вторая нормальная форма. При проектировании реляционной базы данных перед администратором стоит задача выбора оптимальной третьей нормальной формы.

Если, например, в табл. 1 в отношение СТУДЕНТ ввести столбец ПЕРЕЕЗД, значениями которого являются названия городов, в которые ездил студент, то студент АНДРЕЕВ ездил в ТУЛУ и КИЕВ, и отношение СТУДЕНТ перестает быть отношением первой нормальной формы.

Для информации о ПЕРЕЕЗДе можно дополнительно к отношению СТУДЕНТ, без столбца ПЕРЕЕЗД, добавить отношение ТРАНСПОРТ, в котором ключом служит сочетание значений столбцов ВИД-ТРАНСПОРТА и НОМЕР-БИЛЕТА. Алгебра отношений содержит ряд операций над отношениями, основными из которых являются: *проекция, объединение, вычеркивание*. Проекция R отношения со столбцами S_1, S_2, \dots, S_n над отношением P со столбцами S_{i_1}, \dots, S_{i_k} состоит в исключении из R всех столбцов с номерами $j \neq i_1, \dots, i_k$ и исключении тех дублирующих строк, которые появляются при исключении столбцов.

Например, проекция СТУДЕНТ над ГОД-РОЖД есть отношение:

ГОД-РОЖД	1957	1956	1955	1954	1958
----------	------	------	------	------	------

Объединение U отношений Q и R над общими значениями d_{i_1}, \dots, d_{i_k} состоит в построении отношения из конкатенации выборок из Q и R , у которых совпадают d_{i_1}, \dots, d_{i_k} , и исключении дублирующих строк:

$$\underbrace{q_1, q_2, \dots, q_{m_1}}_{\text{(только в } Q)} \quad \underbrace{r_1, r_2, \dots, r_{m_2}}_{\text{(только в } R)} \quad \underbrace{d_{i_1}, \dots, d_{i_k}}_{\text{(общие для } Q \text{ и } R)}$$

Например, объединение СТУДЕНТ и ТРАНСПОРТ над

ПЕТРОВ	1956
--------	------

есть

ПЕТРОВ	1956	ЛЕНИН-ГРАД	МОСКВА	11235	ПОЕЗД
--------	------	------------	--------	-------	-------

Объединение в данном случае дает ответ на вопрос: «Куда ездил студент ПЕТРОВ в 1956 г., с каким билетом и каким видом транспорта?» Операция *вычеркивания* исключает указанные в ней строки из таблицы.

Характер взаимодействия пользователя с реляционной моделью отражается в операторах языков запросов. Типичный класс языков запросов образуют *графические языки*, фразы которых пользователь «записывает» на экране терминала (ЭЛТ). Примером графического языка служит язык *Guery by Example* (Запрос на примерах) [87].

В этом языке пользователь записывает *пример ответа*, который он хочет получить по запросу, а система формирует все множество ответов.

Например, пользователь написал на ЭЛТ:

СТУДЕНТ

система ответила:

НОМЕР	ФАМИЛИЯ	ГОД-РОЖД	ГОРОД
-------	---------	----------	-------

Далее пользователь под полученной строкой в соответствующих столбцах записывает *переменные* и *константы*. Переменная, в отличие от константы, подчер-

живается. Пользователя интересуют все возможные значения переменной, одно из которых он подчеркнул. Например, пользователя интересует, какие студенты живут в Москве? Тогда он пишет: ПЕТРОВ, МОСКВА в соответствующих столбцах таблицы:

НОМЕР	ФАМИЛИЯ	ГОД-РОЖД	ГОРОД
	<u>Р. ПЕТРОВ</u>		МОСКВА

где буква Р означает: «выдай на экран (PRINT)».
Система отвечает:

ФАМИЛИЯ	ГОРОД
АНДРЕЕВ	МОСКВА
СИДОРОВ	МОСКВА
ПЕТРОВ	МОСКВА

Ответ на запрос формируется системой с помощью операции проекции отношений.

Другой пример, соответствующий выполнению операции объединения, состоит в запросе: «Где живут студенты, летающие самолетами?»

Пользователь написал на ЭЛТ:

СТУДЕНТ

ответ системы:

НОМЕР	ФАМИЛИЯ	ГОД-РОЖД	ГОРОД
-------	---------	----------	-------

пользователь:

Р. АНДРЕ-
ЕВ

МОСКВА

ТРАНСПОРТ

ответ системы:

ФАМИЛИЯ	ГОД-РОЖД	ГОРОД	НОМ-БИЛ	ВИД-ТРАНС
---------	----------	-------	---------	-----------

пользователь:

АНДРЕЕВ				САМОЛЕТ
---------	--	--	--	---------

ответ системы:

ФАМИЛИЯ	ГОРОД	ВИД-ТРАНС
АНДРЕЕВ	МОСКВА	САМОЛЕТ
ИВАНОВ	КИЕВ	САМОЛЕТ
СПИДОРОВ	МОСКВА	САМОЛЕТ

2.4.3. Процессоры баз данных. Реляционная модель данных ориентирована на поддержание максимальной степени независимости данных. Она предполагает большую степень абстракции, чем сетевая модель, которая во многом учитывает специфику существующих магнитных дисков. Реляционная модель предъявляет дополнительные требования к аппаратуре и тем самым подсказывает и уточняет дальнейшие направления проектирования аппаратуры, тогда как сетевая модель во многом преломляет требования пользователя через призму уже имеющейся аппаратуры. Такая же ситуация аналогична статусу языков фортран и алгол-60. Если фортран непосредственно соответствовал сложившейся практике работы на машине, то алгол-60 представлял обобщенную модель научных расчетов, которая потребовала модификации структуры машины, например, отображения блочной структуры и рекурсивных процедур.

Реляционная модель данных стала основой разработки процессоров баз данных, использующих ассоциативные принципы обработки информации. Примером такого процессора может служить CASSM — Context Addressed Segment Sequential Memory [87], представляющий массив процессоров, каждый из которых имеет доступ к циркулирующей памяти типа дорожки магнитных дисков или регистра на магнитных доменах. Так как все данные представлены циркулирующими потоками, то процессоры выполняют поиск данных, удовлетворяющих заданному условию, параллельно и одновременно. Реализация массива процессоров требовала создания специального языка SLICK. Аналогичную структуру имеет процессор RAP [72], предусматривающий сетку процессоров над циркулирующей памятью.

Если структура процессора баз данных содержит буфер, над которым осуществляются собственно параллельные операции, то узким местом будет оставаться время обмена между буфером и внешним запоминающим устройством; более эффективной является структура, в которой распределенные процессоры непосредственно соединены с устройством хранения. Примером такой системы может служить RARES (Relating Associative Relational Store), которая содержит магнитные диски с головкой на каждую дорожку, с каждой головкой непосредственно соединен микропроцессор. Кроме того, в отличие от RAP и CASSM, где n -выборки располагаются линейно по дорожке, в RARES n выборки размещены поперек, на нескольких дорожках, так что полная выборка считывается за время считывания одного символа.

В общем случае структура процессора баз данных предполагает иерархию буферов, на верхних уровнях которых размещаются каталоги, а на нижних сами данные. Рентабельность процессора баз данных во многом определяется стоимостью устройств массовой памяти.

Процессор баз данных относится к классу ассоциативных процессоров, основными ассоциативными элементами которого служат (рис. 12):

- 1) модульная память данных,
- 2) модульная память структур,
- 3) кластерный буфер.

Память данных имеет наибольшую емкость и эффективно может быть реализована на магнитных дисках, имеющих головку на каждую дорожку. Микропроцессоры, связанные с головкой, выполняют парал-

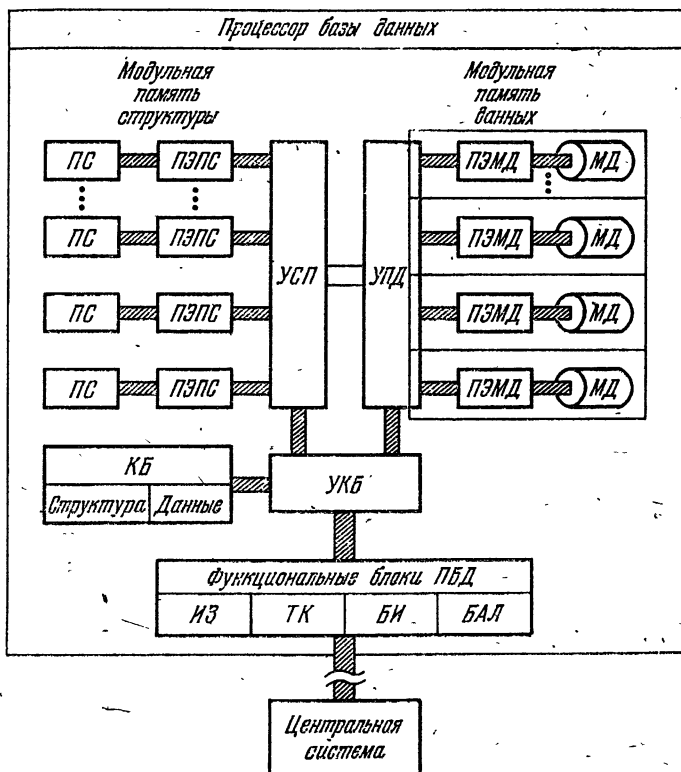


Рис. 12. Общая структура процессора баз данных.

ельные сравнения при последовательном просмотре дорожки.

Память структур также имеет большую емкость, но она должна быть более быстросействующей, чем память данных, поэтому она состоит из модулей электронной памяти. Кластерная память имеет наименьшую емкость, но обладает развитыми средствами поиска

множеств ячеек по их содержимому. Она предназначена для выборочного динамического хранения информации, связанной с одним запросом. С помощью кластерного буфера удастся объединить в одну группу записи, к которым вероятны одновременные обращения.

Разработка процессоров баз данных вызвана резким увеличением сложности задач управления большими объемами данных и недостатками структуры обычных неассоциативных процессоров. Конкретный вид структуры процессора базы данных определяется, с одной стороны, уровнем электронной технологии, а с другой — развитием теории обработки данных сложных структур в направлении максимального упрощения взаимодействия БД с пользователем.

ГЛАВА 3

ОСОБЕННОСТИ СТРУКТУР МОЩНЫХ ЭВМ

3.1. Распределенная обработка

Понятие распределенной обработки актуально для вычислительной техники сегодняшнего дня. Предельным примером распределенной обработки служит объединение территориально разнесенных вычислительных центров (ВЦ) в единую сеть. Распределенная обработка рассматривается как средство решения одной сложной задачи. Действительно, ВЦ объединены в сеть с целью решения одной сложной задачи эффективного обслуживания пользователей, размещенных в разных городах. В аспекте решения одной сложной задачи понятие распределенной обработки связано с декомпозицией задачи на подзадачи, и решение одной сложной задачи как совокупности взаимодействующих подзадач. Декомпозиция задачи, несмотря на внешние отличия конкретных систем, порождает ряд устоявшихся подзадач: ввода/вывода, долговременного хранения больших объемов информации, оперативной обработки, отображения и т. п. Стабильность установившихся подзадач дает возможность погрузить их в аппаратуру соответствующих процессоров. Погружение увеличивает скорость и эффективность обработки, так как одновременная параллельная работа распределенных процессоров сокращает время решения, а специализация аппаратуры делает ее более однородной и тем самым удешевляет процессоры.

В ходе решения сложной задачи подзадачи постепенно разбиваются на все более и более мелкие и организовываются в определенную иерархию. Иерархической

организации подзадач соответствует иерархия процессоров и подпроцессоров. Сложным подзадачам верхних уровней иерархии соответствуют совокупности процессоров и сложные процессоры, например: центральный процессор (ЦП), внешний процессор, архивный процессор и т. д. На самых нижних уровнях, где подзадача сводится к набору элементарных операций, находятся аппаратные блоки процессоров, которые можно рассматривать как подпроцессоры: умножитель, дешифратор, делитель, сумматор и т. п.

Анализ распределенной обработки как параллельной работы процессоров обнаруживает, что проблема сопряжения процессоров занимает центральное место в системе обработки информации. Примером решения этой проблемы может служить построение и стандартизация иерархии протоколов в сети, а для аппаратных блоков — разработка и унификация связей между блоками, что приводит к сокращению количества типов схем.

В иерархии процессоров существуют внутренний и внешний стандарты. Внутренний стандарт возникает как следствие специализации процессора, а внешний необходим для межпроцессорных связей.

В разделах этой главы обсуждаются вопросы распределенной обработки внутри одной ЭВМ, сосредоточенной в узле сети; рассматриваются методы увеличения производительности вычислительной системы, в которой главное место занимает цепочка процессоров, связанных таким образом, что выходная информация предыдущего элемента цепочки служит входной для следующего; анализируются проблемы этого способа объединения. В последних разделах этой главы рассматриваются примеры автоматического преобразования описания программ на традиционных языках высокого уровня с целью выделения и специального оформления структур параллельной обработки, соответствующих структурам мощных ЭВМ.

3.2. Совмещенная обработка команд

Совмещенная обработка команд служит основой увеличения скорости процессора. В этом разделе рассматривается простая совмещенная обработка команд

и вводится понятие *К-структуры* (конвейерной структуры). С целью увеличения производительности процессора выполнение одной операции разбивается на выполнение нескольких микроопераций, каждая из которых реализуется на независимом аппаратном блоке, содержащем один или несколько буферных регистров, и называется *уровнем К-структуры*. На каждом уровне хранится вся информация о частично выполненной операции. Логика взаимодействия уровней обычно описывается в терминах готовности информации для передачи G_i на уровне i и разрешения ее приема P_{i+1} , на уровень $i + 1$, если он свободен. Если уровень $i + 1$ занят G_{i+1} , то передача информации не производится до его освобождения P_{i+1} .

Всего возможно четыре состояния для пары уровней:

$$G_i \cdot P_{i+1}, \quad G_i \cdot G_{i+1}, \quad P_i \cdot P_{i+1}, \quad P_i \cdot G_{i+1},$$

где точка обозначает операцию логического умножения.

В трех случаях:

$$G_i \cdot G_{i+1}, \quad P_i \cdot P_{i+1}, \quad P_i \cdot G_{i+1}$$

смена состояния пары уровней определяется третьим уровнем ($i + 2$), и только в одном случае

$$G_i \cdot P_{i+1}$$

осуществляется передача информации с уровня i на уровень $i + 1$, при этом уровень i освобождается P_i , а уровень $i + 1$ загружается информацией G_{i+1} .

Логика взаимодействия уровней в терминах «готовности и разрешения» аналогична логике обычного старт-стопного взаимодействия.

Совокупность всех уровней обработки образует *К-структуру*. Цель проектирования *К-структуры* состоит в достижении одной и той же пропускной способности для всех последовательно соединенных уровней. На рис. 13 показан процессор с *К-структурой*, которая замыкается оперативной памятью (ОП). Принцип замкнутости *К-структуры* имеет существенное значение для определения производительности процессора. Уровни замкнутой *К-структуры* определяются классом решаемой задачи.

3.2.1. Совмещенное выполнение микроопераций. Типичное выполнение команды процессора состоит в последовательном выполнении следующих микроопераций (как показано на рис. 13):

- 1) выборка команды из ОП в буфер команд;

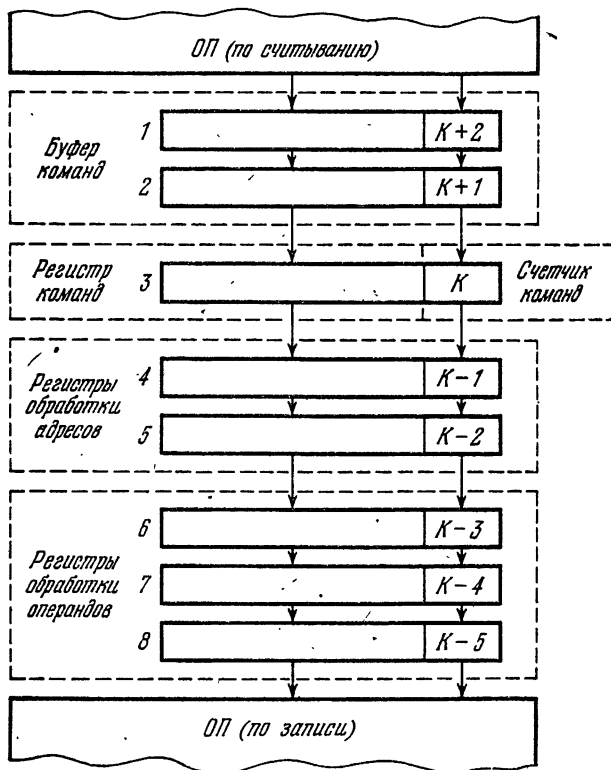


Рис. 13. Графическое изображение К-структуры.

- 2) передача команды между уровнями буфера команд;

- 3) начальная интерпретация команды на уровне регистра команд (РК);

- 4) обработка команды на уровнях регистров адресов операндов;

5) обработка команды на уровнях регистров операндов и результатов арифметического устройства.

В момент интерпретации команды с адресом K , который находится на счетчике команд на уровнях буфера команд, будут обрабатываться команды с адреса-

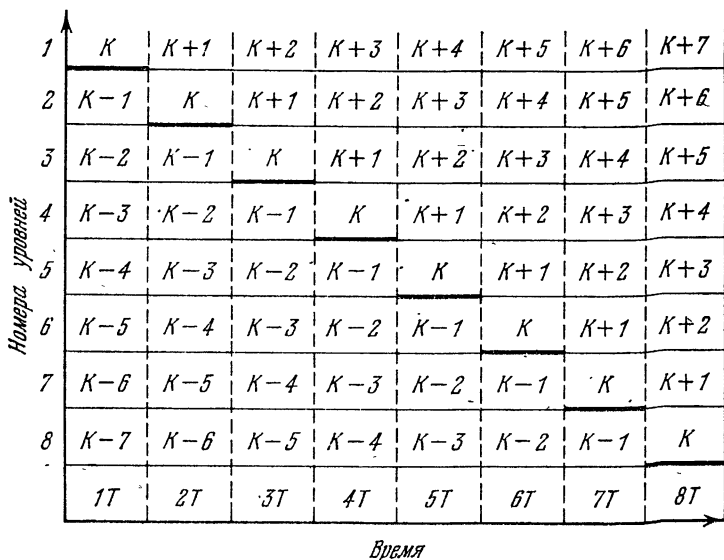


Рис. 14. Временная диаграмма выполнения команд на уровнях К-структуры.

ми $K+1, K+2, \dots$, выбранные «вперед», относительно команды на РК, а на уровнях, находящихся ниже РК, будут находиться команды с адресами $K-1, (K-2), \dots$, которые были выбраны раньше команды K .

На рис. 14 для конкретности принято, что полное выполнение команды состоит в выполнении микроопераций на восьми уровнях. Так как на каждом уровне может быть по одной команде, то производительность в лучшем случае увеличится в 8 раз, и чем больше количество уровней N , тем больше команд может находиться одновременно в обработке. С увеличением количества уровней упрощается микрооперация, выполняемая на каждом уровне. В предельном случае время микрооперации становится равным времени передачи между

уровнями T . В этом случае полное время выполнения команды (T_k) равно

$$T_k \approx 2NT,$$

где N — количество уровней, а T — время передачи, равное времени выполнения одной микрооперации.

Так как одновременно может обрабатываться N команд, то цикл процессора, представляющий эффективное время выполнения одной команды, составит

$$T_c = 2NT/N = 2T.$$

Если команда выполняется на одном уровне, то время T_0 ее выполнения составит

$$T_0 \approx TN.$$

Следовательно, максимальный коэффициент ускорения k равен

$$k = NT/(2T) = N/2.$$

Увеличение количества уровней приводит к значительному увеличению объема аппаратуры, так как на каждом уровне хранится информация о частично выполненной команде.

Чем сложнее и крупнее одна команда, тем больше необходимо хранить информации о ее состоянии на каждом уровне. Таким образом, существует тесная связь между системой команд и затратами оборудования на каждом уровне. Например, если в одноадресной системе команд уровни должны хранить информацию только об одном адресе и об одном операнде, то в двухадресной — о двух и т. д. Одним из способов экономии аппаратуры является передача по уровням не адресов и операндов, а номеров тех регистров, в которых хранятся эти адреса и операнды. Дальнейшая экономия оборудования достигается введением номеров регистров непосредственно в систему команд. Так как внутри уровней существуют свои номера регистров, то номера прямоадресуемых регистров перекодируются в номера тех регистров, которые используются на конкретных уровнях.

Максимальная производительность ЭВМ определяется пропускной способностью самого «узкого» места К-структуры. Для К-структуры, показанной на рис. 15,

таким «узким» местом является ОП. Это объясняется тем, что ОП часто строится с применением ферритовых сердечников, тогда как регистры уровней — на более быстрых полупроводниковых элементах. В примере на

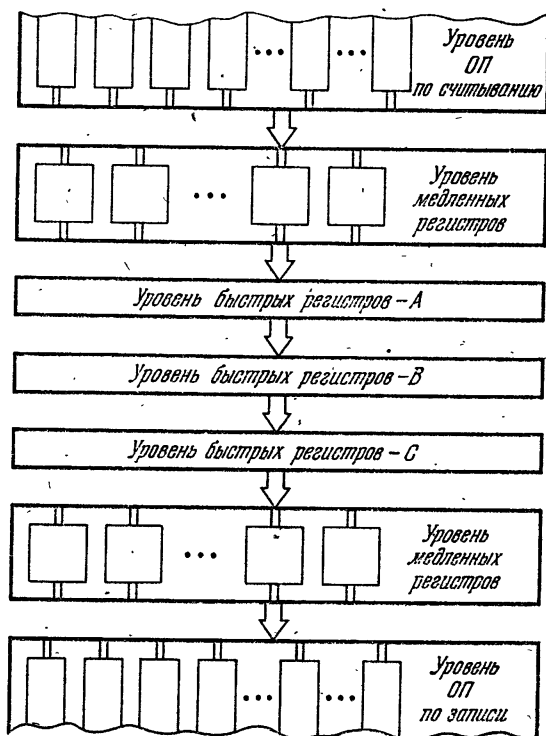


Рис. 15. Б-структура с одинаковой пропускной способностью всех уровней.

рис. 16 пропускная способность К-структуры ограничивается пропускной способностью ОП, и стандартным приемом расширения узкого места служит введение в уровень нескольких независимых параллельно работающих блоков или, в данном случае, независимых модулей ОП.

3.2.2. Заполнение и очистка К-структуры. Оценка общей эффективности К-структуры связана с оценкой

степени ее заполнения. Максимальная эффективность достигается при заполнении всех уровней, т. е. на каждом уровне должна быть команда. Минимальная эффективность соответствует одной команде во всей К-структуре, такая команда *очищает* К-структуру. Очи-

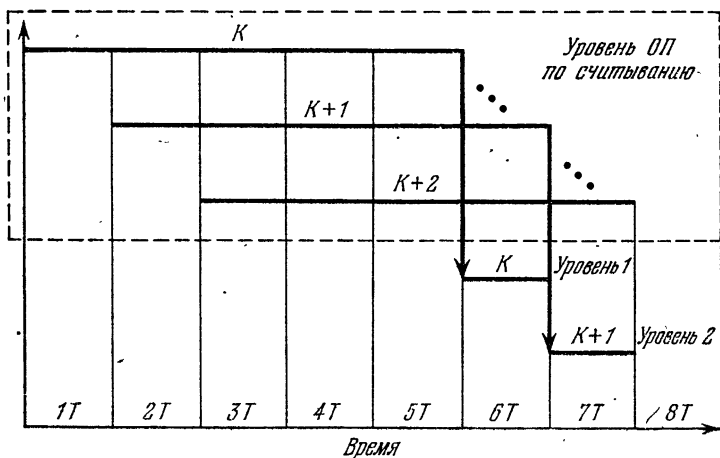


Рис. 16. Временная диаграмма параллельной работы модулей оперативной памяти в К-структуре.

стка вызывается командой условной передачи управления по результату арифметической операции, операцией прерывания, командой вызова подпрограмм и другими командами.

Очистка уровней является логическим следствием того факта, что К-структура эффективна только для обработки последовательно выполняющихся команд, а структура реальной обработки (Р-структура) представляет собой граф с циклами.

Например, если команда условной передачи управления на адрес A по результату арифметической операции имеет адрес K , тогда до тех пор, пока все арифметические операции, загруженные раньше на уровни К-структуры, не выполнятся, неизвестно, какую последовательность команд выполнять: $K + 1$, $K + 2$, ..., или A , $A + 1$, $A + 2$. Поэтому пока очищаются все

уровни ниже РК, уровни выше РК также свободны. Для борьбы с этим явлением уровни выше РК заполняются условно выбранными вперед командами по двум ветвям: $K + 1$, $K + 2$ и $A + 1, \dots$, до тех пор пока не появляется результат арифметической операции. С этого момента команды одной из ветвей уничтожаются, а команды другой ветви продолжают обрабатываться. Степень заполнения уровней К-структуры существенно зависит от класса решаемых задач. Средние оценки длины линейных участков команд для довольно широкого класса логических и вычислительных задач составляют от четырех до семи команд [12].

Аппаратура выборки команд по нескольким ветвям, условного выполнения выбранных вперед команд и отмены ненужных действий может быть упрощена, если использовать ресурсы интегральной технологии, которая обеспечивает проектировщика большим количеством быстрых регистров. Сокращение времени заполнения и очистки уровней К-структуры достигается также введением дополнительных связей уровней с буферной регистровой памятью, в которой запоминается динамическое состояние уровней, а также то, откуда это состояние может быть передано одновременно на все уровни.

3.2.3. Сохранение последовательности команд.

В К-структуре особое место занимает уровень регистра команд (РК), через который последовательно пропускаются все команды программы. Таким образом запоминается требуемый порядок выполнения команд. Этот порядок затем поддерживается при обработке команд на всех уровнях К-структуры.

Учитывая эту функцию уровня РК, его нельзя разделить на несколько параллельных блоков. Предельное ускорение РК можно получить, если оставить на нем только выполнение функции запоминания порядка команд. ЭВМ можно представить как одну К-структуру, в которой группы уровней объединены в процессоры. Процессоры взаимодействуют двумя способами:

- 1) выход одного процессора непосредственно соединен со входом одного или нескольких процессоров;
- 2) выход одного процессора соединен с ОП, с которой соединены входы одного или нескольких процессоров.

Второй способ обладает большой гибкостью и позволяет сгладить колебания мгновенных скоростей процессоров. Минимальный процессор (МП) может состоять из одного уровня РК. На вход такого процессора поступает последовательность команд, а на выходе формируются подпоследовательности, для которых указан порядок их выполнения. Далее процессоры, расположенные после минимального процессора в К-структуре, могут параллельно обрабатывать полученные подпоследовательности.

Чем больше уровней и буферных регистров в процессорах, тем больше [42] могут быть задержки при переходе от одного процессора к другому. Процессоры, образующие непрерывную цепь обработки, не обязательно должны быть все аппаратными, и проблема программно-аппаратной реализации К-структуры состоит в определении того, какие процессоры должны быть программными, а какие аппаратными.

3.3. Независимые функциональные блоки

В реальных К-структурах не удастся добиться одинаковой пропускной способности по всем уровням для всех ситуаций. Обычно на практике допускается для редко встречающихся ситуаций увеличение времени работы отдельных уровней.

Но в тех случаях, когда часто встречающаяся ситуация вызывает увеличение времени работы уровня, приходится вводить в уровень несколько независимых блоков, параллельно выполняющих функцию уровня. Такие блоки называются *функциональными*. Типичными блоками являются модули ОП и блоки арифметического устройства (умножитель, делитель, сумматор и т. п.).

3.3.1. Независимые модули оперативной памяти. Параллельная работа блоков ОП дает возможность сократить время доступа к ОП, которое приблизительно оценивается как время эффективного цикла ОП ($T_{эф}$):

$$T_{эф} = k \cdot T_d / N,$$

где N — количество модулей, k — коэффициент «интерференции» запросов ($k \leq 1$).

При обращении к модулю ОП из адреса формируется номер модуля с помощью функции расстановки [56]. Функция расстановки должна так формировать номера модулей, чтобы для разных последовательно поступающих адресов обращений указывались разные модули. Последовательность поступления адресов заранее неизвестна и определяется динамикой выполнения программы, поэтому выбрать априори удачную функцию расстановки достаточно трудно. На практике реализация функций расстановки опирается на несколько предположений:

1) функция расстановки должна выполняться за минимальное время, так как время выполнения функции добавляется к времени обращения к ОП;

2) функция расстановки должна предпочтительно учитывать большую вероятность появления последовательно возрастающих адресов (линейные участки), например, последовательность адресов команд, последовательность адресов элементов массива данных;

3) функция расстановки должна предполагать, что линейные участки адресов могут начинаться с произвольных начальных адресов.

Первое требование накладывает сильные ограничения на класс возможных функций расстановки, в частности, предпочтительно, чтобы функция расстановки формировалась логическими операциями, например, сложением по mod 2.

3.3.2. Функциональные блоки арифметического устройства. Обычно в научных вычислениях главным ограничивающим фактором является время выполнения операций умножения и деления над числами с плавающей запятой, содержащими 40—60 разрядов.

С целью максимального ускорения арифметических операций каждая операция выполняется на независимом блоке. Типичным примером подобной структуры служит ЦВМ CDC 6600. В CDC 6600 умножение выполняется за 1000 нс, деление за 2900 нс, сложение за 400 нс (над 60-разрядными числами).

Общий выигрыш в производительности зависит от степени загрузки блоков, определяемой характеристиками классов задач. Рассмотрим пример удовлетворительной загрузки функциональных блоков. Пусть

вычисляется выражение

$$(A + B)/C : (A^2 + B^2 + C).$$

В CDC 6600 используется трехадресная регистровая система команд, так что сначала выполняются операции пересылки из ОП в регистры, а затем операции над регистрами. В этом примере считается, что уже все операнды находятся в регистрах:

$$(X1) = A, (X2) = B, (X3) = C.$$

Вычисление значения выражения представляется следующей совокупностью операций:

1. $(X1) + (X2) \rightarrow (X4)$ 400 нс,
2. $(X4)/(X3) \rightarrow (X5)$ 2900 нс,
3. $(X1) * (X1) \rightarrow (X6)$ 1000 нс,
4. $(X2) * (X2) \rightarrow (X7)$ 1000 нс,
5. $(X6) + (X3) \rightarrow (X0)$ 400 нс,
6. $(X0) + (X7) \rightarrow (X7)$ 400 нс,
7. $(X5) * (X7) \rightarrow (X6)$ 1000 нс.

Если бы эти операции выполнялись последовательно, то даже при наличии таких мощных арифметических устройств суммарное время выполнения составило бы 7100 нс. Параллельная работа функциональных блоков дает возможность получить значение выражения за 4300 нс, как показано на рис. 17.

В CDC 6600 время обработки команды на регистре команд составляет 100 нс, так что максимальная скорость равна 10^7 опер/с, но реальная скорость определяется степенью загрузки уровней, и для примера рис. 17 составляет $2 \cdot 10^6$ опер/с. В CDC 6600 регистр команд мультиплексно обслуживает следующие функциональные блоки: два сумматора, один блок деления, два блока сдвига, два блока обработки адресов, один блок логических операций и два вспомогательных блока операций управления. Принцип мультиплексного обслуживания одним быстрым уровнем нескольких медленных блоков другого уровня характерен для К-структур.

В CDC 6600 примером применения этого принципа служит также структура периферийных процессоров,

причем в этом случае интегрально объединены независимые модули памяти, мультиплексено обслуживаемые одним быстрым сумматором. Время цикла сумматора равно 100 нс, а цикл модуля памяти — 1000 нс, таким образом сумматор успевает работать с десятью модулями памяти.

Все 10 модулей подсоединяются к регистрам сумматора с помощью синхронной мультиплексной схемы,

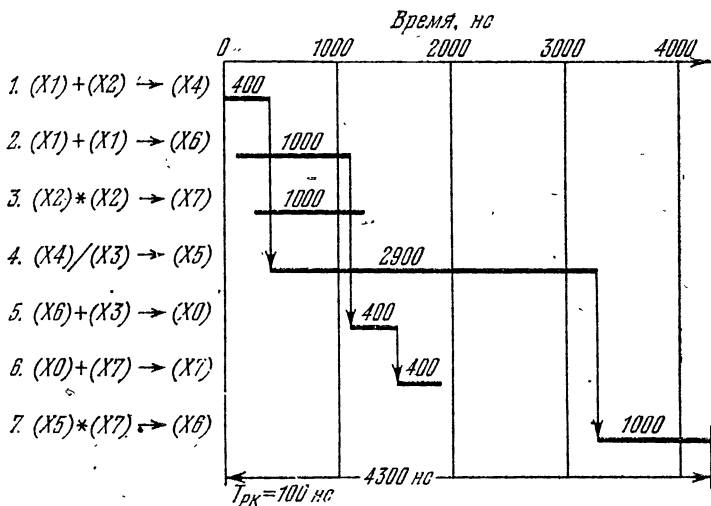


Рис. 17. Временная диаграмма параллельной работы функциональных блоков АУ CDC6600.

пазываемой «стробоскоп». Стробоскоп совершает один оборот за 1000 нс, длительность «окна» стробоскопа составляет 100 нс. В каждые 100 нс через окно к сумматору подключен один модуль памяти. Через окно осуществляется обмен между модулем и сумматором. При выполнении операции считывания из модуля сумматор за 100 нс передает адрес на регистры модуля и через оборот, равный 1000 нс, получает считанное слово.

За общую оценку структуры CDC 6600 с учетом уровней внешней памяти (ВП) можно принять коэффициенты загрузки ЦП при разных объемах ОП. Так, для ОП емкостью $65 \cdot 10^3$ слов степень загрузки процес-

Таблица 2

Параметры ЦВМ Cray1 и Star100

ОП	Cray 1	Star 100
Размер слова (разрядов)	64	64
Емкость (слов)	$256 \cdot 10^3 \cdot 10^6$	$512 \cdot 10^3 \cdot 10^6$
Время цикла (нс)	50	1280
Количество модулей	16	256
Пропускная способность (байт/с)	$640 \cdot 10^6$	$1600 \cdot 10^6$
ЦП		
Время цикла (нс)	12,5	40
Количество подструктур	1	2
Емкость буфера команд (слов)	64	32
АУ		
Максимальная скорость обработки чисел в К-подструктуре (слов/с)		
Сложение	$80 \cdot 10^{16}$	$50 \cdot 10^6$
Умножение	$80 \cdot 10^6$	$25 \cdot 10^6$
Деление	$80 \cdot 10^6$	$12,5 \cdot 10^6$

составляет 35%, а для ОП емкостью $131 \cdot 10^3$ слов степень загрузки — 65%.

С целью увеличения загрузки ЦП в структуру CDC 6600 было введено расширенное запоминающее устройство РОЗУ, с емкостью $2 \cdot 10^6$ слов, с временем обращения к первому слову 3,2 мкс, а затем каждое следующее слово выдается в ОП из РОЗУ через 100 нс. В РОЗУ выделяются дополнительные буферные области (для одного диска — $4 \cdot 10^3$ слов, для одной ленты — 10^3 слов), которые увеличивают пропускную способность внешней памяти и тем самым загрузку ЦП.

Максимальная скорость выполнения операций в CDC 6600 задается сложными операциями: умножением шести разрядов и делением двух разрядов за 1 такт, длительность такта равна 100 нс. Поэтому дальнейший шаг в увеличении производительности прежде всего состоял в ускорении выполнения операций умножения и деления.

Следующей моделью фирмы в направлении использования возможностей К-структуры стала ЦВМ CDC 7600. В CDC 7600 за 1 такт умножается 12 разрядов и делится 3 разряда при длительности такта, равной 27,5 нс. В К-структуру CDC 7600 входит специальная программно-управляемая буферная память, состоящая из 32 модулей, время цикла и емкость каждого модуля соответственно равны 275 нс и 2048 слов. В CDC 7600 имеются команды обмена как между регистрами и буферной памятью, так и между регистрами и оперативной памятью. Емкость ОП CDC 7600 составляет $(512-1024) \cdot 10^3$ слов, и время цикла равно 1760 нс. ОП представлена 16 модулями.

Такт периферийных процессоров CDC 7600 равен 55 нс. Дальнейшее увеличение скорости электронных компонент и сокращение длительности такта уровней дали возможность построить ЦВМ Star 100 и Cray 1, имеющие для всех операций такты выдачи результатов, равные соответственно 20 нс (для одной К-подструктуры) и 12,5 нс (см. табл. 2, а также рис. 18).

Достижение таких скоростей обработки строится на базе:

- 1) предварительного преобразования программы к виду, наиболее удобному для К-структуры, а именно, формирования векторных операций;
- 2) глубокой буферизации медленных ВЗУ, охваченных К-структурой.

ВЗУ, входящие в К-структуру, представляют специальные станции промежуточной обработки, содержащие периферийные процессоры, буферные модули ОП и внешнюю память. Сложные проблемы повышения

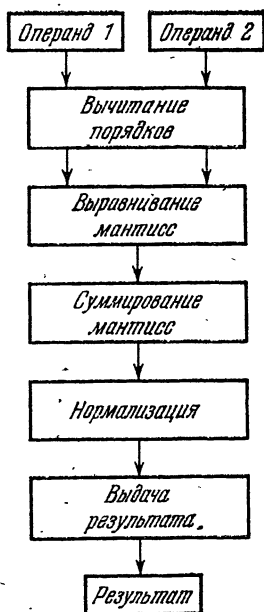


Рис. 18. Группа уровней К-подструктуры арифметического устройства Star 100.

производительности возникли при объединении Star 100 в сеть OSTOPUS. Решением одной из таких проблем явилась разработка единой системы файлов. Единая система файлов охватила все ВЗУ:

— быстрые МД, имеющие головку на каждую дорожку с емкостью $8 \cdot 10^8$ бит и временем обращения 32 миллисекунды,

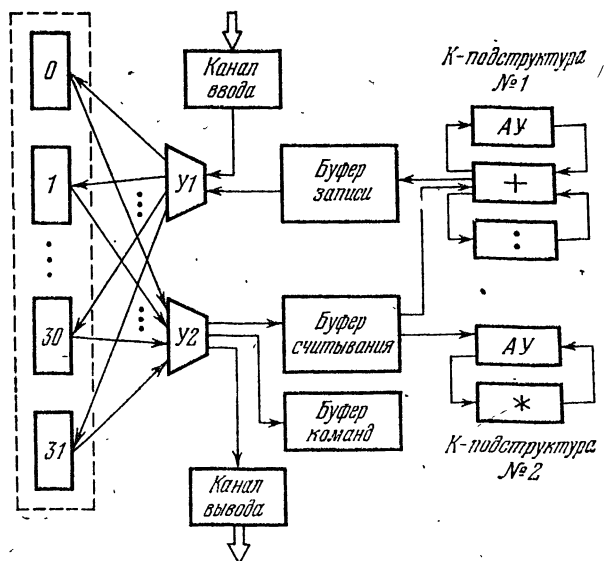


Рис. 19. Блок-схема К-структуры Star 100 с двумя подструктурами № 1 и № 2. У1 — преобразователь 64p в 512p. У2 — преобразователь 512p в 64p. + сумматор для чисел с плавающей запятой, : делитель, * умножитель.

— медленные МД с емкостью $48 \cdot 10^8$ бит и временем обращения 80 миллисекунд.

— ряд специальных устройств, в частности оптическую память IBM photostore с емкостью 10^{12} бит, запись в которую осуществляется лазерным лучом, прожигающим отверстия в полистироловых полосках, нанесенных на поверхность стеклянного диска.

Несмотря на усилия, затраченные на увеличение емкости ВЗУ, мощность обрабатывающих процессоров

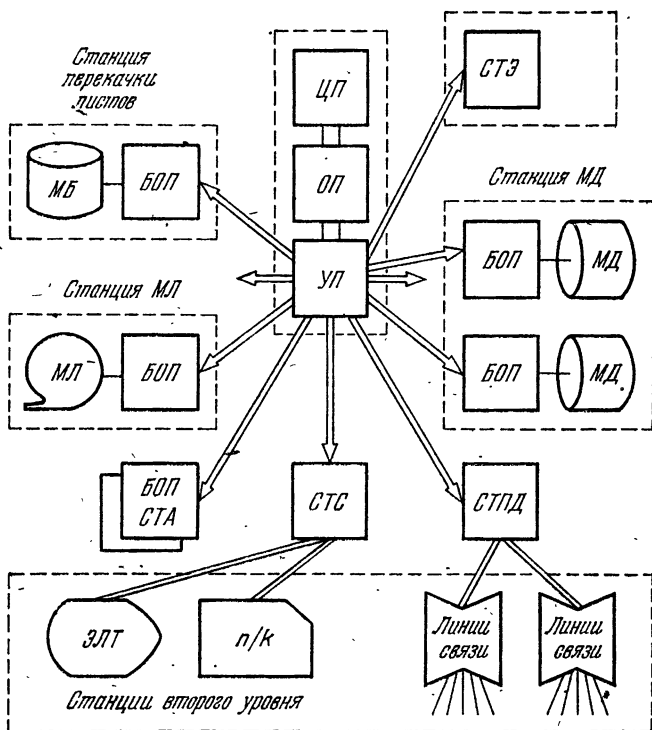


Рис. 20. Блок-схема К-структуры Star 100.

оказалась таковой, что полное заполнение IBM photostore осуществляется за 9—10 месяцев (по данным 1974 г.) (рис. 19, 20).

3.4. Параллельные регистры команд

К-структура, рассматриваемая ранее, предполагала только аппаратное разветвление на параллельные командные потоки. При этом разветвлении аппаратура сохраняет всю информацию, необходимую для дальнейшего слияния потоков. Для упрощения синхронизации групп уровней использовались регистры и области ОП, в которых хранились промежуточные результаты обработки. Аналогичным целям служит программная

подготовка нескольких командных потоков в ОП для параллельной работы подпроцессоров К-структуры, каждый из которых имеет свой регистр команд. Так как информация о разветвлении и слиянии подготавливается заранее программно, то при выполнении используются упрощенные команды синхронизации подпроцессоров. В наиболее простой ситуации одна и та же команда выполняется сразу всеми подпроцессорами. Для такой организации программа предварительно перерабатывается таким образом, что основная масса команд оформляется в виде команд, одинаковых для всех подпроцессоров. Для произвольных наборов команд синхронизация подпроцессоров осуществляется на основе семафоров [15], но эти средства предполагают редкие взаимодействия или *слабую* связь подпроцессоров (ПДП).

Главной характеристикой связи подпроцессоров служит частота синхронизации при обработке общих данных. Чем выше частота, тем *сильнее* связь ПДП. С увеличением связи подпроцессоров должна уменьшаться их внутренняя память. В то же время упрощение подпроцессоров и увеличение их количества сопровождается резким увеличением объема информации о синхронизации. Поэтому необходимо, чтобы предварительно в программе были выделены места, когда одна и та же команда выполняется над несколькими независимыми операциями, и в этих ситуациях синхронизация не нужна. Такие команды называются *векторными*.

Отображение векторных операций в структуре ЭВМ зависит от выбора границы между программой и аппаратурой и может быть выполнено двумя способами. Для первого способа характерно то, что большая часть нагрузки по разделению данных на независимые векторы ложится на программы. Поэтому структура упрощается и состоит из разделенных блоков памяти, каждый из которых связан со своим процессором. Все процессоры объединены уровнем регистра команд. В каждом блоке памяти находятся элементы векторов с одним и тем же номером. Такая структура называется также матричной, и можно говорить, что каждой паре процессор — блок памяти соответствует столбец матрицы.

Для второго способа отображения модули общей памяти связываются через мультиплексные уровни с группой функциональных обрабатывающих модулей. Регистр команд представляется распределенными мультиплексными уровнями управления. Второй способ приводит к более сложной аппаратуре, но за счет этого упрощаются программы подготовки групп векторов и снижаются требования к фиксации длин всех векторов, что приводит к гибким переходам между обычными и векторными операциями.

Программное оформление векторных операций выявило недостатки существующих языков программирования, так, традиционные языки типа алгол-60 и фортран оказались ориентированными на последовательную машину. Более того, пользователю проще рассматривать все свои действия по программе последовательно, так как тем самым сокращается количество анализируемых ситуаций, и удовлетворительным решением проблемы оказалось создание программ, которые из последовательной записи операторов формируют, где возможно, векторные операции.

Для того чтобы упростить алгоритмы этих программ, допускается, что при выполнении векторной операции часть полученных результатов в дальнейшем не используется. Этот подход принят сейчас и в электронике, когда при применении ограниченного числа типов интегральных схем допускается, что часть схемы не будет работать в данном конкретном месте ее применения.

Накопленный опыт использования большого количества параллельных процессоров, управляемых командами, одновременно выполняемыми во всех процессорах, показывает, что во многих случаях удобно ввести фиктивную загрузку процессоров. Таким способом уменьшается частота синхронизации, которая бы потребовалась в случае неоднородной обработки.

Примером такой вычислительной системы, в которой для выполнения одной команды сразу над всеми элементами массива эта команда дублируется в нескольких десятках подпроцессоров, является ЭВМ Иллиак-IV (ILLIAC-IV), спроектированная сотрудниками Иллинойского университета и фирмы Барроус (BURROUGHS).

Иллиак-IV на определенных классах больших задач имеет скорость около $150 \cdot 10^6$ опер/с. Основная мощность машины сосредоточена в шестидесяти четырех процессорных элементах (ПЭ), имеющих собственную память (ПЭП). Каждый из четырех ПЭ соединен с четырьмя соседями. ПЭ, в основном, представляет арифметическое устройство, выполняющее операции над числами с плавающей запятой. Все ПЭ работают под управлением общего устройства управления (УУ), выполняя одновременно одну и ту же команду. Через УУ проходит один поток команд, который размножается сразу для всех ПЭ.

В тех случаях, когда надо выполнить разные действия на ПЭ, применяется команда установки регистра режима конкретного ПЭ. Регистр режима может быть установлен как из УУ, так и из самого ПЭ, в зависимости от результатов вычислений.

ПЭ работает эффективно только с собственной ПЭП, емкость которой равна 2048 64-разрядных слов и время цикла 350 нс [111]. Передача данных от одного ПЭ к другому выполняется через регистры ПЭ командой *маршрутизации*. Эта команда может выполняться одновременно во всех ПЭ, при этом она передает данные на одинаковые расстояния между ПЭ. Единицей расстояния служит переход между соседними ПЭ. Максимальное расстояние для всех ПЭ, рассматриваемых как узлы квадратной решетки, не более 7 (рис. 21).

Каждый ПЭ имеет мощную аппаратуру выполнения операций над числами с плавающей запятой. ПЭ содержит «быстрый» сумматор-умножитель, сдвигатель, логическое устройство, 5 адресуемых регистров для операндов, индексный регистр, сумматор адреса. Аппаратура дает возможность выполнять операции с тактом, равным 62,5 нс (рис. 22, табл. 3). Общая мощность CDC 6600 оценивается в $(2-8) \cdot 10^6$ опер./сек., а мощность одного ПЭ оценивается в $(2-3) \times 10^6$ опер./сек. Такая ВС должна иметь огромную память.

В Иллиаке эта проблема решается применением лазерной памяти с емкостью 10^{12} бит и буферных магнитных дисков (МД), имеющих головку на дорожку при максимальной скорости передачи 10^9 бит/сек. Время одного оборота МД составляет 40 мкс.

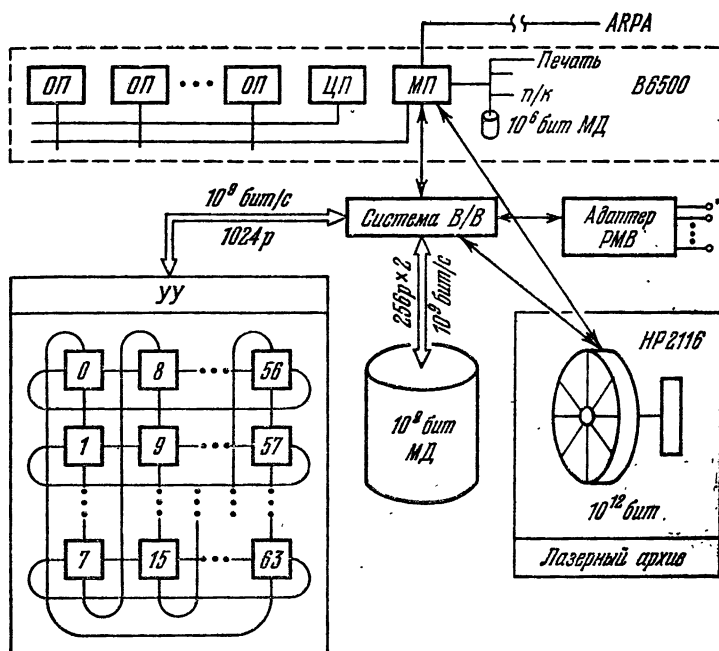


Рис. 21. Общая схема системы Иллиак.

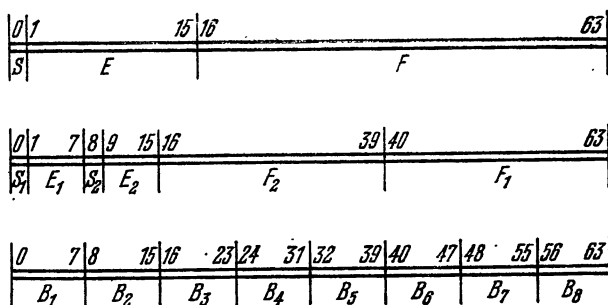


Рис. 22. Основные форматы данных процессорных элементов Иллиака.

Таблица 3

Таблица времен выполнения операций ПЭ Пиллака и CDC 6600

Операция	ПЭ, нсек	6600, нсек	Примечание
СЧ	450	800	считывание
+	450	600	сложение с плавающей запятой
×	680	1000	умножение с плавающей запятой
:	3575	2900	деление с плавающей запятой
РР	65	300	пересылка регистр-регистр
ЗП	30	1000	запись

В Иллиаке команды частично дешифрируются в УУ, а затем передаются во все ПЭ для одновременного выполнения.

ПЭ формирует адрес обращения к ПЭП следующим образом:

$$A = A_k + I_{уу} + I_{пэ_i},$$

где A_k — адрес в команде, $I_{уу}$ — централизованный индексный регистр в УУ, $I_{пэ_i}$ — индексный регистр в ПЭ_i. Полный адрес составляет 18 регистров, где 12-й регистр — адрес внутри ПЭ, а 6-й регистр — номер ПЭ. Команда занимает 32 регистра (рис. 23, условные обозначения см. в Списке сокращений на стр. 149).

Регистры D, A, B, S, R адресуются в командах. Операции ПЭ всегда используют регистры, основные операции следующие: передача регистр — память, арифметические операции (над регистрами A и B, где B — второй операнд), передача регистр-регистра, пересылка между R-регистрами разных ПЭ, загрузка D-регистров.

Идея структуры Иллиака заключается в организации программируемого буфера для большой внешней архивной памяти. С помощью быстрого параллельного канала с большим набором шин осуществляется перекатка информации в 64 разделенных буфера (ПЭ + ПЭП), каждый из которых программируется

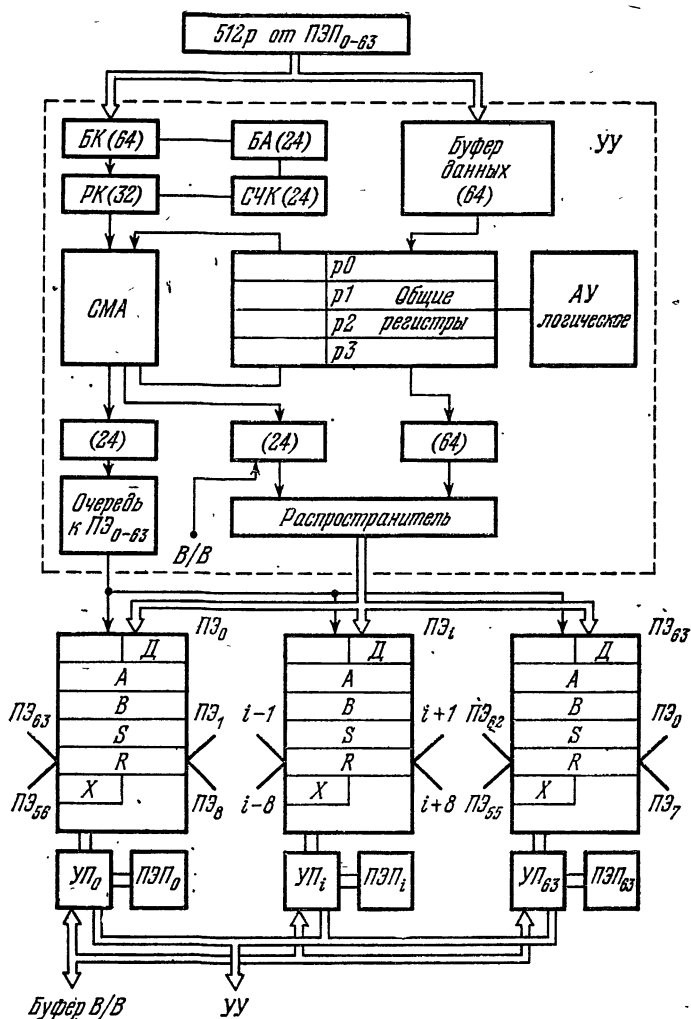


Рис. 23. Устройство управления и матрица процессорных элементов Иллиака.

фактически отдельно на одновременную параллельную обработку (рис. 24).

С целью рационального упрощения основных программ управление разделенными программируемыми буферами организуется как одновременное выполнение одинаковых команд сразу во всех процессорных элементах.

Руководитель проекта Слотник (Slotnick) приводит пример параллельного решения простой задачи распределения температуры на плоской квадратной пластине, если задана температура на границе пластины и установлен равновесный режим. При решении этой задачи сравниваются последовательный и параллельный релаксационный методы для тридцати шести внутренних и двадцати восьми граничных точек.

Релаксации продолжают по точкам до тех пор, пока вычисляемое значение температуры в точках начинает изменяться в пределах заданных ошибок δ , от прохода к проходу. В последовательном режиме, проходя по точкам слева направо, оказывается, что значение у правого конца быстрее приходит к точному режиму, чем у левого. При параллельном алгоритме быстрее к точному решению приходят точки, близкие к границе, и хуже в центре.

Для разностного уравнения

$$u_{ij} = (u_{i-1,j} + u_{i,j+1} + u_{i+1,j} + u_{i,j-1})/4$$

в начальный момент заданы значения u_{ij} по границе, и принимается, что u_{ij} внутри области равны нулю. И параллельный, и последовательный способ требуют одного и того же количества итераций, но параллельный способ в 64 раза быстрее.

Далее на примерах рассматриваются особенности вычислений на Иллиаке. Пусть необходимо сложить покомпонентно два вектора. В программе существенное значение имеет количество координат N , поэтому рассматриваются два случая: $N = 64$ и $64 < N \leq 128$. В первом случае векторы A и B имеют по 64 элемента, расположенные в ячейках α и $\alpha + 2$ всех ПЭ, элементы результирующего вектора размещаются в

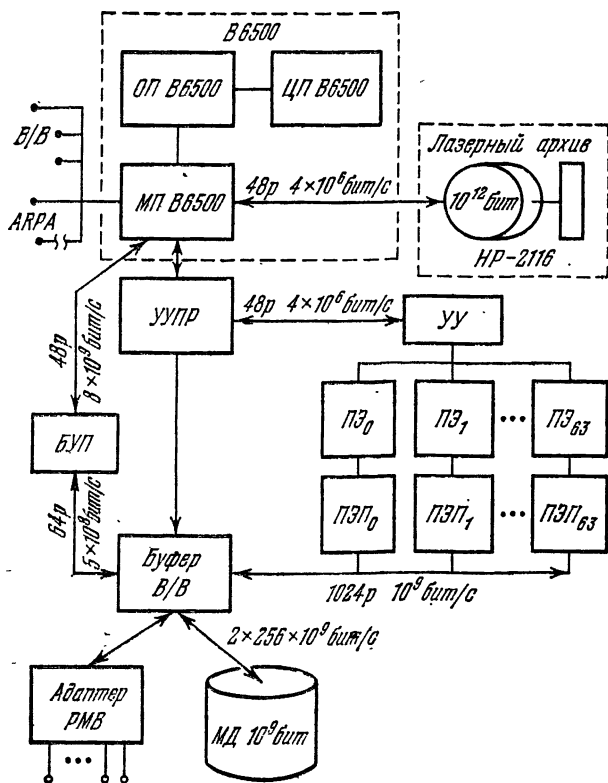


Рис. 24. Связи матрицы ПЭО-63 Иллиака с системой внешней памяти.

ячейках $\alpha + 4$ всех ПЭ. Программа будет состоять из трех команд:

- 1) загрузить А-регистр из ячейки α во всех ПЭ;
- 2) сложить А-регистр с $\alpha + 2$ во всех ПЭ;
- 3) записать А-регистр в $\alpha + 4$ во всех ПЭ.

Во втором случае векторы А и В состоят из N элементов, где $64 < N \leq 128$, элементы А размещаются с 1 по 64-ю ячейку всех ПЭ от ПЭП₀ до ПЭП₆₃, а с 65 и далее — в ячейках $\alpha + 1$ с ПЭП₀ до ПЭП _{$N-64$} , аналогично размещаются элементы вектора В в ячейках $\alpha + 2$, $\alpha + 3$ и, соответственно, для $\alpha + 4$ и $\alpha + 5$. Для этого случая программа будет состоять из шести команд:

- 1) загрузить А-регистр из ячейки α ,
- 2) сложить А-регистр с ячейкой $\alpha + 2$,
- 3) записать А-регистр в ячейку $\alpha + 4$,
- 4) загрузить А-регистр из ячейки $\alpha + 1$,
- 5) сложить А-регистр с ячейкой $\alpha + 3$,
- 6) записать А-регистр в ячейку $\alpha + 5$.

Каждая команда программы непосредственно соответствует оператору автокода Иллиака — ASK.

Так как матрица ПЭП/ПЭ Иллиака представляется как программируемый буфер, то частичное недоиспользование аппаратуры типично для буферов, когда при этом достигается конечная цель — максимальная скорость.

Не во всех задачах параллельное выполнение указывается в явном виде, особенно в тех случаях, когда используются традиционные алгоритмические языки, например, фортран или алгол. Поэтому очень важно применение программы-оптимизатора, автоматически обнаруживающей параллелизм.

Например, простые преобразования заменяют последовательный цикл на параллельный.

Пусть задано

$$DO = 10 \quad I = 2,64$$

$$10 \quad A(I) = B(I) + A(I - 1),$$

цикл разворачивается в следующую линейную последовательность:

$$A(2) = B(2) + A(1)$$

$$A(3) = B(3) + A(2)$$

$$A(4) = B(4) + A(3)$$

$$\overset{\cdot}{A}(\overset{\cdot}{6}\overset{\cdot}{4}) = \overset{\cdot}{B}(\overset{\cdot}{6}\overset{\cdot}{4}) + \overset{\cdot}{A}(\overset{\cdot}{6}\overset{\cdot}{3}),$$

тогда то же самое можно записать иначе:

$$A(2) = B(2) + A(1)$$

$$A(3) = B(3) + B(2) + A(1)$$

$$A(4) = B(4) + B(3) + B(2) + A(1)$$

$$\overset{\cdot}{A}(\overset{\cdot}{6}\overset{\cdot}{4}) = \overset{\cdot}{B}(\overset{\cdot}{6}\overset{\cdot}{4}) + \overset{\cdot}{B}(\overset{\cdot}{6}\overset{\cdot}{3}) + \overset{\cdot}{B}(\overset{\cdot}{6}\overset{\cdot}{2}) + \dots + \overset{\cdot}{B}(\overset{\cdot}{2}) + \overset{\cdot}{A}(\overset{\cdot}{1})$$

и, окончательно,

$$S = A(1)$$

$$DO \quad 10 \quad N = 2,64$$

$$S = S + B(N)$$

$$10 \quad A(N) = S.$$

Этот цикл легко отображается в команды языка Иллпака, например, со следующим распределением памяти:

$A(1)$ в ячейке α ПЭП0,
 $B(2)$ в ячейке α ПЭП1,
 $B(3)$ в ячейке α ПЭП2,
 $B(64)$ в ячейке ПЭП63.

Соответственно используются следующие регистры ПЭ: А — регистр-сумматор, R — регистр маршрутизации, Д — регистр режимов.

Программа для Иллпака может быть представлена в следующем виде:

1. Загрузить единицы в Д, т. е. включить ПЭ.
2. Загрузить А из α во всех ПЭ.
3. $i \leftarrow 0$ (очистка).
4. Загрузить R из А во всех ПЭ (на эту команду не воздействует регистр Д).
5. Маршрутизировать R на 2^i , где переход между соседними ПЭ считается за $i = 1$, и все ПЭ пронумерованы подряд, а ПЭ63 соединен с ПЭ0.
6. $j \leftarrow 2^i - 1$.
7. Установить нули в регистрах Д ПЭ_k, где k изменяется от 0 до j , т. е. выключить ПЭ_k.

8. Суммировать A с R во всех ПЭ (на эту команду воздействует регистр D , поэтому суммирование не произойдет в тех ПЭ, где в D — нули).

9. $i \leftarrow i + 1$.

10. Если $i < 6$, то перейти к п. 4, иначе перейти к п. 11.

11. Установить единицы в D , т. е. включить все ПЭ.

12. Записать A в ячейку $\alpha + 1$.

Следует заметить, что команды передачи управления и модификации индексов выполняются в УУ, тогда как арифметические команды и команды пересылки частично дешифрируются и распространяются во все ПЭ.

Далее выполнение предыдущей программы рассматривается по шагам, скобки $\langle \rangle$ указывают на содержимое регистров, пересылка из A в R обозначается $R = A$, а маршрутизация — $R \Rightarrow$, все операнды, разделенные запятыми, располагаются в *последовательных* ПЭ _{i} , где i изменяется от 0 до 63, например, регистр A во всех ПЭ:

$\langle A \rangle A_1, B_2, B_3, B_4, B_5, \dots$
 $\dots, B_{60}, B_{61}, B_{62}, B_{63}, B_{64}$
 ПЭ₀ ПЭ₁ ПЭ₂ ПЭ₃ ПЭ₄, ...
 $\dots, ПЭ_{59} ПЭ_{60} ПЭ_{61} ПЭ_{62} ПЭ_{63}$

$\langle A \rangle A_1, B_2, B_3, B_4, B_5, \dots$
 $\dots, B_{60}, B_{61}, B_{62}, B_{63}, B_{64}$
 $i = 0$

$\langle R \rangle A_1, B_2, B_3, B_4, B_5, \dots$
 $\dots, B_{60}, B_{61}, B_{62}, B_{63}, B_{64}$

$\langle R \rangle \Rightarrow B_{64}, A_1, B_2, B_3, B_4, \dots$
 2^i
 $\dots, B_{59}, B_{60}, B_{61}, B_{62}, B_{63}$
 $j = 0$
 $D_0 = 0$

$\langle A \rangle A_1, B_2 + A_1, B_3 + B_2, B_3 + B_4, B_5 + B_4,$
 $B_6 + B_5, \dots, B_{61} + B_{60}, B_{62} + B_{61}, B_{63} + B_{62},$
 $B_{64} + B_{63}$
 $i = 1$

$R = A$
 $R \Rightarrow B_{63} + B_{62}, B_{64} + B_{63}, A_1, B_2 + A_1, B_3 + B_2,$
 2^i
 $B_3 + B_4, \dots, B_{57} + B_{58}, B_{59} + B_{58}, B_{60} + B_{59},$
 $B_{61} + B_{60}, B_{62} + B_{61}$

$j = 1$
 $D_0, D_1 = 0$
 (A) $A_1, B_2 + A_1, B_3 + B_2 + A_1, B_4 + B_3 + B_2 + A_1,$
 $B_5 + B_4 + B_3 + B_2, B_6 + B_5 + B_4 + B_3, \dots$
 $\dots, B_{57} + B_{58} + B_{59} + B_{60}, B_{58} + B_{59} + B_{60} +$
 $+ B_{61}, B_{59} + B_{60} + B_{61} + B_{62}, B_{60} + B_{61} + B_{62} +$
 $+ B_{63}, B_{61} + B_{62} + B_{63} + B_{64}$
 $i = 2$
 $R = A$
 $R \Rightarrow B_{61} + B_{60} + B_{59} + B_{58}, B_{62} + B_{61} + B_{60} + B_{59},$
 2^i
 $B_{63} + B_{62} + B_{61} + B_{60}, B_{64} + B_{63} + B_{62} + B_{61},$
 $A_1, B_2 + A_1, \dots, B_{53} + B_{54} + B_{55} + B_{56}, B_{54} +$
 $+ B_{55} + B_{56} + B_{57}, B_{55} + B_{56} + B_{57} + B_{58}, B_{56} +$
 $+ B_{57} + B_{58} + B_{59}, B_{57} + B_{58} + B_{59} + B_{60}$
 (A) $A_1, B_2 + B_1, B_3 + B_2 + A_1, B_4 + B_3 + B_2 + A_1,$
 $B_5 + B_4 + B_3 + B_2 + A_1, \dots, B_{55} + B_{56} + B_{57} +$
 $+ B_{58} + B_{59} + B_{60} + B_{61} + B_{62}, B_{56} + B_{58} +$
 $+ B_{59} + B_{60} + B_{61} + B_{62} + B_{63}, B_{57} + B_{58} +$
 $+ B_{59} + B_{60}.$

Шесть проходов цикла сформируют все 64 суммы. Для того чтобы операции могли сразу выполняться во всех ПЭ, в некоторых случаях операнды необходимо разместить специальным образом в ПЭП.

Пусть, например, необходимо выполнить скалярное умножение двух матриц, тогда приходится размещать операнды так, чтобы получить максимальную скорость в основном цикле.

В примере предполагается, что матрица A размещена в памяти по строкам, а B — по столбцам:

$$A = \begin{vmatrix} a_0 & a_1 & a_2 \\ a_3 & a_4 & a_5 \\ a_6 & a_7 & a_8 \end{vmatrix}, \quad B = \begin{vmatrix} b_0 & b_3 & b_6 \\ b_1 & b_4 & b_7 \\ b_2 & b_5 & b_8 \end{vmatrix}$$

Получение результата состоит из ряда последовательных действий. Предварительно каждая строка матрицы A сдвигается влево циклически, при этом количество сдвигов данной строки равно ее номеру. Каждый столбец матрицы B сдвигается вверх циклически, и количество сдвигов данного столбца равно его номеру. После этого матрицы выглядят следующим

образом:

$$A = \begin{vmatrix} a_0 & a_1 & a_2 \\ a_4 & a_5 & a_3 \\ a_8 & a_6 & a_7 \end{vmatrix}, \quad B = \begin{vmatrix} b_0 & b_4 & b_8 \\ b_1 & b_5 & b_6 \\ b_2 & b_3 & b_7 \end{vmatrix}$$

(следует заметить, что в ПЭ есть 4 общих регистра). Далее производится очистка регистра R_2 , и трехкратно выполняется нижеследующая последовательность команд:

1) перемножить поэлементно A и B и результат оставить на регистре R_1 ,

2) сложить R_1 и R_2 и сумму поместить в регистр R_2 ,

3) все строки матрицы A сдвигаются вправо на единицу,

4) все столбцы матрицы B сдвигаются вниз на единицу.

Подобного вида программы, сводящиеся к независимому накоплению попарных произведений, характерны для вычисления свертки (корреляций); преобразования Фурье и т. п.

Американские фирмы выполнили ряд исследований с целью сравнения возможностей системы Иллиак и требований радиолокационных станций с фазированными решетками [94], и оказалось, что для этих применений достаточно иметь сильно сокращенный, урезанный и, следовательно, удешевленный вариант Иллиака. Проектирование МО Иллиака выполнялось небольшой группой, не более 14 человек, с помощью инструментальной машины. Первый интерпретатор Иллиака был написан еще на B5500 с коэффициентом замедления 10^6 , затем интерпретатор был переведен на B6500 с коэффициентом замедления $2 \cdot 10^5$.

Был разработан транслятор с автокода ASK, и с использованием компилятора компиляторов TWST были реализованы языки: ASK, GLYPNIR, IVTRAN, TRANQUIL (последний в настоящее время не используется). TRANQUIL и GLYPNIR относятся к группе алголоподобных языков, тогда как ASK и IVTRAN — фортран-ориентированные языки [131].

Цель проектирования МО состояла в том, чтобы скрыть специфику конкретной системы команд Иллиака и дать пользователю мощные средства обработки

массивов на уровне алгоритмических языков. Такие языки, как алгол-60 и фортран, имеют ограниченные средства обработки массивов, а именно, выборку по индексу элемента массива, причем в фортране накладываются дополнительные ограничения на размерность и границы массива. Поэтому были сделаны попытки ввести новые средства обработки массивов.

Алголоподобный язык GLYPNIR [121] имеет синтаксические конструкции, ориентированные на структуру Иллиака. За основную единицу данных было выбрано слово. Слова могут объединяться в более сложные конструкции — *C*-слова, представляющие вектор из 64-х слов. Имеются также векторы слов и векторы *C*-слов. Название *C*-слово связано с аббревиатурой CU—Control Unit, т. е. устройство управления. В соответствии с системой команд Иллиак расширено понятие логической переменной введением переменных *some* и *every* для 64 бит.

Some имеет значение «истина», если хотя бы один бит в слове — единица; *every* имеет значение «ложь», если хотя бы один бит — ноль.

Существенно расширены управляющие конструкции, которые могут задать операции сразу для «всех» элементов, например, типа «*forall do*», т. е. «для всех выполнить». В программах, кроме блочной структуры, могут использоваться средства явного управления распределением памяти с помощью указателей. Алгоритмы трансляции оценивают количество пересылок между ПЭ и выбирают вариант распределения памяти с наименьшим количеством пересылок.

Большие усилия при проектировании МО Иллиака были затрачены на сохранение уже имеющейся библиотеки по численным методам, в частности, на перенесение стандартных программ, написанных на фортране, в систему Иллиак. Результатом работ явилось введение программ преобразований входного текста в команды промежуточного языка, который учитывает специфические качества Иллиака (рис. 25).

Система фортран-иллиак-IV (FORTRAN-ILLIAK-IV) содержит стандартные средства: компилятор, редактор связей, который собирает и объединяет независимо скомпилированные модули, загрузчик, библиотеку и подсистему ввода/вывода.

Система фортран-иллиак-IV осуществляет преобразование исходного текста, имеющего последовательный вид, в текст с явно заданным параллелизмом. Использование конструкций параллелизма типа DO FOR ALL приводит к *коллапсу* циклов, т. е. прежде всего исче-

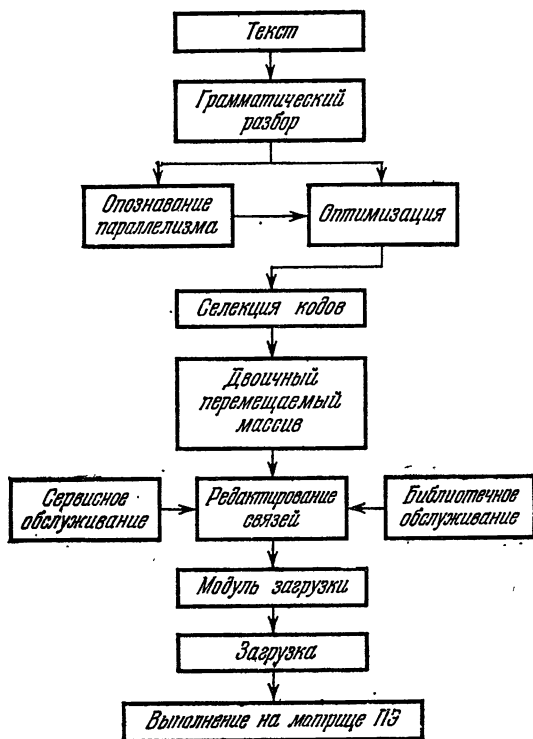


Рис. 25. Блок-схема преобразования входного текста на языке IVTRAN. Регистры: D — режимов, A — сумматора, B — операндов, S — памяти, R — маршрутизации, X — индекса.

зает внутренний цикл, затем цикл, охватывающий внутренний, и т. д. Одновременно необходимо заменить операторы вызова подпрограмм на сами подпрограммы. Таким образом, структура Иллиак требует статического распределения памяти под конкретное число ПЭ.

Так, все массивы, к которым обращаются операторы после DO FOR ALL, должны быть распределены по разным ПЭП, причем DO FOR ALL не может содержать внутри себя другой DO FOR ALL. Оператор DO FOR ALL имеет следующую структуру:

DO K FOR ALL (i_1, i_2, \dots, i_n)/S,

где K — метка последнего оператора в цикле, (i_1, i_2, \dots, i_n) — мультииндекс и S — логический массив (выражение).

Оператор выбирает для обработки те значения, указываемые мультииндексом, для которых значение S — истина.

Ниже рассматривается вычисление \sqrt{a} , где a — элементы массива:

DO 1 FOR ALL(I, J, K)/[1...3]. C.
[1..7].C.[1..10]

IF(A(I, J, K).LT.0.0)A(I, J, K)=-A(I, J, K)
A(I, J, K)=SQRT(A(I, J, K)),

где C. указывает пересечение диапазонов изменения индексов, соответственно, I, J, K; LT — «меньше, чем»; SQRT — квадратный корень.

Использование оператора DO FOR ALL приводит к формированию шестидесяти четырех срезов, в которых нельзя устанавливать эквивалентность массивов разной размерности, например,

A(100) и B(10, 10).

Разработка МО Иллиак на инструментальной машине проводилась с использованием стандартного программного интерфейса, так что по мере разработки можно было постоянно заменять интерпретаторы на работающие модули.

Операционная система Иллиак реализована в виде совокупности параллельных процессов, выполняющихся под управлением МСР В6500. ОС имеет два основных режима: рабочий и тестовый. В тестовом режиме программы диагностики указывают место неисправности с точностью до разъема блока. Это дает возможность заменить неисправный блок на исправный, и ОС сразу же проверит новую аппаратуру. Починка неис-

правного блока проводится на диагностическом процессоре. Связь ОС Иллиака с МСР осуществляется через общие файлы на МД.

Опыт проектирования и эксплуатации Иллиака дает возможность оценить сложность проблемы автоматического распараллеливания программы. Оказалось, что затраты времени на работу программы оптимизации под конкретную структуру Иллиака значительно перекрываются тем выигрышем, который получается за счет сокращения времени счета, по оценке разработчиков.

Предварительная переработка массивов в специальные векторы, удобные для параллельной обработки, может быть реализована не только на разделенной памяти типа Иллиак, но и в общей памяти, где отдельные области общей памяти будут соответствовать разным ПЭП.

Общая пропускная способность общей памяти может быть даже больше пропускной способности одновременно работающих ПЭП, если количество ПЭП меньше количества модулей общей памяти.

Общая память, состоящая из большого модуля, может генерировать каждый следующий операнд через 1 такт общего мультиплексора К-структуры или нескольких К-подструктур.

Единственный способ взаимной синхронизации большого количества процессоров — это организация одной общей К-структуры процессоров, через которую пропускается поток данных. В этом случае обеспечивается одновременный опрос всех признаков синхронизации. Так как данные поступают каждый такт, то скорость обработки их будет равна максимальной скорости К-структуры. При последовательном расположении процессоров в К-структуре задержка при переходе от процессора к процессору равна количеству уровней К-подструктуры. Такая задержка необходима для взаимной синхронизации по общим данным, и если эти данные не зависят друг от друга, как в векторной операции, то группы уровней запускаются вместе в один и тот же момент.

Обработка массивов в общей памяти с использованием векторных операций была реализована в CDC STAR 100. Общий буфер, объединяющий несколь-

ко K-подструктур, для всех векторных массивов и быстрота перехода с векторных операций на скалярные дал возможность получить $50 \cdot 10^6$ операций сложения в секунду, $25 \cdot 10^6$ операций умножения в секунду, $12 \cdot 10^6$ операций деления в секунду при пропускной способности общей памяти $1600 \cdot 10^6$ байт/с.

При этом, если Иллиак-IV очень чувствителен к классу решаемых задач и его производительность колеблется от 2 до 200 миллионов опер/с, то STAR 100 имеет более устойчивую производительность почти во всех областях применения, включая обработку строк.

ЗАКЛЮЧЕНИЕ

Методика проектирования структур основывается на анализе требований системных и прикладных программ и возможностей электронной технологии. Принцип единого интегрального или комплексного проектирования вызывает серьезные проблемы согласования решений разных разработчиков. В процессе разделения труда разработчики объединяются в отдельные специализированные группы. Эти группы решают важные частные проблемы, без чего вообще нельзя приступать к проектированию ЭВМ в целом. Такие группы складываются по конкретной тематике, например: магнитные материалы, печатные платы, алгоритмические языки или формальные грамматики. В перспективе следует ожидать дальнейшего усложнения отдельных проблем и, в связи с этим, возможного расширения групп вплоть до научно-исследовательских институтов, например: институт структур или институт математического обеспечения ЭВМ.

Радикальным подходом к решению проблем интегрального проектирования ЭВМ служит создание комплексной системы автоматизации проектирования (САПР) [45]. Комплексная САПР предполагает развитие структур, как основного фактора, передвигающего границу максимально достижимой производительности, и рост функциональной сложности интегральных схем. Создание таких САПР стало реальным благодаря разработке вычислительных машин высокой производительности.

Важным свойством САПР является то, что в ней могут быть отображены как независимые этапы проектирования, так и связи между ними в форме прог-

рамм и данных, обладающих преимуществами точного машинного описания процесса проектирования, причем отдельные группы разработчиков могут считать, что существуют только специализированные подсистемы программ, удовлетворяющие их интересы, и учитывать остальное окружение как новые входные данные.

Функции программ связи этапов проектирования состоят в том, чтобы поддержать правильное отображение всех изменений, которые возникают на одном этапе проектирования, на все другие этапы.

Процесс проектирования имеет рекурсивный характер, так как возникает необходимость в корректировке начальных этапов по мере решения задач следующих этапов. При этом принципиально важно обеспечить высокую скорость изменения всего проекта при изменении одного из его этапов. Изменения проекта по мере рекурсивных проходов, осуществляемые в САПР, выполняются максимально быстрым способом, так как это реализуется путем обработки данных на ЭВМ.

Существенные выгоды применения САПР достигаются именно на тех этапах проектирования, которые в настоящее время получили точное и недвусмысленное описание. Общая схема этапов проектирования может быть представлена в виде трех иерархических уровней: структурного, логического и технического.

В общем, очевидно, что разработка современных мощных ЭВМ невозможна без автоматизации этапа технического проектирования, в противном случае проект реализуется после окончания срока своего полного морального устаревания.

Для этапа логического проектирования уже сейчас накопился достаточный опыт, из которого ясно, что слабость средств автоматизации логического проектирования приводит к серьезному возрастанию сроков разработки. Это можно объяснить тем, что коррекция решений этапа логического проектирования вызывает лавинные изменения технического проекта. Проблемы структурного проектирования становятся в этом ракурсе еще более острыми, так как незначительные изменения структуры могут вообще зачеркнуть большую часть работы по техническому проектированию. Исходя из этого, целесообразно сосредото-

чить максимум усилий именно на этапе структурного проектирования и его связях с этапом логического проектирования.

Решение задач по разработке САПР во многом зависит от уровня развития математического обеспечения ЭВМ. Реализация новых методов программирования во многом упрощает создание САПР.

Центральной подсистемой САПР служит архив данных, к которому обращаются функциональные программы. Развитие баз данных, встроенных в операционные системы, открывает широкие возможности по использованию общих данных, по сокращению дублирования, по увеличению надежности сохранности данных и т. д., так что база данных может существенно улучшить параметры архива САПР. Конкретная форма базы данных должна отвечать требованиям высокой эффективности. Эта проблема, достаточно сложная, не представляется непреодолимой, и ее решение может основываться на учете специализации пользователей.

В заключение следует отметить, что уже сейчас существует настоятельная потребность в изобретении новых структур, что невозможно без автоматизации структурного проектирования. Эти структуры, вероятно, будут более регулярными, в силу влияния больших интегральных схем, и, конечно, более сложными, как следствие отображения новых методов программирования.

Эту мысль можно подкрепить ссылкой на книгу Г. И. Марчука и В. Е. Котова [37]:

«Прогресс микроминиатюризации и автоматизации проектирования элементов ЭВМ вызывает качественную переоценку их функциональных возможностей, коренным образом меняет методы логического проектирования и их структуру».

СПИСОК СОКРАЩЕНИЙ

АБД	— администратор базы данных
АУ	— арифметическое устройство
БА	— буфер адресов
БАЛ	— блок арифметики и логики
БД	— база данных
БИ	— блок индексов
БК	— буфер команд
БОП	— буферная оперативная память
БП	— блокировка прерываний
БФ	— буферная область
В/В	— ввод/вывод
ВЗУ	— внешнее запоминающее устройство
ВП	— внешняя память
ВС	— вычислительная система
ВТ	— вычислительная техника
ВЦ	— вычислительный центр
ВЯ	— включающий язык
ИЗ	— интерпретатор запросов
ИПП	— информационное поле процесса
КБ	— кластерный буфер
МБ	— магнитные барабаны
МД	— магнитные диски
МЛ	— магнитная лента
МО	— математическое обеспечение
МП	— мультиплексор
ОЗУ	— оперативное запоминающее устройство
ОП	— оперативная память
опер/с	— количество операций в секунду
ОС	— операционная система

ОСРВ — операционная система реального времени
 ПДП — подпроцессоры
 п/к — перфокарты
 п/л — перфоленты
 ПП — прикладные программы
 ПС — модуль памяти структур
 ПЭ — процессорный элемент
 ПЭМД — процессорный элемент модуля данных
 ПЭП — процессорный элемент памяти
 ПЭПС — процессорный элемент памяти структур
 РГБД — рабочая группа по банкам данных
 РК — регистр команд
 РМВ — реальный масштаб времени
 РОЗУ — расширенное оперативное запоминающее устройство
 РОП — рабочая область пользователя
 РП — регистр прерывания
 РРВ — режим разделения времени
 СВО — системные буферные области
 СК — адрес последнего сообщения
 СМА — сумматор адреса
 СН — адрес начального сообщения
 СПФ — справочник файлов
 СС — синхронное считывание
 СТА — станция — архив
 СТПД — станция передачи данных
 СТС — станция средств ввода/вывода
 СТЭ — станция эксплуатации
 СУБД — система управления базой данных
 СЧК — счетчик команд
 ТК — транслятор ключей
 ТЛ — таблица листов
 ТОП — таблица операций
 ТОС — таблица общих сегментов
 ТОЧ — таблица очередей
 ТП — таблица процессов
 ТС — таблица символов
 УВВ — устройство ввода/вывода
 УКБ — управление кластерным буфером
 УМГ — универсальный макрогенератор
 УП — управление памятью
 УПД — управление памятью данных
 УСП — управление структурной памятью
 УУ — устройство управления

УУПР — устройство управления и прерывания
ЦВМ — цифровые вычислительные машины
ЦП — центральный процессор
ЭВМ — электронные вычислительные машины
ЭЛТ — электронно-лучевая трубка
ЯЗ — язык запросов
ЯМД — язык манипулирования данными
ЯОД — язык определения данных

ЛИТЕРАТУРА

1. Автоматизация программирования./Под ред. А. П. Ершова.— Физматгиз, 1961, 368 с.
2. Айлиф Дж. Принципы построения базовой машины.— М.: Мир, 1973, 119 с.
3. Амдаль Дж., Блоу Дж., Брукс Ф. Архитектура системы./Под ред. А. А. Ляпунова и О. Б. Лупанова.— Кибернетический сборник. Вып. 1.— М.: Мир, 1965, с. 101—136.
4. Баррон Д. Рекурсивные методы в программировании.— М.: Мир, 1974, 80 с.
5. Баррон Д. Ассемблеры и загрузчики.— М.: Мир, 1974, 78 с.
6. Бертэн Ж., Риту М., Рутжне Ж. Работа ЭВМ с разделением времени.— М.: Наука, 1970, с. 207.
7. Богданов В. В., Ермаков Е. А., Маклаков А. В. Программирование на языке Алго.— С.: Статистика, 1976, 118 с.
8. Браун П. Обзор макропроцессоров.— М.: Статистика, 1975, 78 с.
9. Браун П. Макропроцессоры и мобильность программного обеспечения.— М.: Мир, 1977, 254 с.
10. Васильев В. А. Язык Алгол-68: Основные понятия./Под ред. С. С. Лаврова.— М.: Наука, 1972, 128 с.
11. Вирт Н. Систематическое программирование: Введение.— М.: Мир, 1977, 184 с.
12. Вулихман В. Е. Статистическое исследование программ машины БЭСМ-6. В кн.: Труды семинара отдела структурных и логических схем, № 9.— М.: ИТМ и ВТ, 1972, с. 33—64.
13. Глушков В. М., Калининченко Л. А., Лазарев В. Г., Сидоров В. И. Сети ЭВМ./Под ред. акад. В. М. Глушкова.— М.: Связь, 1977, 280 с.
14. Дал У., Дейкстра Э., Хоор К. Структурное программирование.— М.: Мир, 1975, 247 с.
15. Дейкстра Э. Взаимодействие последовательных процессов. В кн.: Языки программирования./Под ред. Ф. Женюп.— М.: Мир, 1972, с. 9—84.
16. Деннис Дж. Б., Ван Хорн Э. Семантика программирования для мультипрограммных вычислений. В кн.: Мультипрограммирование и разделение времени.— М.: Мир, 1970, с. 113—143.

17. Джадд Д. Р. Работа с файлами.— М.: Мир, 1975, 145 с.
18. Джейрмейн К. Программирование на IBM/360.— М.: Мир, 1971, 870 с.
19. Донован Дж. Системное программирование.— М.: Мир, 1975, 541 с.
20. Ершов А. П. Метод описания алгоритмических языков, ориентированной на реализацию.— Новосибирск, 1977, 39 с.
21. Задыхайло И. Б., Камынин С. С., Любимский Э. З. Вопросы конструирования вычислительных машин из блоков повышенной квалификации.— Препринт № 68 ИПМ АН СССР.— М., 1971, 13 с.
22. Задыхайло И. Б. и др. Вычислительная система с внутренним языком повышенного уровня.— Препринт № 41, ИПМ АН СССР.— М.: 1975, 42 с.
23. Задыхайло И. Б., Котов Е. И., Красовский А. Г., Мямлин А. Н., Смирнов В. К. О повышении эффективности символьных преобразований.— Препринт № 15 ИПМ АН СССР.— М.: 1974, 33 с.
24. Задыхайло И. Б., Вольдман Г. Некоторые соображения об определении степени непроцедурности языков программирования.— Препринт № 51 ИПМ АН СССР.— М.: 1977.
25. Вычислительная система IBM/360. Пер. под ред. В. С. Штаркмана.— М.: Сов. радио, 1969, 440 с.
26. Функциональная структура IBM/360. Пер. под ред. В. С. Штаркмана.— М.: Сов. радио, 1971, 88 с.
27. Информационные системы общего назначения. Пер. под ред. Е. Л. Ющенко.— М.: Статистика, 1975, 471 с.
28. Каган Б. М., Каневский М. М. Цифровые вычислительные машины и системы.— М.: Энергия, 1974, 680 с.
29. Карпман Л. Я. Об одном способе обработки индексов сложной структуры.— Кибернетика, 1974, № 4, с. 129—134.
30. Карпман Л. Я., Шлеховская И. И., Якуба А. А. и др. Структурная интерпретация языка высокого уровня ALI/360.— Управл. маш. и сист., 1974, вып. XI—XII, № 6, с. 44—52.
31. Катцан Г. Вычислительные машины системы 370.— М.: Мир, 1974, 508 с.
32. Королев Л. Н. Структуры ЭВМ и их математическое обеспечение.— М.: Наука, 1978, 352 с.
33. Лавров С. С., Силагадзе Г. С. Входной язык и интерпретатор системы программирования на базе языка ЛИСП для машины БЭСМ-6.— М.: ИТМ и ВТ, 1969, 116 с.
34. Марков А. А. Теория алгоритмов. В кн.: Труды математического института им. Стеклова, т. 42.— М.: Изд. АН СССР, 1954, с. 375.
35. Мартин Дж. Программирование для вычислительных систем реального времени.— М.: Наука, 1975, 360 с.
36. Маурер У. Введение в программирование на языке ЛИСП.— М.: Мир, 1976, 104 с.
37. Марчук Г. И., Котов В. Е. Модульная асинхронная развиваемая система (концепция), ч. I, II. Препринт 86, 87. Новосибирск, 1978, с. 48, 52.
38. Михайлов А. И., Черный А. А., Гпляревский Р. С. Основы информатики.— М.: Наука, 1968, 756 с.

39. Михелев В. М., Аролович В. С. Предотвращение и диагностика системных тупиков.— М.: ИПМ АН СССР, 1972, 42 с.
40. Многопроцессорные вычислительные системы. (ИПУ АН СССР.)— М.: Наука, 1975, 139 с.
41. Мямлин А. Н., Соснин А. А. Супер-ЭВМ: архитектура и области применения.— М.: ИПМ АН СССР, 1977, 49 с.
42. Проектирование сверхбыстродействующих систем./Под ред. В. Бухгольца.— М.: Мир, 1965, 348 с.
43. Прохоров С. П., Прохорова Т. В. Система БЭСМ апп.— М.: ВЦ АН СССР, 1976.
44. Рендел Б., Рассел Л. Реализация АЛГОЛ-60. М.: Мир, 1967, 476 с.
45. Рябов Г. Г., Лакшин Г. Л. Поэлементное моделирование вычислительных систем.— М.: ИТМ и ВТ, 1978, с. 89.
46. Средства отладки больших систем. Под ред. Р. Растина.— М.: Статистика, 1977, 134 с.
47. Супервизоры и операционные системы./Под ред. Дж. Каштла и П. Робинсона.— М.: Мир, 1972, 157 с.
48. Сэлтон Г. Автоматическая обработка, хранение и поиск информации./Под ред. А. И. Кытова.— М.: Сов. радио, 1973, 560 с.
49. Толчкова А. А. Особенности реализации примитивных функций языка APL 360 в APL-процессоре.— Кибернетика, 1976, № 2, с. 28—33.
50. Уплкс М. Системы с разделением времени.— М.: Мир, 1972, 124 с.
51. Эйсымонт Л. К. О возможности параллельных схем реализации языка для программирования задач переработки текстовой информации.— Упр. сист. и маш., 1977, № 2, с. 56—64.
52. Уолш Д. Руководство по созданию документации для математического обеспечения.— М.: Наука, 1975, 127 с.
53. Фостер Дж. Обработка списков.— М.: Мир, 1974, 72 с.
54. Хенли Дж. Автоматизированная библиотека и информационные системы.— М.: Мир, 1974, 120 с.
55. Хигман Б. Сравнительное изучение языков программирования.— М.: Мир, 1974, 205 с.
56. Хопгуд Ф. Методы компиляции.— М.: Мир, 1972, 160 с.
57. Цикритзис Д., Бернстейн Д. Операционные системы.— М.: Мир, 1977, 336 с.
58. Шура-Бура М. Р. Программирование математических задач для быстродействующих вычислительных машин.— М.: Физматгиз, 1956, 9 с.
59. Система стандартных программ./Под ред. М. Р. Шура-Бура.— М.: Физматгиз, 1958, 231 с.
60. Курс программирования для ГАММА-60./Под ред. М. Р. Шура-Бура.— М.: ИЛ, 1962, 309 с.
61. Шура-Бура М. Р. Вопросы организации математического обеспечения ЭВМ.— М.: ИПМ АН СССР, 1971, 109 с.
62. A bel N. et al. TRANQUIL, a language for an array processing computer.— AFIPS Conference Proceedings, 1969, SICC, v. 34, p. 57—73.

63. Abrams P. S. APL Machine, DPh.— Thesis Stanford Univ., 1970.
64. Ahmad M. Interactive schemas for high speed division.— Computer J., Novem. 1972, v. 15, № 4, p. 333—336.
65. Amdahl delivers its "supercomputer".— Computing, April, 1975, v. 8, p. 3.
66. Architected philosophy implements features directly in hardware.— Computer design, April 1971, v. 10, № 4, p. 26.
67. Atkinson T. D., Raviola G., Garliardi U. O., Schwenk H. S. Modern central processor architecture.— Proceedings of the IEEE, June 1975, v. 63, № 6, p. 863—870.
68. Bachman Ch. Database expert backs codasyl.— Computer Weekly, 8 April 1976, v. 20, № 492, p. 7.
69. Baecker H. D. Garbage collection for virtual memory computer systems.— Comm. ACM, Nov. 1972, v. 15, № 10, p. 981—986.
70. Barsamian H. Firmware sort processor with LSI components.— AFIPS Conference Proceedings, 1970, v. 36, p. 183—190.
71. Baum R. I., Hsiao D. K. Database computers: A step towards data utilities.— IEEE Trans. Comput. 1976, v. 25, № 12, p. 1254—1259.
72. Beaven P. A., Lewin D. W. An associative parallel processing system for non-numerical computation.— Computer J., 1972, v. 15, № 4, p. 343—349.
73. Bennett K. K., Neumann M. D. Extension of existing compilers by the sophisticated use of macros.— Comm. ACM, Septem. 1964, v. 7, № 9, pp. 541, 542.
74. Berkling K. I. A computing machine based on tree structures.— IEEE Trans.— Computer, 1971, v. 20, № 4, p. 404—418.
75. Black C., Wassermann S. 360/195 IBM's most powerful.— Electronic News, 25 August 1969, v. 14, № 724, p. 37.
76. Bobrow D. G. Requirements for advanced programming systems for list processing.— Comm. ACM, July, 1972, v. 15, № 7, p. 618—627.
77. Bouknight W. T., Denenberg S. A. et al. The Illiac IV System.— Proceedings of the IEEE, April 1972, v. 60, № 4, p. 369—388.
78. Boyd H. W. An associative processor architecture for air traffic control.— Lect. Notes Comp. Sci., 1975, v. 24, p. 400—416.
79. Brown D. Control Data 7600 4—8 times faster than 6600.— Electronic News, 9 Decemb., 1968, v. 13, № 685, Sec. 1, p. 32.
80. Brown D. CDC calls its Stak—100 a success.— Electronic News, 16 Novemb., 1970, v. 15, № 791, p. 39, 42.
81. Budnick P., Kuck D. J. The organization and use of parallel memories Illiac IV'.— IEEE Trans. on computer, Decemb. 1971, p. 1566—1573.
82. Burroughs introduces revolutionary fourth generation small scale compiders.— Computer, Nov.—Dec. 1972, v. 5, № 5, p. 59—60.
83. CDC Star 100 Now set for spring.— Electronic News, 18 Sept. 1972, v. 17, № 889, p. 38.

84. Codd E. F. A relational model for large shared data banks.— Comm. ACM, June 1970, v. 13, № 6, p. 377—387.
85. Curtice R. M. Some tools for data base development.— Datamation, 1974, v. 20, № 7, p. 102, 103, 106.
86. Curtice R. M. The outlook for data base management.— Datamation, 1976, v. 22, № 4, p. 46—49.
87. Chamberlin D. D. Relational data base management systems.— Computing surveys, March 1976, v. 8, № 1, p. 43—66.
88. Chevanee R. I. Design of high level oriented processors.— SIGPLAN Notices, v. 12, № 1, Jan. 1977, p. 40—51.
89. Christensen C. On the implementation of Ambit, a language for symbol manipulation.— Comm. ACM, Aug. 1966, v. 9, № 8, p. 570—573.
90. Daley R. C., Dennis J. B. Virtual memory, processes and sharing in MULTICS.— Comm. ACM, May 1968, v. 11, № 5, p. 303—312.
91. Date C. T. An introduction to date base systems.— Reading, MA: Addison — Wesley, 1975, p. 366.
92. Denning P. Fault — tolerant operating systems.— ACM Computing Surveys, Dec. 1976, v. 8, № 4, p. 359—389.
93. Doran R. W. A computer organization with an explicitly treestructured machine language.— The Australian Computer J., Febr. 1972, v. № 1, p. 21—30.
94. Downs H. R. Real-time algorithms and data management on Illiac IV.— IEEE Trans. on Computer, 1973, v. 22, № 8, p. 775—777.
95. Elcod T. H. The CDC 7600 and scope 76.— Datamation, April 1970, v. 15, № 4, p. 80—85.
96. Farber D. J., Griswold R. E., Polonsky I. P. SNOBOL: a string manipulation language.— J. ACM, Jan. 1964, v. 11, № 1, p. 21—30.
97. Fenickel R. R., Yochelson J. C. A LISP garbage-collector for virtual memory computer systems.— Comm. ACM, Novem. 1969, v. 12, № 11, p. 610—612.
98. Freeman D. N. Macrolanguage design for System/360.— IBM Systems J., 1966, v. 5, № 2, p. 63—77.
99. Fujitsu teams with other companies to boost challenge to IBM computers.— Electronics, 26 Decemb. 1974, v. 47, № 26, p. 8E, 10 E.
100. Garliardi D. O. Trends in computing system architecture.— Proceedings of the IEEE, Jan. 1975, v. 63, № 6, p. 858—862.
101. Gliman L., Rose A. J. APL/360 an interactive approach.— N. Y.: J. Wiley & Sons, Inc., 1970, 335 p.
102. Gilman L., Rose A. J. Programming language.— N. Y., 1974, p. 384.
103. Griswold R. E., Poage J. E., Polonsky I. P. The SNOBOL-4 programming language.— N. Y.: Pr. H., 1968, p. 221.
104. Guy G., Steele Jr. Multiprocessing compactifying garbage collection.— Comm. ACM, Sept. 1975, v. 18, № 9, p. 495—508.
105. Habermann A. N. Prevention of system deadlocks.— Comm. ACM, July 1969, v. 12, № 7, p. 373—377, 385.

106. Hansen B. P. Operating system principles.— N. Y., 1973, p. 366.
107. Harvey Ch. Designing and maintaining real time databases.— Computer Weekly, 26 Febr. 1976, v. 20, № 486, p. 10, 11.
108. Hassitt A., Lageschulte J. W., Lyon L. E. Implementation of a high-level language machine.— Comm. ACM, April-1973, v. 16, № 4, p. 199—212.
109. Hauck E. A., Dent B. A. Burroughs B6500/B7500 stack mechanism.— AFIPS Conference Proceedings, 1968, v. 32, p. 245—251.
110. Hohg W. C., Jones P. D. The Control Data Star 100 paging station.— Proceedings, 1973, v. 42, № C. C., p. 421—426.
111. Honeywell series 60 eases update process.— Electronic Design, 24 May 1974, v. 22, № 11, p. 25, 26.
112. The IBM database range VANDL/1, DL/1 Entry, DL/1 and IMS.— Database J., 1975, v. 6, № 10, p. 2—8.
113. Illiac IV concept changing. EDP 1/0 art.— Electronic News, 12 May 1969, v. 14, № 708, sec. 2, p. 40.
114. Iverson K. E. A programming language.— N. Y., London: John Wil. Sons, 1962.
115. Johnstone J. L. RTOS Extending OS/360 for real time spaceflight control.— AFIPS Conference Proceedings, SJCC, 1969, v. 34, p. 15—27.
116. Kuck D. J. Illiac IV software and application programming—IEEE Trans. Comput., Aug. 1968, v. C-17, p. 758—770.
117. Kuck D. J. Supercomputers for ordinary users.— AFIPS Conference Proceedings, Jan. 1972, v. 41, p. 213—220.
118. Large scale computer family.— Data Processing Magazine, Novem. 1970, v. 12, № 11, p. 57, 58.
119. Lamport L. The parallel execution of DO loops.— Comm. ACM, Febr. 1974, v. 17, № 2, p. 83—93.
120. Lamson B. W. A scheduling philosophy for multiprocessing systems.— Comm. ACM, May 1968, v. 11, № 5, p. 347—360.
121. Lawri D. H., Layman T., Baer D., Randell J. M. Glypnier — a programming language for Illiac IV.— Comm. ACM, March 1975, v. 18, № 3, p. 157—164.
122. Lawson H. W. Programming language oriented instruction streams.— IEEE Trans. on Comput., 1968, v. C-17, № 5, 476—485.
123. Lipovski G. I. The architecture of a large associative processor.— AFIPS Conf. Proc., 1970, v. 36, p. 385—396.
124. Lunde Amund. Empirical evaluation of some features of instruction set processor architectures.— Comm. ACM, March 1977, v. 20, № 3, p. 143—182.
125. Lynch W. C. How to stuff an array processor.— Proc. of the Third Texas Conf. on Comput. System, N. Y., 1974, p. 3—3-1, 3—3-2.
126. Mahmoud S. A., Riordon I. S. Software controlled access to distributed data bases.— J. Oper. Res. and Inform. Proc., 1977, v. 15, № 1, p. 22—36.
127. Martin J. Computer data base organization Engl.— Cliffs. N. Y.: Pr. H., 1975, p. 558.

128. McCarthy J. Recursive functions of symbolic expressions and their computation by machine.— Comm. ACM, April 1960, v. 3, № 4.
129. McCrea P. G. An economical high speed display processor.— The Australian Computer J., 1975, v. 7, № 1, p. 3—6.
130. McIlroy M. D. Macro instruction extensions of compiler language.— Comm. ACM, Apr. 1960, v. 3, № 4.
131. Millstein R. E. Control structures in Illiac IV Fortran.— Comm. ACM, Oct. 1973, v. 16, № 10, p. 621—627.
132. Nadler P. L. Analysis of an algorithm for real time garbage collector.— Comm. ACM, Sept. 1976, v. 19, № 9, p. 491.
133. Nessett D. M. The effectiveness of cache memories in a multiprocessor environment.— The Australian Computer J., 1975, v. 7, № 1, p. 33—38.
134. Pearsey T. Distributed computing systems.— The Australian Computer J., 1972, v. 4, № 1, p. 3—11.
135. Poysin L. The CYCLADES network: Present state and development trends.— Proc. Symp. Comput. Networks: Trends and Appl. Gatherburg, Md, N. Y., 1975, p. 8—13.
136. Saltzer J. H., Gintell J. W. An instrumentation of MULTICS.— Comm. ACM, Aug. 1970, v. 13, № 8, p. 495—505.
137. Saltzer J. H. Protection and control of information sharing in MULTICS.— Comm. ACM, July 1974, v. 17, № 7, p. 388—402.
138. Shoeters T. Burroughs — and giant machines.— Computers and Automation. 1968, v. 17, № 5, p. 44, 45.
139. Smith J. L. Data base organization for an array processor.— The Australian Computer J., 1972, v. 4, № 3, p. 98—103.
140. Strachey C. A general purpose macrogenerator.— Computer J., Oct. 1965, v. 8, № 3, p. 225—241.
141. 'Superdisc' can store over six billion bytes.— Electronic Design, 25 Oct. 1973, v. 21, № 22, p. 28.
142. Super-super computer.— Datamation, v. 23, № 3, 1977.
143. Tanenbaum A. S. Structured computer organization.— Engl. Cl., Rd, H., 1976, p. 443.
144. The star shines brighter.— Datamation, 1969, v. 15, № 11, p. 127.
145. Thesis D. T. Vector supercomputers.— Computer, 1975, v. 7, № 4, p. 52—61.
146. T1 will install its 1st 'super' system in '72.— Electronic News, 18 May, 1970, v. 15, № 764, p. 80.
147. Wadler P. C. Analysis of an algorithm for real time garbage collection.— Comm. ACM, Sept. 1976, v. 19, № 9, p. 491—500.
148. Waite W. M. A language independent macro processor.— Comm. ACM, July 1967, v. 10, № 7, p. 433—440.
149. Waite W. M. Building a mobile programming system.— Computer J., Febr., 1970, v. 13, № 1, p. 28—31.
150. Waite W. M. The mobile programming system-STAGE 2.— Comm. ACM, July 1970, v. 13, № 7, p. 415—421.
151. Weiler P. W., Kopp R. S., Dorman R. G., A real time operating system for manned spaceflight.— IEEE Trans. on computer, 1970, v. C—19, № 5, p. 388—398.

- 152 Wilkes M. Time-sharing computer systems.— Ams., 1975, p. 166.
153. Wulfinghoff D. R. Code activitade switching: a solution to multeprocessing problems.— Computer Design, April 1970, v. 10, № 4, p. 67—71.
154. Yachan Chu. Durect — execution computer architecture.— SIGMICRO Newsletter, 1976. v. 7, № 1. p. 49—53.
155. Yasaki E. K. Amdahl's fast 470: two more on way.— Datamation, Oct. 1975, v. 21, № 10, p. 123.
156. Yet another shop drops.— Electronics, 1969, v. 42, № 18, p. 39, 40.
157. Yngve U. H. A programming language for mechanical translation.— Mechanical Translation, 1958, v. 5, № 1, p. 25—41.
158. Yngve U. H. Comit as an IR language.— Comm. ACM, May 1962, v. 5, № 5, p. 19—28.

Вячеслав Федорович Жиров

**МАТЕМАТИЧЕСКОЕ ОБЕСПЕЧЕНИЕ
И ПРОЕКТИРОВАНИЕ СТРУКТУР ЭВМ**

(Серия: «Библиотечка программиста»)

М., 1979 г. 160 стр. с илл.

Редакторы *Н. Н. Васина, М. В. Еремин*

Техн. редактор *Л. В. Лихачева*

Корректор *Л. Н. Боровина*

ИБ № 11257

Сдано в набор 27.12.78. Подписано к печати 17.05.79. Т-08989. Бумага 84×108¹/₃₂, тип. № 3. Обыкновенная гарнитура. Высокая печать. Условн. печ. л. 8,4. Уч.-изд. л. 7,75. Тираж 27 000 экз. Заказ № 383. Цена книги 45 коп.

Издательство «Наука»
Главная редакция физико-математической
литературы
117071, Москва, В-71, Ленинский проспект, 15

4-я типография издательства «Наука».
630077, Новосибирск, 77, Станиславского, 25.

45 к.

5ВГ66
Ж 736

