Н. П. БРУСЕНЦОВ

МИКРОКОМПЬЮТЕРЫ

Н. П. БРУСЕНЦОВ

МИКРОКОМПЬЮТЕРЫ

Допущено Министерством
высшего и среднего специального образования СССР
в качестве учебного пособия
для студентов вузов, обучающихся по специальности
«Прикладная математика»

МОСКВА «НАУКА»
ГЛАВНАЯ РЕДАКЦИЯ
ФИЗИКО-МАТЕМАТИЧЕСКОЙ ЛИТЕРАТУРЫ
1985

ББК 22.18
Б89
УДК 519.6

Брусенцов Н. П. Микрокомпьютеры.— М.: Наука. Гл. ред. физ.-мат. лит., 1985.— 208 с.

Микрокомпьютеры рассматриваются в книге с точки зрения программиста. Разобраны принципы автоматического процессирования, элементы и структуры данных, базовые операции, адресация и управление, способы описания архитектуры компьютера. Детально описаны архитектуры выпускаемых отечественной промышленностью 8- и 16-битных микрокомпьютеров. На примере диалоговой системы структурированного программирования с развиваемым языком РАЯ показана ключевая роль архитектуры в проблемах трудоемкости, надежности и эффективности программного оснащения микрокомпьютеров.

Для студентов вузов по специальности «Прикладная математика».

Рецензенты:
кафедра вычислительных систем и автоматизации научных исследований МФТИ,
доктор технических наук *С. В. Черемных*

Б $\frac{1702070000-135}{053(02)-85}$ 55-85

ОГЛАВЛЕНИЕ

ПРЕДИСЛОВИЕ

Микрокомпьютеры, т. е. компьютеры, выполненные технологическими средствами интегральной микроэлектроники, в огромной степени расширили возможности практического применения цифровой техники, сделали ее ключевым фактором научно-технического прогресса. Решение таких народнохозяйственных задач, как повышение производительности труда и улучшение качества продукции, повышение эффективности научных исследований, укрепление обороноспособности страны, совершенствование системы управления и обслуживания прямо связаны с необходимостью ускоренного внедрения микрокомпьютеров в соответствующие отрасли.

Дело это крайне сложное: недостаточно наладить выпуск (даже обеспеченный надлежащим сервисом) экономически целесообразной, надежной и функционально комплектной микрокомпьютерной аппаратуры. Аппаратура — это всего лишь средства, эффективность которых в решающей степени зависит от того, как их сумеют употребить, а чтобы употребить их с пользой, надо обстоятельно знать как сами средства, так и ту конкретную область, в которой они должны применяться. Короче говоря, специалистам различных областей науки и техники необходимо овладеть применением микрокомпьютеров, подобно тому как они овладевают связанными с их профессиональной деятельностью разделами математики.

Надо признать, что благоприятная возможность для этого пока отсутствует. Успехам технологии, благодаря которым микрокомпьютер стал экономически широкодоступным, не сопутствовали серьезные достижения в том, чтобы обеспечить доступность его в смысле простоты освоения и программирования. Архитектура микрокомпьютеров развивается по пути все большей сложности, в связи с этим трудоемкость разработки и низкая надежность программ остаются острейшими проблемами.

Эта книга может быть полезна разработчикам и пользователям микрокомпьютерных систем в двух отношениях. Во-первых, она содержит общедоступное введение в мир компьютерных понятий, процедур, элементов и структур данных, знакомит со способами их представления и описания (гл. 2 и 3), дает систематические описания двух используемых в нашей стране микрокомпьютерных архитектур: 8- и 16-битной (гл. 4 и 5). Во-вторых, в ней рассмотрен нетрадиционный подход к построению архитектуры микрокомпьютера, основанный на идеях структурированного программирования, диалогового функционирования и развиваемого адаптивного языка РАЯ. Примером реализации данного подхода служит описанная в гл. 6 диалоговая система программирования ДССП, эмулирующая новую архитектуру на имеющихся микрокомпьютерах.

Практика использования этой системы показала, что трудоемкость создания программ уменьшена в ней по сравнению с система-

ми программирования на языке ассемблера в несколько раз, причем имеется реальная возможность проверять правильность, т. е. гарантировать надежность программ. Строго структурированные программы в ДССП значительно понятнее обычных программ, их легче модифицировать, приспосабливать к условиям конкретного применения. Для ДССП характерны простота и единство синтаксиса, легкость освоения и использования системы, элементарность и универсальность базового языка, возможность его наращивания, создания проблемно-ориентированных версий.

Развиваемый подход не лишен, разумеется, недостатков, причем наиболее очевидным и существенным из них является несовместимость структурированного программирования с традиционным «стилем» программирования, основанном на переходах по меткам. Программисту приходится в корне перестраивать привычный способ мышления. Впрочем, новичкам перестраиваться не нужно, они усваивают дисциплину структурирования легко и скоро.

Примером практического применения ДССП служит разработка микрокомпьютерной системы обучения «Наставник», кратко описанная в гл. 7. В противоположность большинству компьютерных систем обучения «Наставник» ориентирован не на обилие технических возможностей, а на достижение максимального дидактического эффекта минимальными средствами. Он основан на принципе «книга плюс компьютер» — работа учащихся с печатным учебным материалом индивидуально направляется, контролируется и оптимизируется в режиме диалога с компьютером. Благодаря компактности кода ДССП и реализуемых в ней программ, удалось обойтись крайне малыми ресурсами: система, обслуживающая 32 терминала учащихся, функционирует на микрокомпьютере как резидентная в главной памяти емкостью менее 48 КВ. «Наставник» особенно целесообразен применительно к новым и быстро развивающимся дисциплинам, обучение которым не может быть осуществлено в широких масштабах с должной оперативностью вследствие нехватки компетентных преподавателей. К таким дисциплинам относится и курс «Микрокомпьютеры».

Автор выражает благодарность декану факультета вычислительной математики и кибернетики МГУ академику А. Н. Тихонову и заведующему кафедрой АСВК члену-корреспонденту АН СССР Л. Н. Королеву за внимание к данной работе и поддержку, а также рецензентам — доктору физико-математических наук И. А. Радкевичу и доктору технических наук С. В. Черемных, высказавшим ряд существенных замечаний и рекомендаций. Особая благодарность сотрудникам и аспирантам лаборатории ЭВМ МГУ, трудом которых созданы ДССП, «Наставник» и все, что им предшествовало, начиная с машины «Сетунь», 25 лет назад. Весь этот путь вместе с автором прошли С. П. Маслов, Х. Рамиль Альварес, В. П. Розин и А. М. Тишулина.

ГЛАВА 1

ОБЩЕЕ ПОНЯТИЕ О МИКРОКОМПЬЮТЕРАХ

Микрокомпьютеры, микрокомпьютерные системы — это самая молодая, быстро развивающаяся и наиболее перспективная отрасль компьютерной техники. Уже сегодня практически не осталось областей человеческой деятельности, в которые не проникли или не проникают микрокомпьютеры. Объем производства их исчисляется десятками миллионов экземпляров в год и неуклонно растет. В любом деле надлежащее использование микрокомпьютеров позволяет значительно повысить эффективность и качество труда, улучшить параметры машин и процессов, оптимизировать организацию и управление в системах. Микрокомпьютер является самым гибким и универсальным средством автоматизации как производства, так и сферы обслуживания, управления, научных исследований, выступает как наиболее мощное и действенное орудие научно-технического прогресса.

Умение рационально использовать микрокомпьютер в своей профессиональной деятельности становится важным показателем полноценности современного специалиста, незнание микрокомпьютерной техники расценивается как отсталость. С широким распространением микрокомпьютеров, в частности, особенно популярной их разновидности — персональных компьютеров возникло понятие «компьютерная грамотность», недвусмысленно намекающее на то, что способность пользоваться компьютером ставится в один ряд с умением читать и писать. Такое может показаться совершенно немыслимым, если иметь в виду таинственные и недоступные для большинства людей электронные вычислительные машины, у которых колдуют электроники в белых халатах. Но это вполне реально и естественно по отношению к микрокомпьютерам, встраиваемым в бытовые приборы и в игрушки, получающим все большее распространение в виде программируемых калькуляторов, кассовых автоматов, прогрессивной оргтехники, персональных компьютеров.

В микрокомпьютерном воплощении компьютерная техника становится таким же ключевым элементом нашей культуры, каким в свое время явилось возникновение языка, письменности, математики. Поэтому понимание принципов этой техники, умение ос оз-

нанно использовать ее возможности применительно к тем или иным приложениям (а именно в таком смысле мы употребляем термин «компьютерная грамотность») следует рассматривать сегодня как неотъемлемую принадлежность образованного человека. Не требуется, чтобы все знали подробности инженерной реализации компьютеров или премудрости системного программирования, но овладение основными понятиями компьютерной техники, хотя бы общее представление о том, что и как может делать компьютер и чего не может, обстоятельное знание возможностей и методов применения компьютеров в своей профессиональной области и, наконец, умение воспользоваться этими возможностями и методами при решении конкретных практических задач должны быть обязательными для каждого специалиста.

Совершенно очевидно, что удовлетворение данного требования в масштабе народного хозяйства является исключительно трудной задачей. Микрокомпьютеры — это не только сложный и необычный, но вместе с тем, молодой и бурно развивающийся вид техники. Традиционная система образования и переподготовки кадров в этих условиях оказывается недостаточно эффективной: вследствие новизны и сложности микрокомпьютерной техники имеет место острая нехватка компетентных преподавателей, слабо разработана методика обучения, мало удовлетворительных учебников, причем и то, что есть, быстро устаревает и отстает — преподавателей надо переподготавливать, методики совершенствовать, учебники создавать заново. Все это требует не только значительных средств и усилий, но также времени — инерционность системы препятствует быстрому развитию, сковывает прогресс.

Решение проблемы находится в области самой микрокомпьютерной техники и заключается в том, что, во-первых, эту технику необходимо сделать как можно более доступной для освоения и использования людьми, а во-вторых, ее надо применить в качестве средства обучения, позволяющего осуществить в самых широких масштабах в сжатые сроки и при минимальных трудозатратах овладение как микрокомпьютерной, так и любой другой техникой.

Настоящая книга посвящена рассмотрению микрокомпьютеров главным образом с точки зрения практической реализации этих двух важных возможностей. Она содержит общую характеристику микрокомпьютеров, описание наиболее популярных в нашей стране микрокомпьютерных архитектур, рассмотрение структурированного программирования и диалога как основных средств рационализации и интенсификации разработок микрокомпьютерных систем, пример простой и эффективной системы обучения, реализованной на базе микрокомпьютера.

§ 1. Что такое микрокомпьютер

Термин «микрокомпьютер» образован из двух слов — «компьютер» и «микро».

Слово *компьютер* происходит от латинского глагола computo — считать, вычислять. В английском языке, из которого заимствовано это слово, computer буквально значит «вычислитель», т. е. выполняющий вычисления. Но как технический термин слово компьютер с развитием цифровых машин приобрело иной смысл: согласно толковому словарю английского языка computer — это электронное устройство для программируемой обработки данных. Под данными понимаются не только и не столько числа, а всякая информация в дискретном (цифровом) представлении — всевозможные тексты, изображения, сигналы и т. п.

Существенно, что компьютер выполняет обработку данных автоматически, реализуя загруженную в его память программу, которая, так же как и данные, представлена цифровым кодом. Таким образом, компьютером называют автоматическую цифровую машину, управляемую хранимой программой, причем, как правило, имеется в виду электронный, т. е. реализованный средствами электроники компьютер. В случаях, когда какой-либо из перечисленных атрибутов не имеет места, к слову компьютер добавляют определяющее слово или слова, благодаря чему стандартный смысл термина изменяется в том или ином отношении. Так, если компьютер не электронный, то указывают, какой он, например: «магнитный компьютер», «гидравлический компьютер». Если он не цифровой, то говорят соответственно «аналоговый компьютер» или «компьютер непрерывного действия», «гибридный компьютер» или «аналого-цифровой компьютер». Если же имеется в виду цифровая электронная машина или устройство, в которых не используется или используется в незначительной степени программное управление, то говорят не «компьютер», а «калькулятор». Например: «карманный калькулятор», «настольный калькулятор». Слово «калькулятор» также значит «вычислитель», причем в противоположность слову «компьютер» оно употребляется именно в этом значении, т. е. означает устройство для манипулирования числами, для вычислений. Впрочем, существуют программируемые калькуляторы и наиболее развитые версии их практически не отличаются от компьютеров.

Понятно, что и язык, и компьютерная техника развиваются, и поэтому нельзя раз и навсегда зафиксировать значения терминов. Однако охарактеризованное выше значение термина «компьютер» (программируемый цифровой обработчик всевозможных данных, реализованный на базе электроники) можно считать устоявшимся: программируемость, дискретность представления и всеобщность

обрабатываемых данных составляют сущность компьютера, а электронное воплощение этой сущности подразумевается потому, что практически все современные компьютеры являются электронными. В прошлом говорили «электронный компьютер», «транзисторный компьютер», «автоматический электронный компьютер», «быстродействующий цифровой электронный компьютер» и т. п. Сегодня нет необходимости всякий раз явно перечислять эти качества, поскольку все они подразумеваются в содержании термина «компьютер», в случае же отступления от общепринятого смысла употребляются дополнительные слова, как указано выше. Дополнительные определяющие слова используются, естественно, и при необходимости обозначить какой-нибудь отдельный вид компьютера, например: «настольный компьютер», «портативный компьютер», «персональный компьютер».

Мы уделяем так много внимания и места обсуждению терминологии, потому что рациональность терминологии составляет важное условие успеха в таком сложном деле, каким являются компьютеры, а стихийно сложившаяся и настойчиво закрепляемая стандартами и терминологическими справочниками русская компьютерная терминология едва ли может быть признана рациональной. Рекомендуется не допускать в научно-техническую литературу слово «компьютер», несмотря на то, что оно в русском языке уже имеется и широко используется в обиходе, научно-популярных книгах и в массовой печати, а употреблять в качестве узаконенного русского термина словосочетание «электронная вычислительная машина» или аббревиатуру ЭВМ. Но термин «электронная вычислительная машина» не является синонимом слова «компьютер» в его современном значении, потому что он явно обозначает именно вычислительную машину, крайне суживая представление о компьютере как об универсальном инструменте управления, автоматизации, манипулирования любыми данными.

Можно возразить, что использование термина «компьютер» вместо «ЭВМ» по существу ничего не меняет, поскольку «компьютер» в буквальном переводе с английского значит «вычислитель», и, следовательно, опять имеем ту же «вычислительную машину». Но на самом деле это не так. В современном английском языке слово «компьютер» приобрело в качестве основного новое, указанное выше значение, а в прежнем значении используется преимущественно его бывший синоним —«калькулятор». В русском языке термин «компьютер» употребляется в его новом значении и с вычислениями в узком смысле может вообще не связываться.

Мы будем употреблять слово «компьютер» без каких-либо ограничений, причем в том широком смысле, который был здесь охарактеризован. Аббревиатура «ЭВМ» будет использоваться как

название машин преимущественно вычислительного назначения, а также в составе собственных имен, например: ЕС ЭВМ, СМ ЭВМ.

Теперь вернемся к термину «микрокомпьютер». Он обозначает разновидность компьютера, отличительная особенность которой заключается в том, что по меньшей мере устройство преобразования данных и управления работой компьютера (процессор) осуществлено средствами микроэлектроники, как правило, на одном кристалле кремния, в виде одной микросхемы (большой интегральной схемы — БИС) и называется *микропроцессором*. Короче, *микрокомпьютер* — это компьютер, выполненный на базе микропроцессора.

Микропроцессор реализует такие функции, как выборку в предписанной программой последовательности, декодирование и управление выполнением команд, а также выполнение операций тестирования и преобразования данных. Таким образом он организует и отчасти осуществляет заданную в виде программы последовательность действий — процесс, откуда и название — процессор, микропроцессор.

Помимо микропроцессора, в состав микрокомпьютера входит запоминающее устройство для хранения программы и данных (главная память) и органы управления периферийным оборудованием — контроллеры внешних устройств. Все это может быть размещено и обычно размещается на одной плате с печатными электрическими соединениями — *одноплатный микрокомпьютер*. Однако наряду с одноплатными, существуют микрокомпьютеры, выполненные на нескольких печатных платах, а также *однокристальные микрокомпьютеры* в виде большой интегральной схемы, на которой размещены процессор, память и некоторые средства управления периферией.

Многоплатная реализация характеризуется наибольшей гибкостью: можно варьировать и расширять набор контроллеров, наращивать память, подключать вспомогательные процессоры. Однокристальный микрокомпьютер наименее гибок, но обладает такими преимуществами как компактность, надежность, низкая стоимость.

Микрокомпьютер в комплексе с периферийным оборудованием и программным оснащением образует *микрокомпьютерную систему*. Возможно большое многообразие микрокомпьютерных систем, различающихся по назначению, составу, конструктивному оформлению и другим атрибутам. В частности, микрокомпьютерные системы могут быть многопроцессорными и многокомпьютерными (мультикомпьютерными). Такие системы называют обычно мультимикропроцессорными или мультимикрокомпьютерными комплексами. Для комплекса характерно тесное взаимодействие входящих в него процессоров или компьютеров, функционирование системы

как единого целого. Другой формой объединения компьютеров является сеть. В сети связь компьютеров друг с другом сравнительно слаба — они могут обмениваться сообщениями, но в остальном функционируют независимо.

Микрокомпьютеры широко применяются также вне микрокомпьютерных систем в виде однокристальных или одноплатных устройств, встраиваемых в качестве контроллеров во всевозможные приборы и машины. Например, такие микрокомпьютеры используются для управления металлообрабатывающими станками, в радиоизмерительных приборах, в дисплейных терминалах и другом периферийном оборудовании, в бытовых приборах и т. п.

§ 2. Как он действует

В первом приближении действие микрокомпьютера, как и компьютера вообще, можно охарактеризовать следующим образом. Имеется запоминающее устройство, называемое также памятью компьютера, в которое посредством устройств ввода вводят (загружают, записывают) программу, которую компьютер должен выполнять, и данные, т. е. то, над чем (или с чем) должны производиться предписанные программой процедуры. Выполнение программы осуществляет *процессор* — устройство, которое считывает из памяти элементы программы (команды) в том порядке, в котором их надлежит выполнять, считывает указанные в командах элементы данных, производит над ними заданные действия и записывает в память полученные результаты. Процессор может также считывать элементы программ и данных с устройств ввода (например, с клавиатуры, с перфоленты, с магнитных носителей) и выдавать их на устройства вывода — на дисплей, на принтер, на перфоратор ленты или на магнитный диск. Кроме того, процессору свойственно воспринимать поступающие извне запросы и реагировать на них предусмотренным в программе образом.

Принципиальными представляются прежде всего два момента:

1) символическая запись программы и данных,

2) автоматическое процессирование — выполнение предписанных программой действий в последовательности, формируемой с учетом складывающихся в ходе выполнения программы условий и внешних запросов.

В отношении символического представления (кодирования) программ и данных имеется много общего между компьютерами и письменностью. Более того, можно сказать, что в компьютерах воссоздана система алфавитного (неиероглифического) письма. Компьютер работает не с вещественными предметами и явлениями того или иного рода, а с символами предметов и явлений, представленными так называемым двоичным цифровым кодом — словами, в которых используются только две литеры: 0 и 1. Цифровым код назван потому, что литерами в нем являются цифры и слова обычно истолковываются как двоичные числа. Это удобно, поскольку значения слов оказываются естественно упорядоченными и над ними определены арифметические операции. Однако с более общей точки зрения компьютерное слово вполне аналогично словам человеческого языка и отличается от них только тем, что обходится минимальным алфавитом — две литеры вместо 33 букв в русском или 26 в латинском алфавитах.

Алфавит, включающий только две литеры, избран в компьютерах по техническим причинам: различать и опознавать одно из двух проще, чем из большего числа возможностей, а кроме того, двузначная логика является самой элементарной и наиболее изученной. Литеры 0 и 1 представляются в памяти компьютера как наличие и отсутствие заряда, тока, намагниченности, а при считывании и передаче от устройства к устройству — в виде импульса или отсутствия импульса тока (напряжения). Существенно, чтобы состояния и сигналы, соответствующие литерам, были *дискретными*, надежно различимыми. На практике это достигается следующим путем.

Пусть физический носитель сигнала (например, электрический потенциал) характеризуется величиной, которая представляет собой непрерывную функцию времени $u(t)$. Условимся, что литере 1 сопоставлено некоторое значение этой величины U_1 (для определенности примем $U_1 > 0$), а литере 0 соответствует нулевой потенциал, т. е. $u = 0$. Значения U_1 и 0 являются номинальными, реальный же приемник сигналов должен фиксировать поступление литеры 1 при значениях u на его входе, близких к U_1, а поступление литеры 0 при малых по сравнению с U_1 значениях.

Избирательность приемника можно количественно охарактеризовать допуском k, $0 < k < 0{,}5$: при $u/U_1 < k$ сигнал воспринимается как литера 0, при $u/U_1 > 1 - k$ — как литера 1, а при $k \leqslant u/U_1 \leqslant 1 - k$ приемник не обеспечивает уверенного опознавания сигнала. Чтобы гарантировать дискретное функционирование, следует исключить неопознаваемые сигналы, т. е. установить для источников сигналов более строгий, чем для приемника, допуск k_c: $k_c < k$. При этом генерируемые источниками сигналы будут обладать значениями, достаточно близкими к номинальным: $u/U_1 < k_c$ или $u/U_1 > 1 - k_c$, чтобы заведомо опознаваться приемником. Дискретность обеспечивается за счет «разрыва» в промежутке от $k_c U_1$ до $(1 - k_c)U_1$.

В действительности, конечно, никакого разрыва нет: функция $u(t)$, как было сказано, является непрерывной и при изменении

от 0 до U_1 пробегает все промежуточные значения. Но время в компьютере также дискретно, а именно: интервалы, на которых приемник воспринимает установившиеся значения сигналов, отделены друг от друга «разрывами»— интервалами, в течение которых происходит изменение, установление значений сигналов. Элементы устройств компьютера, как правило, представляют собой сочетание приемник/источник — по входу элемент является приемником, а по выходу работает как источник сигналов. При этом фазы входа и выхода противоположны: интервал, на котором приемник воспринимает установившееся значение входного сигнала, является интервалом изменения сигнала, генерируемого источником на выходе элемента. Соединенные друг с другом элементы образуют двухфазную систему так, что выходы всех четных и входы нечетных элементов работают в одной фазе, а выходы всех нечетных и входы всех четных — в противоположной. Путем подобных ухищрений на основе совершающихся в непрерывном времени непрерывных физических явлений осуществляется дискретно действующий компьютер.

Простота технической реализации двоичного алфавита оборачивается маломощностью двоичного кода. Для представления десятичной цифры требуется четыре двоичных, например: цифры 1, 2, 9 в двоичной записи изображаются соответственно словами 0001, 0010, 1001. Для представления литер алфавита, включающего русские и латинские буквы, цифры и другие необходимые печатные знаки, приходится использовать по 7, 8 и более двоичных литер. Так, при вводе текста в компьютер с клавиатуры каждое нажатие клавиши порождает соответствующую этой клавише с учетом состояния регистров, в котором находится клавиатура, комбинацию из 7—8 двоичных литер. В памяти компьютера печатные знаки хранятся в виде восьмерок двоичных литер, которые при выдаче их на печатающее устройство претерпевают в нем обратное преобразование — печатаются как буквы и цифры.

Возможности кодирования далеко не исчерпываются представлением в компьютере письменных текстов. В принципе закодировать можно все что угодно. Например, графическое изображение произвольного вида можно с любой желаемой точностью представить совокупностью строк, каждая из которых является последовательностью черных и белых точек. Сопоставив белой точке, скажем, 0, а черной — 1, получаем возможность записать строки, на которые разложено изображение, в память компьютера и выполнять над ними те или иные преобразования, а затем отображать их на электронно-лучевой экран или на растровое печатающее устройство. Если требуется обрабатывать цветные изображения, то каждой точке сопоставляют не одну двоичную литеру, а несколько: предоставив

по три литеры на точку, можно кодировать восемь оттенков цвета, четырьмя литерами — 16 оттенков и т. д.

Другое замечательное качество компьютеров — их способность автоматически осуществлять самоуправляющиеся последовательности операций (процессы) — обеспечивается на основе принципа хранимой программы. *Программа* представляет собой последовательность команд, которые, будучи выполненными в надлежащем порядке, составляют определенную процедуру обработки данных, манипулирования связанными с компьютером объектами и т. п. В памяти компьютера готовая к выполнению программа хранится в виде *кода*: каждая команда закодирована, как правило, одним двоичным словом, а в отдельных случаях двумя-тремя словами. Процессор считывает команды одну за другой (в простейшем случае в том порядке, в котором они расположены в памяти), декодирует их и организует выполнение предписанных ими действий.

Процессор характеризуется *набором команд*, которые он может выполнять, которые составляют его язык, *машинный язык*. Программирование действий компьютера сводится в конечном счете к представлению этих действий последовательностями команд, выполняемых процессором, т. е. к выражению их на машинном языке данного компьютера. Каждая компьютерная команда указывает *операцию*, которую должен выполнить компьютер, и, как правило, объекты этой операции — *операнды*. Но имеются команды и без операндов или с подразумеваемыми операндами, например: «СТОП» значит «Остановить процессор».

Хотя компьютерные команды нередко называют инструкциями, в них больше подразумевается, чем содержится явно. Типичный пример: для процессора, выполняющего операции преобразования и тестирования данных в *аккумуляторе* — регистре, содержащем один из операндов, замещаемый затем результатом операции,— команда «Сложить А» означает: «Прочитать код числа, хранящийся в ячейке А запоминающего устройства, другое число взять из аккумулятора, образовать сумму этих чисел и поместить код суммы в аккумулятор, установив значения признаков, характеризующие полученный результат — нуль/не нуль, минус/не минус, перенос 1/перенос 0, переполнение/не переполнение, а затем считать и декодировать очередную команду».

В наборе команд процессора имеются команды ввода и вывода, а также пересылки данных между внутренними устройствами компьютера, преобразования данных путем выполнения арифметических и неарифметических операций, тестирования данных, в частности результатов операций, на удовлетворение тем или иным условиям (отношениям) с целью выбрать одно из предусмотренных в программе продолжений процесса, команды управления последовательностью

выполнения программы, обращения к обособленным ее частям — подпрограммам, управления работой органов процессора и внешних устройств и др. Наборы команд процессоров могут значительно различаться как по составу и характеру операций преобразования данных, так и по предоставляемым средствам управления. Необходимо, однако, чтобы данные средства отвечали требованию алгоритмической полноты, т. е. позволяли запрограммировать процесс произвольного вида.

В современных микрокомпьютерах управление процессом или ходом выполнения программы основано на *условном переходе*. Память компьютера организована как последовательность пронумерованных ячеек, и процессор, обращаясь к памяти за командой или данными, сообщает запоминающему устройству номер (адрес) ячейки, в которой хранится требуемая команда или данные. На линейных участках программы команды выполняются в порядке возрастания их адресов. В составе процессора имеется так называемый *программный счетчик*, или *счетчик команд*, значение которого после считывания очередного слова программы автоматически увеличивается на 1, составляя адрес следующей по порядку номеров за выполняемой в текущий момент команды. Переход с одного линейного участка на другой осуществляется *командой перехода*, при выполнении которой на программный счетчик устанавливается указанный в ней адрес начальной команды нового линейного участка. Как правило, команда перехода содержит условие, при выполнении которого осуществляется переход. Если это условие не выполнено, переход не производится — установки нового адреса на программный счетчик не происходит, продолжается прежняя линейная последовательность команд. Имеется также команда безусловного перехода, вызывающая переход независимо от каких-либо условий.

Примеры команд условного перехода: «Если значение аккумулятора равно 0, перейти по адресу А», «Если на К-м периферийном устройстве включен сигнал готовности, перейти по адресу А».

В сочетании с автоматическим приращением значения программного счетчика на линейных участках условный переход является алгоритмически полным и весьма экономным средством управления ходом программы. Выбор в зависимости от заданного условия того или иного продолжения процесса обеспечивается условным переходом на участок программы, составляющий альтернативу дальнейшему продолжению выполняемого участка. Проиллюстрируем это фрагментом программы, осуществляющим тестирование значения аккумулятора Ac и в случае Ac $<$ 0 увеличение, а в случае Ac $>$ 0 уменьшение этого значения на 1. Каждая строка содержит одну команду и начинается адресом этой команды.

```
200 Если Ac = 0, перейти по адресу 205.
201 Если Ac > 0, перейти по адресу 204.
202 Увеличить Ac на 1.
203 Перейти по адресу 205.
204 Уменьшить Ac на 1.
205 Продолжение программы.
```


Осуществление так называемого *цикла*, т. е. повторного выполнения участка программы, достигается использованием в конце этого участка команды перехода на его начало. Следующий фрагмент иллюстрирует это на примере цикла ожидания: процессор тестирует в цикле *флажок готовности* F устройства ввода (F = 0, пока устройство не готово передать вводимое значение, а когда готово, то F = 1) и, как только готовность наступает, производит пересылку введенного значения в аккумулятор, а флажок устанавливает на 0.

```
300 Скопировать F в Ac.
301 Если Ac = 0, перейти по адресу 300.
302 Скопировать введенное значение в Ac.
303 Установить F = 0.
```


Подобным образом с помощью команд условного и безусловного перехода осуществимы любые другие схемы процессирования, причем, как уже было сказано, управление процессом реализуется экономно. Действительно, на линейных участках не требуется никаких управляющих предписаний, поскольку действует соглашение о выполнении команд в порядке возрастания адреса, а в точках ветвления процесса необходима на каждую новую ветвь одна команда перехода, содержащая адрес и код условия.

Вместе с тем, как будет показано в дальнейшем, техника переходов неудовлетворительна в том отношении, что создаваемые на ее основе машинно-эффективные программы трудны для понимания и обслуживания их человеком. Трудность быстро возрастает с увеличением размеров программ и является главным препятствием в решении насущных проблем повышения продуктивности программистского труда и обеспечения гарантированной надежности программных систем. Тем не менее, техника переходов остается практически монопольным средством управления, и все имеющиеся компьютеры, в том числе все микрокомпьютеры, построены на базе этой техники.

Исключительное значение имеет способность компьютера тестировать данные и выбирать дальнейшее продолжение процесса в зависимости от исхода теста. Именно в этом «умении» принимать решение, действовать с учетом сложившейся обстановки компьютеры подобны живым существам. В более прозаической трактовке данная способность называется обратной связью: информация о текущем состоянии объекта обработки используется для управления действиями обработчика. Обратная связь широко используется в тех-

нике и помимо компьютеров, но в сочетании с программным управлением, универсальностью символьной обработки и электронной скоростью реакции она приобрела в компьютерах необыкновенную гибкость, точность и всеобщность. На основе обратной связи осуществляются итеративные процессы внутри компьютера и работа компьютера в контуре управления с внешними объектами. Обратной связи компьютеры обязаны адаптивностью, самодиагностируемостью, способностью к диалогу, своим особым положением в ряду современных средств автоматического управления, измерений, контроля качества изделий.

Подводя итог обсуждению принципиальных основ компьютерной техники, еще раз обратим внимание на чрезвычайную мощность и потенциальную беспредельность заложенных в компьютере возможностей: символическое представление объектов означает абсолютную универсальность, дискретность представления — неограниченное многообразие и точность, программируемое процессирование с использованием обратной связи — возможность автоматического функционирования, исследования и контроля окружающей среды.

§ 3. Структура микрокомпьютерной системы

Термин «микрокомпьютерная система», так же как «компьютерная система» и многие другие возникшие на практике словосочетания, строго не определен. Обычно им обозначают микрокомпьютер (или несколько микрокомпьютеров) в комплексе с периферийным оборудованием и программным оснащением для определенного применения или класса применений. Например: микрокомпьютерная система разработки программ для встраиваемых микрокомпьютеров, микрокомпьютерная система обработки текстов, микрокомпьютерная информационная система, микрокомпьютерная система автоматизированного обучения и т. п. Впрочем, когда речь идет о системе общего или вычислительного назначения, то обычно говорят не «микрокомпьютерная система», а просто «микрокомпьютер», или «компьютер», или «персональный компьютер», хотя имеется в виду полный комплекс аппаратуры с операционной системой и языками программирования. С другой стороны, название «микрокомпьютерная система программирования» обозначает систему программирования для микрокомпьютеров, т. е. язык и его программную поддержку, но не аппаратуру. В конкретном контексте, как правило, ясно, что имеется в виду. В данном случае мы пользуемся термином «микрокомпьютерная система» как собирательным для обозначения неопределенного представителя всей совокупности микрокомпьютерных систем.

В микрокомпьютерной системе, как и во всякой компьютерной системе, прежде всего различают *аппаратуру* и *программное оснащение* (программное обеспечение). Аппаратура составляет ту материальную основу, на которой выполняются программы, осуществляющие функции развития и специализации системы, а также управление работой аппаратуры. По традиции аппаратуру, в противоположность программам, принято считать жесткой, неизменяемой частью системы. Это верно в том отношении, что на фиксированной конфигурации аппаратуры можно выполнять самые разные и создавать новые программы. Что же касается изменчивости программ, то готовые, отлаженные программы менее подвержены изменениям, чем аппаратура, и можно указать множество примеров создания новых компьютеров под существующее программное оснащение.

Дело в том, что разработка программ крайне трудоемка, причем в то время как стоимость аппаратуры с усовершенствованием технологии и автоматизацией производства неуклонно и быстро снижается, удешевить создание программ не удается. В микрокомпьютерных системах стоимость программного оснащения нередко в десятки раз превышает стоимость аппаратуры. И вполне закономерной представляется тенденция перепоручать аппаратуре все больше функций, осуществлявшихся до того программным путем, а также фиксировать программы в аппаратуре путем неизменяемой записи в постоянные запоминающие устройства (так называемые «кремниевые», т. е. поставляемые в виде кремниевых ПЗУ, операционные системы).

Как видно, разделение функций между аппаратурой и программами не является ни однозначным, ни неизменным. Более того, такие «гибриды» как программы в ПЗУ или микропрограммы в управляющей памяти процессора невозможно безоговорочно отнести ни к аппаратуре, ни к программному оснащению.

Наряду с разделением компьютерной системы на аппаратуру и программы, в ней различают *внутреннюю* и *внешнюю* (*центральную* и *периферийную*) части. К внутренней части относится центральный процессор или комплекс процессоров и непосредственно взаимодействующие с ним органы: главная (внутренняя, первичная) память, так называемые порты ввода/вывода и отчасти блоки управления (контроллеры) внешних устройств, специализированные процессоры, а также сопрягающая все эти органы с центральным процессором и друг с другом так называемая системная магистраль. Короче говоря, внутренняя часть системы — это собственно компьютер (микрокомпьютер). Конструктивно она, в зависимости от назначения системы, может представлять собой настольный или портативный прибор с встроенным источником питания либо плату

с разъемом для установки ее в блок или в стойку вместе с другой электронной аппаратурой, либо, наконец, микросхему в стандартном корпусе, рассчитанном на включение в соответствующую панельку или на припайку.

Внешнюю часть микрокомпьютерной системы составляют устройства периферийной (вторичной) памяти, ввода/вывода данных, а также аппаратура сопряжения с различного рода внешними объектами — источниками сигналов, исполнительными механизмами, линиями связи и т. п. Если микрокомпьютер оформлен в виде отдельного прибора, внешние устройства, также в приборном исполнении, подключаются к нему посредством кабелей. Одноплатные же и однокристальные микрокомпьютеры скорее сами бывают включены в свое внешнее окружение и получают от него не только управляющие сигналы и данные для обработки, но и необходимое электропитание. Контроллеры, управляющие работой внешних устройств и осуществляющие взаимодействие их с центральным процессором, территориально могут располагаться как при внешних устройствах, так и внутри компьютера или частью в компьютере и частью в управляемом устройстве.

Каждое внешнее устройство в отношении центрального процессора характеризуется особым протоколом взаимодействия, определяющим порядок пуска и останова механизмов, обмена данными и служебной информацией, обнаружения и исправления ошибок, сигнализации занятости/готовности и т. п. Соблюдение этих протоколов со стороны центрального процессора обеспечивается специальными обслуживающими программами — *драйверами* (т. е. «водителями») внешних устройств. Таким образом, для подключения к компьютеру нового периферийного устройства необходимы соответствующий аппаратный контроллер и программа-драйвер, обеспечивающая центральному процессору возможность взаимодействовать с контроллером, а посредством него и с самим устройством.

Драйверы внешних устройств являются, можно сказать, периферийными программами, т. е. программным оснащением периферийной части компьютерной системы. Однако, в отличие от аппаратуры, программное оснащение не разделяют на центральное и периферийное или внутреннее и внешнее. Программы разделяются на *системные* и *прикладные*. Это разделение настолько существенно, что и программисты делятся соответственно на системных и прикладных.

Назначение системных программ — обеспечить функционирование компьютерной системы как единого целого и создать надлежащую основу для разработки и выполнения прикладных программ. Компьютерные и в особенности микрокомпьютерные команды слишком элементарны для того, чтобы непосредственно из них можно

было эффективно строить программы, реализующие практические приложения компьютеров (хотя в отдельных случаях идут и этим путем). Кроме того, не оснащенный программами компьютер воспринимает и выполняет только команды, представленные двоичным кодом, неблагоприятным для человека, даже если тройки двоичных цифр заменить восьмеричными или четверки — шестнадцатеричными цифрами. Впрочем, использование восьмеричного или шестнадцатеричного кода, вводимого, например, с клавиатуры, уже предполагает наличие драйвера клавиатуры и какой-нибудь программы-монитора, обеспечивающей минимальные возможности управления самим компьютером, подобно тому как драйверы обеспечивают управление внешними устройствами. Короче говоря, микрокомпьютер с дисплеем, клавиатурой и дисками становится той понятливой, предупредительной и исполнительной системой, с которой легко и удобно работать лишь после того, как системные программисты возведут над его набором команд многоярусную программную надстройку.

Системное программное оснащение состоит из двух главных частей: 1) средств управления работой компьютерной системы, которые в наиболее развитой форме образуют так называемую *операционную систему*; 2) средств, позволяющих использовать компьютер для повышения эффективности программистского труда при разработке исходного текста программы, переводе программы в машинный код, отладке, тестировании и документировании. В совокупности с языком, на котором создаются программы, эти средства составляют *систему программирования* (систему автоматизации программирования). Компоненты системы программирования, впрочем, как и любые другие программы, работают под управлением операционной системы, обеспечивающей возможность обращения к ним, подобно обращению к компонентам аппаратуры, например, к печатающему устройству или к магнитному диску.

Одной из важнейших функций операционной системы является управление периферийными устройствами, в частности, манипулирование так называемыми *файлами*. Ввод, вывод и сохранение во внешней памяти программ и данных осуществляется в виде файлов — поименованных совокупностей соответствующих текстов. Это вполне аналогично хранению документов в папках, у которых на корешках имеются надписи, идентифицирующие содержимое папок. Система должна обеспечивать возможность создания файлов, доступ к ним по именам для записи и чтения содержимого, пересылки в главную память, из главной памяти, а также между устройствами ввода/вывода и т. п.

В зависимости от назначения микрокомпьютерной системы ее оснащают операционной системой того или иного типа. Так, разни-

чают операционные системы, предназначенные для разработки программ, поддерживающие те или иные языки программирования, и операционные системы для выполнения программ в реальном времени, обеспечивающие быстрое обслуживание запросов извне и параллельное выполнение нескольких программ.

§ 4. Аппаратура микрокомпьютерных систем

Центральным органом микрокомпьютерной системы является микропроцессор. Физически он представляет собой большую (БИС) или сверхбольшую (СБИС) интегральную схему — тонкую пластинку кристаллического кремния в форме прямоугольника со сторонами размером от 3 до 7 мм, на которой размещены десятки — сотни тысяч транзисторов, реализующих все функции центрального процессора компьютерной системы: выборку, декодирование и организацию выполнения команд, преобразование и тестирование данных, обращение к главной памяти и к периферийным устройствам, обработку запросов прерывания и т. д. Пластинка вмонтирована в пластмассовый или керамический плоский корпус шириной 10—15 мм и длиной 20—70 мм. Вдоль длинных сторон корпуса расположены выводы в количестве 16—60 или более для соединения микропроцессора с другими устройствами. Существуют также микропроцессоры, выполненные на нескольких кристаллах, в частности, так называемые секционированные микропроцессоры, у которых в зависимости от примененного количества секций варьируется длина машинного слова.

Основными характеристиками микропроцессора являются следующие:

— *длина машинного слова*, т. е. число двоичных литер (*битов*), обрабатываемых микропроцессором в один прием, в качестве одного операнда;

— *объем адресного пространства* — диапазон значений адресов главной памяти, которую способен адресовать микропроцессор, способы адресации памяти;

— количество и схема организации *внутренних регистров*;

— количество *портов ввода/вывода* и их пропускная способность, характер механизма *прерываний*;

— *набор выполняемых команд*, в том числе команд управления ходом программы;

— *быстродействие*, выражаемое количеством операций, производимых в 1 с, или средней длительностью цикла команды, или длительностью элементарного цикла (такта), или тактовой частотой.

Кроме того, важное значение имеют такие общетехнические характеристики как потребляемая (рассеиваемая) мощность, устойчивость в отношении нестабильности электропитания и параметров окружающей среды, помехоустойчивость, средний срок безотказной работы и, конечно, цена.

Микропроцессоры — один из самых молодых, стремительно развивающихся и безусловно перспективных видов техники. Первый микропроцессор появился в 1971 г. — четырехбитный Intel 4004, содержавший 2250 транзисторов на кристалле размером 2,8 × × 3,8 мм и поставлявшийся в комплекте из 4-х кристаллов, в который входил кристалл ПЗУ на 256 8-битных слов и кристалл оперативной памяти емкостью 32 бита. Менее чем через год начался выпуск 8-битного микропроцессора Intel 8008, а к концу 1973 г. был создан Intel 8080 — быстродействующий 8-битный процессор, адресующий 64К слов памяти (К = 1024). Именно с него началось массовое производство и широкое применение микрокомпьютерной техники. В дело включились десятки фирм, технология быстро совершенствовалась, выпуск отдельных моделей 4-битных и 8-битных микропроцессоров составил миллионы и даже десятки миллионов экземпляров в годовом исчислении. Происходившее с наращиванием объема производства быстрое снижение цен открывало все более широкие возможности применения, что в свою очередь увеличивало спрос и подхлестывало производство. Это было повторением эскалации, происходившей несколькими годами ранее в промышленности миникомпьютеров, но повторением в значительно больших масштабах.

К концу 70-х годов, наряду с улучшенными 4-битными и 8-битными моделями, выпускались 16-битные микропроцессоры, сравнимые по производительности с процессорами миникомпьютеров и существенно превосходящие их по другим технико-экономическим параметрам. Дальнейшее развитие привело к 32-битным микропроцессорам с быстродействием более миллиона операций в секунду и адресным пространством в несколько миллионов слов, что соответствует параметрам наиболее мощных процессоров больших компьютеров недавнего прошлого, например, IBM 370/158. Разумеется, микропроцессоры этого класса не так дешевы, как 8- или 16-битные, и выпускаются значительно меньшими тиражами: к тому же и применимость их с надлежащей эффективностью намного труднее. К сожалению, наращиванию мощности постоянно сопутствует увеличение сложности микропроцессора, и в преодолении этой роковой связи реальных достижений пока не видно.

Внутреннюю организацию микропроцессоры унаследовали от миникомпьютеров, причем по мере роста числа транзисторов, размещаемых на кристалле, организация эта становится все более сложной, начиная с одноаккумуляторной при минимуме специализированных регистров (программный счетчик, указатель стека, ин-

декс-регистр) до набора регистров общего назначения, численность которых продолжает увеличиваться. Увеличиваются и усложняются также наборы команд и способов адресации. Короче говоря, происходит повторение пути развития миникомпьютеров, причем без сколько-нибудь существенных усовершенствований, а порой и в худшем варианте. Отдельные призывы упрощать процессор (например, наглядно показавшая преимущества простоты разработка RISC — Reduced Instruction Set Computer) не находят понимания и должной поддержки. Теоретические рассуждения о преимуществах стековых компьютеров остаются на бумаге — практически стеки используются в микропроцессорах главным образом для сохранения адресов возврата из подпрограмм.

На фоне поразительных технологических достижений полупроводниковой микроэлектроники, обеспечивших стремительный прогресс в части физических характеристик микропроцессоров, развитие принципов эффективного процессирования представляется находящимся в состоянии полного застоя. Создаваемые в результате усовершенствования технологии новые технические возможности употребляются на все большее усложнение микропроцессоров ради дальнейшего наращивания их «сырой» мощности, хотя проблема давно уже не в недостатке этой мощности, а в сложности ее эффективного использования.

Главная память микрокомпьютеров реализуется на той же физико-технической основе, что и микропроцессоры, и ее характеристики улучшаются такими же быстрыми темпами.

Основными параметрами памяти являются *емкость* и *быстродействие*. Емкость запоминающего устройства — это количество составляющих его битов. Но на практике емкость памяти выражают обычно не в битах, а в байтах: *байт* — это восемь битов. Емкость памяти конкретных компьютеров характеризуют также числом машинных слов, но при этом следует указывать длину слова в битах или в байтах, поскольку машинные слова бывают разной длины, например: 4 бита (полбайта, или нибл), 1, 2, 3 и более байтов. Байт — сравнительно малая единица: восьми битам соответствует 2^8, т. е. 256, различных значений; байт используют как внутренний эквивалент литеры внешнего алфавита (алфавита ввода/вывода). Емкость главной памяти современных микрокомпьютеров равна, как правило, десяткам — сотням тысяч байтов, а у так называемых супермикрокомпьютеров составляет миллионы и десятки миллионов байтов. Однако точное число байтов, составляющих память двоичного компьютера, естественно, равно целой степени двойки, поэтому употребляемые обычно единицы килобайт (КВ) и мегабайт (МВ) означают соответственно двоичную тысячу ($2^{10} = 1024$) и двоичный миллион ($2^{20} = 1024 \times 1024 = 1048576$) байтов.

Главная память микрокомпьютера может быть скомпонована из оперативных (ОЗУ) и постоянных (ПЗУ) запоминающих устройств. ОЗУ обеспечивают возможность как считывания, так и записи в память двоичных кодов, причем в одинаковом темпе. ПЗУ — память только для чтения того содержимого, которое заложено в нее в процессе изготовления запоминающего устройства. Существуют также программируемые пользователем так называемые «полупостоянные» запоминающие устройства (ППЗУ) и запоминающие устройства, допускающие повторную запись после стирания прежнего постоянного содержимого ультрафиолетовым облучением (СППЗУ — стираемые программируемые ПЗУ). В отличие от ПЗУ и ППЗУ, в которых запись осуществляется напылением или выжиганием перемычек, так сказать, навечно, в СППЗУ значения битов определяются путем создания изолированных электрических зарядов, которые при неблагоприятных условиях могут исчезать, т. е. это все-таки временная память.

Несмотря на то, что постоянная память не обеспечивает возможности оперативного манипулирования ее содержимым, она крайне важна в микрокомпьютерных системах и широко в них используется. Дело в том, что современные полупроводниковые устройства оперативной памяти имеют существенный недостаток — записанное в них не сохраняется при выключенном электропитании. Компьютер, укомплектованный одной только оперативной памятью, при выключении питания «забывает» все, чему научился в виде накопленных в памяти программ, и после включения питания оказывается совершенно беспомощным — все надо начинать с нуля, с ввода вручную начального загрузчика. Если же этот загрузчик находится в имеющейся в составе компьютера постоянной памяти, то вводить его не требуется. В постоянной памяти целесообразно хранить и многие другие программы, например, тесты для контроля исправности аппаратуры, монитор, обеспечивающий возможность управления компьютером с клавиатуры, а в ряде случаев и интерпретатор языка программирования. Таким образом, постоянная память позволяет развивать «способности» компьютера, повышать уровень его «интеллекта».

Не менее внушительны преимущества хранения в постоянной памяти прикладных программ в специализированных микрокомпьютерных системах, встраиваемых в машины и приборы: исключается необходимость загрузки, значительно повышается надежность функционирования и устойчивость по отношению к условиям окружающей среды, упрощается обслуживание. Емкость ПЗУ в одноплатных микрокомпьютерах обычно в несколько раз больше емкости ОЗУ, а в однокристальных микрокомпьютерах ПЗУ обладает емкостью, в 10—20 раз превышающей емкость ОЗУ.

Быстродействие запоминающих устройств характеризуют так называемым *циклом памяти* — минимальным промежутком между обращениями. Величиной, обратной циклу, является максимально допустимая частота обращений к памяти. Величина цикла в большой степени зависит от используемой полупроводниковой технологии и режима запоминающих элементов. В запоминающих устройствах микрокомпьютеров, так же как и в микропроцессорах, используются, как правило, характерные малым потреблением мощности полупроводниковые структуры типа МОП (металл-оксид-полупроводник), обеспечивающие значения цикла памяти порядка десятых долей микросекунды. В наиболее популярных ОЗУ динамического типа, обходящихся минимумом транзисторов, но требующих периодической регенерации хранимого кода, значение цикла составляет обычно 0,4—0,6 мкс.

Количество битов, размещаемых на одном кристалле памяти (в одной БИС), со времени появления первых микропроцессоров увеличивается в среднем в 4 раза каждые два года, в результате чего последовательно появились кристаллы емкостью 256, 1024 (т. е. 1К), 4К, 16К, 64К и 256К битов. Пропорционально увеличивалась и емкость запоминающих устройств микрокомпьютеров, поскольку размеры и цена микросхем памяти при увеличении ее емкости сохранялись практически неизменными.

Кристаллы памяти различаются не только по количеству размещенных на них битов, но и по типу их организации. Так, для кристаллов ОЗУ типична организация, обеспечивающая побитную выборку, т. е. каждый бит обладает собственным адресом и обращение по адресу есть обращение к соответствующему одному биту. В запоминающем устройстве с параллельной выборкой n-битного слова каждому биту слова сопоставляется свой кристалл, и, таким образом, n кристаллов обеспечат емкость устройства в словах, численно равную емкости кристалла в битах. Например, использовав 8 16К-битных кристаллов, получаем память емкостью 16К байтов. Если же требуется память большей емкости, то добавляют по n кристаллов нужное число раз. Например, чтобы на 16К-битных кристаллах получить 64К байтов, следует взять четыре комплекта по 8 кристаллов: $16 \times 4 = 64$.

Для кристаллов ПЗУ типична организация, обеспечивающая побайтную выборку. При этом кристалл содержит линейно адресуемую последовательность байтов, т. е. покрывает непрерывный участок адресного пространства, так что размещенная на этом участке программа записывается в одном (возможно, сменном) кристалле ПЗУ. Число байтов, естественно, в 8 раз меньше числа имеющихся на кристалле битов, например, 64К-битный кристалл в случае побайтной выборки становится 8К-байтным.

Несмотря на различия в организации, кристаллы ОЗУ и ПЗУ совместимы, т. е. способны совместно работать в едином адресном пространстве микропроцессора. Совместимости не препятствует и различие в быстродействии кристаллов, потому что связь между процессором и памятью, равно как и между другими компонентами микрокомпьютерной системы, осуществляется посредством асинхронно действующей магистрали. Асинхронность означает, что взаимодействующие устройства не обязаны иметь единое тактирование — обмен сигналами производится по принципу «рукопожатия»: устройство, передающее сигнал, не снимает его с линии, пока не получит от принимающего подтверждения о приеме. При этом каждый может работать в свойственном ему темпе.

Системная магистраль является гибким средством сопряжения микрокомпьютерной системы, позволяющим сравнительно просто осуществлять изменения ее конфигурации путем добавления или замены устройств, в том числе главной памяти и даже самого центрального процессора. Физически магистраль представляет собой многопроводную линию с гнездами для подключения печатных плат, а в одноплатном компьютере — микросхем. В состав магистрали входят группы проводов адреса, данных, управления и электропитания. Память подключаемых к магистрали устройств адресуема, т. е. доступ к нужному регистру или ячейке памяти того или иного устройства обеспечивается при наличии на адресных проводах соответствующего адреса.

Активный абонент магистрали (например, центральный процессор) при обращении к данному регистру устанавливает на адресные провода код его адреса, причем, если в регистр должна быть произведена передача данных, то передаваемый код устанавливается на проводах данных, после чего на провода управления выдается сигнал, обозначающий действие: «запись» или «чтение». В случае записи код с проводов данных принимается в указанный адресом регистр и при этом устройство, принявшее код, выдает сигнал на управляющий провод «подтверждения» действия. Если же производится чтение, то регистр выдает на провода данных копию хранящегося в нем кода в сопровождении управляющего сигнала, уведомляющего запросчика о выполнении чтения.

Существенно, что абоненты магистрали различаются не по месту подключения к ней, а по адресам, присвоенным их регистрам и ячейкам памяти. Благодаря этому можно в любое гнездо магистрали включить любое из рассчитанных на работу с ней устройств. К сожалению, попытки стандартизации системной магистрали пока не увенчались успехом. В настоящее время практически каждое семейство микрокомпьютеров имеет свою собственную системную магистраль, которая в каком-то отношении «лучше других».

Периферийная аппаратура представлена на магистрали интерфейсными регистрами контроллеров внешних устройств (портами ввода/вывода). Через эти сопрягающие внутреннюю и внешнюю части системы регистры производится передача как собственно данных, так и управляющей, а также контрольной или диагностической информации. Таким образом, взаимодействие с периферией сводится к стандартным операциям чтения и записи по данному адресу. Однако в отличие от главной памяти, являющейся пассивным абонентом магистрали, периферийные устройства, вообще говоря, выступают в качестве активных абонентов, наделенных правом захватывать управление магистралью и инициировать взаимодействие с другими ее абонентами.

Устройство с простой логикой функционирования, такое как клавиатура или ленточный перфоратор, может быть представлено на магистрали парой регистров: один — для передачи данных, другой — для управления. Контроллеры устройств, работающих в режиме прямого доступа к памяти, например магнитных дисков, представлены несколькими регистрами. Такие сложные контроллеры обычно рассчитаны на поочередное обслуживание нескольких однотипных устройств. Существуют также контроллеры, обеспечивающие посредством специальной периферийной магистрали обслуживание целого комплекса приборов, удовлетворяющих требованиям стандарта данной магистрали. Наконец, имеются контроллеры, реализующие стандартное сопряжение системной магистрали с линиями связи, благодаря которому микрокомпьютер может работать в компьютерной сети, обмениваться сообщениями с другими компьютерами.

Набор периферийных устройств, которым укомплектовывается микрокомпьютерная система, определяется ее назначением и способом использования, причем нельзя не считаться с тем, что многие периферийные устройства существенно дороже самого микрокомпьютера и требуют особого технического обслуживания. Например, применение построчного принтера с высококачественной печатью или жесткого диска значительной емкости оправдано, как правило, только для обслуживания в режиме разделения времени нескольких микрокомпьютеров, объединенных в локальную сеть. Проблема периферийного оборудования индивидуальных микросистем по существу еще не решена. Серьезное значение она приобрела лишь в начале 80-х годов, после того как стало ясно, что персональный компьютер при надлежащем оснащении является высокоэффективным профессиональным инструментом. Появление малогабаритных и непритязательных к условиям эксплуатации магнитных дисков типа «Винчестер» с быстрым доступом и емкостью 10—20 МВ открыло в этом деле невиданно широкие перспективы.

Создатели персонального компьютера вышли из положения, воспользовавшись дешевым серийным телевизором в качестве дисплея и гибким диском или кассетным магнитофоном в качестве внешней памяти. Это обеспечило для микрокомпьютеров новый огромный рынок и небывало расширило доступ к компьютерам, сделало их предметом массового пользования. Если допустить, что персональные компьютеры употребляются главным образом для весьма несложных применений, а чаще всего как средство развлечения, то комплектование их бытовыми приборами приемлемо. Но с точки зрения современного пользователя-профессионала периферийные устройства персонального компьютера совершенно неудовлетворительны.

Гибкий диск (а тем более магнитофон) может служить средством ввода/вывода программ и данных, но не запоминающим устройством для хранения оперативно используемых файлов: время доступа у него в 10—20 раз больше, чем у настоящих (жестких) дисков, емкость, не превышающая обычно 0,5—1 МВ, оказывается одного порядка с емкостью главной памяти микрокомпьютера, надежность и срок службы в режиме частых обращений крайне малы, так как запись/считывание производятся контактным способом, т. е. сопряжены с истиранием магнитного покрытия и головки. Телевизор в качестве дисплея также не удовлетворяет современным требованиям к устройствам этого рода, однако нет принципиальных препятствий для создания массового дисплея, который явился бы приемлемым компромиссом технических требований и цены.

Особую проблему представляют принтеры и плоттеры (построители графических изображений). Имеется широкий ассортимент устройств этого рода, созданных для миникомпьютеров, но по микрокомпьютерным меркам они дороги и неуклюжи, подобно тому, как были дороги и громоздки на миникомпьютерах принтеры от больших вычислительных машин. Подходящими для массового использования представляются растровые (точечные) принтеры/плоттеры, в которых универсальность сочетается с низкой стоимостью, несложностью обслуживания и приемлемым для большинства применений качеством печати. При этом изображение строится тем же способом, что и на получивших наибольшее распространение растровых дисплеях.

Оборудование, подключаемое к микрокомпьютерам, работающим в разного рода измерительных и управляющих системах (мультиплексоры, преобразователи, датчики и т. п.), отмечено спецификой не столько микрокомпьютеров, сколько самих этих систем, принадлежностью которых оно по существу является.

§ 5. Программное оснащение микрокомпьютеров

Подавляющее большинство микрокомпьютеров работает в качестве *контроллеров*, реализующих строго определенную в каждом конкретном случае логику управления, реагирования на поступающие извне или получаемые путем опроса сигналы и значения параметров объекта управления. До появления микрокомпьютеров подобные контроллеры создавались, как правило, в виде непрограммируемых автоматов, реализуемых на транзисторах или микросхемах малой интеграции и электромагнитных реле. Программное оснащение такого микрокомпьютера представляет собой обычно неизменяемую программу, записанную в постоянную память. В другом случае это может быть ряд фиксированных программ, загружаемых в память микрокомпьютера поочередно.

Когда логика управления несложна, программа может быть небольшой и ее нетрудно написать и отладить даже непосредственно в коде команд микропроцессора, как это вначале и делали. Но возможности микропроцессоров, а с ними и размеры программ быстро возросли, потребовав эффективных средств программирования. Эти средства были позаимствованы, естественно, у миникомпьютеров, которые в свое время также начинали с того, что работали в качестве контроллеров. Первым делом были созданы системы программирования на языке ассемблера, т. е. на языке машинных команд, но записываемых не в кодах, а в мнемонических обозначениях, и не с конкретными числовыми адресами, а с символическими. Программирование на языке ассемблера существенно легче программирования в кодах; но по сравнению с программированием на языках более высокого уровня оно является все же сложным и трудоемким. Несмотря на это, значительную часть микрокомпьютерных программ, в особенности управляющих, приходится создавать на языке ассемблера, поскольку этот язык обеспечивает возможность наиболее полного и эффективного использования машины.

Программное оснащение системы разработки программ на языке ассемблера включает такие компоненты как редактор исходного текста программы, ассемблер, загрузчик, редактор связей, отладчики, а также монитор или операционную систему, под управлением которых эти компоненты работают. Необходимые для реализации такой системы компьютерные ресурсы сравнительно велики и осуществить ее в удобном для практического использования варианте удается только на большой вычислительной машине или на миникомпьютере, оснащенном необходимым периферийным оборудованием — жесткими дисками, алфавитно-цифровым дисплеем, принтером и ленточным перфоратором, кассетной магнитной лентой или гибким диском для вывода готовой программы в машинном коде.

Систему программирования, работающую на компьютере иной модели, чем та, для которой предназначаются разрабатываемые в этой системе программы, называют *кросс-системой*. Помимо перечисленных выше компонент кросс-система включает интерпретатор языка команд того микрокомпьютера, на котором выполняются созданные в ней программы, а ее ассемблер является *кросс-ассемблером*. Микрокомпьютер, для которого создаются программы в кросс-системе, называют целевым. Существуют также кросс-системы программирования на языках высокого уровня, в них преобразование исходной программы в машинный код существенно более сложное, чем ассемблирование, осуществляется кросс-компилятором.

В отличие от кросс-системы, система программирования, работающая на компьютере той же модели, что и целевой, называется *резидентной* системой программирования. Такие системы функционируют обычно на микрокомпьютерах, укомплектованных гибкими дисками, но бывают и бездисковые, перфолентные системы, в которых ввод каждой компоненты приходится повторять снова и снова при каждом ее использовании. Достоинством же резидентной системы является то, что разработка и отладка программ проводится на таком же микропроцессоре, какой будет эти программы выполнять. В дополнение к стандартным периферийным устройствам система разработки укомплектовывается *программатором* — устройством, производящим запись кода готовой программы в ППЗУ целевого микрокомпьютера, и может быть укомплектована так называемым *внутрисхемным эмулятором*, при помощи которого возможна отладка компонент целевой микросистемы под управлением и с привлечением ресурсов системы разработки (инструментальной системы).

Языком программирования в резидентной системе может быть, как и в кросс-системе, язык ассемблера или язык высокого уровня, обычно фортран, или паскаль, или один из специально разработанных для микрокомпьютеров языков, например, модула, PL/M, PL/Z, MPL и др. Операционные системы, как поддерживающие разработку, так и обеспечивающие функционирование программ на целевых микрокомпьютерах, являются, как правило, машинозависимыми, а точнее, создаются особо для каждого семейства микрокомпьютеров. Однако существует настойчивая тенденция стандартизации микрокомпьютерных операционных систем — фактическим стандартом для 8-битных микрокомпьютеров стала однопрограммная дисковая операционная система CP/M. В отношении 16-битных машин на такое же положение претендуют MS DOS и аналоги популярной миникомпьютерной операционной системы UNIX. Стандартизация операционных систем позволит улучшить положение с переносимостью пакетов программ с одних

микрокомпьютеров на другие. Кроме того, пользователям не надо будет адаптироваться на каждом компьютере к новой операционной системе.

Применительно к микрокомпьютерам операционные системы должны и могут быть значительно упрощены даже по сравнению с теми, какими они стали на миникомпьютерах. С одной стороны, у микрокомпьютерных систем более массовый и менее осведомленный пользователь, а с другой стороны, нет необходимости, например, обеспечивать многозадачный режим или разделение времени между одновременно работающими терминалами, поскольку для микрокомпьютеров характерно индивидуальное использование и ресурсы их не настолько дороги, чтобы более полное их использование могло оправдать требующееся для этого усложнение. В подавляющем большинстве случаев можно обойтись двумя разновидностями микрокомпьютерных операционных систем: системой общего назначения, обеспечивающей разработку, отладку и выполнение программ с участием человека, т. е. используемой человеком в качества рабочего инструмента (инструментальной системы), и системой выполнения программ в режиме реального времени на целевом компьютере. Эта последняя должна обеспечивать мультипрограммное функционирование с интенсивным использованием системы прерывания, таймера, параллельного процессирования и т. п.

Важнейшим требованием к программному оснащению микрокомпьютерных систем является простота его освоения и использования, общедоступность. Испытанный способ удовлетворения этого требования в частных случаях заключается в узкой специализации. Так, обращению с управляемой микрокомпьютером стиральной машиной легко обучается любая домохозяйка, потому что смысл надписей и обозначений на клавишах понятен, а до того, как реализованы действия, вызываемые нажатиями этих клавиш, ей нет дела. Тот же принцип осуществлен, например, в компьютерных системах подготовки текстовых документов — пользователю предоставлен дисплей, на котором он видит набираемый или редактируемый текст и отдаваемые нажатиями клавиш команды, вызывающие обычные и непосредственно видимые на дисплее преобразования. Умение работать в такой системе никак не связано с умением программировать (если, конечно, не используется возможность вводить определяемые цепочками команд процедуры) и сводится главным образом к моторным навыкам манипулирования клавиатурой.

Можно привести много других примеров такого же рода, но и из названных ясно, что это не примеры того, как сделать компьютер доступным, а скорее, наоборот, примеры того, как использовать компьютер без непосредственного доступа к нему. В сущности, это

примеры встроенных компьютеров. При всем их огромном практическом значении, с точки зрения облегчения доступа к компьютеру они не представляют интереса. Более понятным и доступным широкому кругу пользователей компьютер может быть сделан только путем усовершенствования самой компьютерной системы — придания ей цельности, логичности, естественности. В значительной степени это определяется характером программного оснащения.

Наибольшие достижения в обеспечении доступности и простоты использования компьютерных систем связаны с появлением уже упоминавшихся персональных компьютеров. Проблема доступности программирования на *персональных компьютерах* решена предоставлением пользователю языка программирования бейсик, который задолго до появления микрокомпьютеров был разработан специально для начального обучения программированию непрофессиональных программистов: BASIC — Beginner's All-purpose Symbolic Instruction Code. Создатели бейсика стремились сделать его как можно более простым и вполне преуспели в этом, но в значительной степени ценой обеднения языка. Бейсик справедливо подвергается критике как лишенный средств структурирования программ и не обладающий возможностями развития или расширения язык — по существу, в нем нет даже понятия процедуры. Но этот слабый и неперспективный по современным представлениям язык оказывается решающим фактором успеха персональных компьютеров.

Дело в том, что наряду с отмеченными недостатками бейсику присущи также важные достоинства, не только компенсирующие, но и значительно превосходящие его недостатки. Речь идет не о его убыточной простоте, которая с увеличением объема создаваемых программ быстро переходит в свою противоположность, а о диалоговом характере и единстве бейсик-системы. Все управление системой осуществляется в форме диалога — процессор воспринимает команды непосредственно с клавиатуры и немедленно реагирует на них, как и на команды собственно языка программирования. Это коренным образом облегчает как освоение компьютера, так и любое его использование: человек имеет дело не с множеством формальных правил, которые надо знать и пунктуально соблюдать, а с бейсик-процессором, на котором можно экспериментировать, пробовать, получая наводящие указания и советы.

Конечно, диалогом нельзя оправдать порочные стороны бейсика — преимущества диалога просто покрывают их, но никак с ними не связаны. Другими словами, система программирования может быть диалоговой, включать в себя все необходимые средства управления компьютером и, вместе с тем, отвечать самым строгим требованиям в отношении структуры программ, расширяемости языка и т. п. Система такого рода будет описана в гл. 6.

ГЛАВА 2

АРХИТЕКТУРА МИКРОКОМПЬЮТЕРОВ: СПОСОБЫ ОПИСАНИЯ И ИНТЕРПРЕТАЦИЯ ДАННЫХ

§ 1. Понятие об архитектуре компьютера

Компьютер — понятие многостороннее: это и сложный схемотехнический агрегат, и универсальная логико-арифметическая модель, и большая электронная система и еще множество конструкторских, технологических, эргономических, психологических, теплотехнических, гигиенических и других проблем. Все эти стороны важны и просто необходимы. Но не схемы и электрические цепи составляют сущность компьютера и не в количествах и скоростях переключений электронных вентилей состоит производимый компьютером полезный эффект. Сущность компьютера в автоматической реализации программно-управляемых последовательностей операций — процессов.

Именно с этой стороны компьютер характеризует его *архитектура* — воплощенная в аппаратуре и встроенных программах основа программируемого процессирования, предоставляемая пользователю в виде совокупности средств представления данных, набора команд, способов адресации, протоколов взаимодействия с внешними объектами и других составляющих программно-управляемого процесса. Короче говоря, это все то, из чего и при помощи чего создаются осуществляемые на компьютере процессы, то, что надо знать о компьютере системному программисту. Это прежде всего язык команд, вместе с тем, что за ним стоит, что в нем отражено.

В понятие архитектуры входят как предоставляемые компьютером возможности реализации процессов (кодирование, запоминание и перемещение данных, операции преобразования и тестирования данных, механизмы доступа к памяти и организации последовательностей операций), так и средства, позволяющие задавать нужное использование этих возможностей, программировать требующийся процесс (коды операций, способы адресации памяти, команды, форматы данных). Существенно, однако, что все относящееся к архитектуре доступно для использования программистом при программировании компьютера. Программно недоступные, не отраженные в языке команд объекты (магистрали, буферные регистры, скрытые механизмы убыстрения доступа к памяти, схемы исправления ошибок и т. п.) к архитектуре не относятся.

Имеет место порочная тенденция отождествлять понятия архитектуры и внутренней организации компьютера. Даже в монографиях и учебниках, специально посвященных компьютерной архитектуре, нередко предмет рассмотрения сводят к блок-схемам устройств, магистралям, синхронизации сигналов и другим важным, но не имеющим отношения к архитектуре вещам.

Заметим, что компьютеры одинаковой архитектуры могут быть весьма различными в отношении внутренней организации и физической реализации. Можно указать множество примеров самых различных реализаций одной и той же архитектуры. Так, архитектура PDP-8 — родоначальница миникомпьютеров — была воплощена во многих моделях, в числе которых можно найти варианты и с магистральной, и с немагистральной организацией, и параллельного, и последовательного действия, и всевозможные конструктивно-технологические исполнения от базирующихся на «дискретных» транзисторах до монолитных микропроцессоров, причем как с вентильной логикой, так и микропрограммируемых. При всем этом разнообразии архитектура оставалась одной и той же, а следовательно, и все созданные на основе ее программы были выполняемы на всех моделях и с одинаковыми результатами.

Таким образом, показателем тождественности архитектуры компьютеров является то, что всякая программа в машинном коде, выполняющаяся на одном из этих компьютеров, выполнима на любом другом из них, причем получаемые результаты совпадают. Компьютеры, удовлетворяющие данному условию, называют обладающими одной и той же архитектурой, или совместимыми на уровне архитектуры, или совместимыми на уровне машинного языка (кода). Несмотря на то, что эти компьютеры могут значительно различаться в других отношениях, например, в отношении скорости выполнения программ, с точки зрения программистов, они совершенно одинаковы и их программное оснащение является единым для всех компьютеров.

Тождественность архитектуры компьютеров выгодна, как и всякая стандартность, во многих отношениях: возможна существенная экономия затрат на разработку и эксплуатацию программного оснащения, на освоение новых моделей машин, на обучение персонала и т. п. Однако полная унификация архитектуры всех компьютеров, на которой настаивают приверженцы неограниченной стандартизации, едва ли осуществима, и попытки насильственного проведения ее неизбежно принесут вред. Дело в том, что в зависимости от назначения компьютера требования к его архитектуре могут варьироваться в широких пределах, а кроме того, архитектуре свойственно развиваться, обновляться по мере совершенствования методов программирования, расширения возможностей аппарату-

ры и т. п. Поэтому зафиксировать навечно в качестве единой какую-то, даже «очень хорошую», архитектуру в принципе невозможно.

На практике удовлетворительное решение проблемы достигается путем компромисса. Так, семейства компьютеров с широким диапазоном мощностей и большим разнообразием применений (например, семейство DEC PDP-11) характеризуются не полной тождественностью архитектуры всех членов, а менее строгим требованием *совместимости снизу вверх*: младшие семейства реализуют архитектуру старших моделей не в полном объеме и не обеспечивают возможности выполнения всех программ, работающих на старших моделях, однако все программы, выполнимые на данной модели, выполнимы на всех старших по отношению к ней моделях.

Ясно, что с точки зрения пользователя, архитектура является главной характеристикой компьютера. Внутренняя организация — это то, как компьютер устроен, как реализованы выполняемые им функции, а архитектура — это то, что он может делать и, в известной степени, как в рамках данных возможностей заставить его делать то, что требуется.

Последнее особенно важно и определяет ведущее направление совершенствования архитектуры компьютеров — создание архитектуры, обеспечивающей легкость освоения, программирования и практического использования компьютера как можно более широким кругом людей. Появление миникомпьютеров и последующее развитие микрокомпьютерной техники на основе интегральной электроники сделали компьютер экомомически широкодоступным: компьютеры выпускаются миллионными тиражами и стали таким же предметом ширпотреба, как телевизоры или магнитофоны. Но при всем этом компьютер отличается от телевизора и магнитофона тем, что подавляющее большинство потенциальных покупателей не знает, что с ним делать, как к нему подступиться. Другими словами, экономическая доступность должна сочетаться с доступностью для понимания и эффективного применения.

Надо признать, что на пути к легко понимаемой и простой в использовании архитектуре сделано пока совсем немного. Более того, в истории развития компьютерной архитектуры прослеживается стойкая тенденция наращивания сложности ради увеличения машинной эффективности, т. е. достижения большей скорости выполнения операций и компактности программ при данных параметрах используемых физических элементов. Те упрощения, в результате которых возникла архитектура миникомпьютеров, были направлены на сокращение количества и удешевление аппаратуры путем ограничения главным образом арифметических возможностей. Они не преследовали цель сделать архитектуру более понятной, а программирование менее трудоемким — целью была минимизация стоимости самого компьютера.

Проблема создания архитектуры, обеспечивающей возможность существенного повышения продуктивности программистского труда и облегчающей доступ к компьютеру непрофессиональным программистам, назревала по мере того, как вследствие удешевления аппаратуры сфера экономически целесообразного применения компьютеров становилась всеобъемлющей. Предпринимались (и предпринимаются) попытки обойти эту проблему путем создания «дружелюбного программного оснащения» (friendly software), проблемно-ориентированных языков и других программно реализуемых «оболочек», призванных скрыть от пользователя истинный облик компьютера.

Отчасти это удается. Например, упоминавшиеся в гл. 1 диалоговые системы на основе языка бейсик позволили значительно расширить круг пользователей микрокомпьютеров. Этот впечатляющий пример показывает, как много значит простота в таком сложном деле, каким стала компьютерная техника.

Однако «недружелюбную» архитектуру надо не покрывать, а искоренять. Этого требуют прежде всего микрокомпьютеры с многообразием и специфичностью их применений. Ни бейсик, ни паскаль не устранили необходимости программирования на языке ассемблера, доступность и эффективность которого можно обеспечить только рационализацией архитектуры.

Архитектура микрокомпьютера должна быть простой для понимания, легко и эффективно расширяемой, располагающей к рациональному, структурированному программированию, обеспечивающей диалоговый режим на всех стадиях разработки и использования программ. Нельзя сказать, что разработчики компьютеров понимают важность этих требований и стремятся удовлетворить их. Поэтому желаемую архитектуру приходится пока эмулировать на имеющейся аппаратуре путем программной интерпретации соответствующего языка.

§ 2. Способы и средства описания архитектуры

Описать архитектуру компьютера — значит разъяснить смысл всех тех объектов (имен, кодов, форматов и т. п.), которыми манипулирует программист, создающий программу непосредственно в машинных кодах или на языке ассемблера. За этими символическими объектами стоят регистры процессора, структура главной памяти, способы адресации, форматы команд и данных, алгоритмы выполнения команд, механизмы обращения к подпрограммам, система прерываний, ввод/вывод и другие программно доступные атрибуты компьютера.

Традиционный способ описания архитектуры заключается в том, что приводят схему процессора с изображением его регистров и связей между ними, представляют структуру памяти и используемые способы адресации, а затем описывают команды, сообщая по каждой (или по группе единообразных) ее формат, коды операций, назначение полей и действия, производимые компьютером в процессе выполнения команды. При этом обычно вводятся такие понятия как «содержимое регистра», обозначаемое заключенным в скобки именем этого регистра, стрелки для обозначения передачи содержимого из регистра в регистр и т. п.

Как правило, подобная символика используется для сокращенного описания набора команд в виде таблицы и не дает исчерпывающей информации о действиях процессора. Словесные разъяснения, призванные пояснить значение символов и сообщить не учитываемые ими детали и особенности, не всегда достигают цели — описания страдают неполнотой и неточностью. Поэтому были созданы специальные символические языки для описания архитектуры компьютеров, не получившие, однако, сколько-нибудь широкого распространения.

Более практичным представляется использование для описания архитектуры понятий и синтаксических конструкций, заимствованных из языков программирования. В этом случае удается обойтись единой, устоявшейся и привычной программистам системой понятий и обозначений, которая при незначительном расширении вполне обеспечивает возможность компактного, недвусмысленного и исчерпывающего описания архитектуры компьютера. Примеры практического освоения данной возможности встречаются в литературе последних лет все чаще.

Необходимое применительно к условиям описания архитектуры расширение (и в известном смысле уточнение) средств, заимствуемых из арсенала программиста, состоит в том, что элементы и структуры памяти («типы данных») должны специфицироваться с точностью до бита, а операции должны определяться в терминах микроопераций, выполняемых над битами. Поэтому то, что в языке программирования считается вполне достаточным специфицировать, например, как переменную типа целое, в описании архитектуры должно быть охарактеризовано в отношении длины в битах и используемого кода (прямой, обратный, дополнительный, двоично-десятичный и т. д.). Соответственно, недостаточно определить операцию, скажем, только как сложение — необходимо дать ее полную характеристику на уровне операций над битами: как представлены числа со знаком, как фиксируется факт переполнения и т. п. Поскольку первоосновой архитектуры являются биты и операции над ними, то детализация описания до уровня битов является исчерпывающей:

в любом случае известно, какие биты и каким образом будут вовлечены в дело.

Естественно, возникает опасение, не будет ли столь подробное и обстоятельное описание настолько сложным и громоздким, что реальность его создания и практического использования окажется под сомнением. Действительно, оно не может быть простым, поскольку сама описываемая архитектура сложна, и оно не может быть проще поверхностного словесного описания, но может быть намного более обозримым и понятным, чем сопоставимое по полноте и точности словесное описание, а кроме того, в нем легче обеспечить эту полноту и точность. В решающей степени посильность как разработки, так и практического применения описания зависит от того, насколько рационально оно организовано, от логичности и четкости его структуры. Структурирование данных и алгоритмов позволяет обуздать сложность, обеспечить систематичность и ясность.

В основе структурирования лежит принцип постепенного разложения рассматриваемого объекта (в нашем случае — архитектуры компьютера) в иерархию все более мелких его частей вплоть до неделимых далее элементов (в нашем случае — битов и операций над битами). В обратном порядке это проявляется как постепенный синтез все более крупных частей создаваемого объекта с использованием синтезированных на предыдущих этапах вплоть до завершения объекта в целом. Существенно то, что сложный объект не мыслится в виде огромной совокупности примитивных элементов и неисчислимого количества связей между ними, а представлен в первом приближении как объединение немногих сравнительно автономных частей, выполняющих четко определенные функции и доступных для более подробного изучения путем анализа их собственных частей, затем частей этих частей и т. д.

Можно указать сколько угодно примеров использования данного принципа организации сложных систем как естественных, сотворенных природой, так и искусственных, созданных людьми. Он проявляется в самом строении вещества: молекулы состоят из атомов, атомы построены из электронов и ядер, которые в свою очередь разложимы на «составные части», и т. д. Толковый инженер непременно строит машину в виде легко сочленяемых блоков, которые разделяются на подблоки, сборки, узлы, что позволяет рационально организовать разработку проекта, производство, контроль качества и техническое обслуживание выпущенных изделий. Наконец, невозможно представить себе, как могла бы действовать армия, не будучи организованной в корпуса, дивизии, полки и далее во все более мелкие подразделения. (Отметим, что в этой веками выверенной структуре на каждой ступени иерархии деление производится не более чем на 3—4 части.)

Странно, но применительно к компьютерам и программированию принцип структурирования пришлось открывать заново, причем открыт он был лишь через 25 лет после появления компьютеров *) и на протяжении вот уже 15 лет не может обрести действенного воплощения на практике, хотя теоретически всеми принят, и сам термин «структурированное программирование» давно стал передовым лозунгом. Беда, может быть, в том, что в этот лозунг вкладывают различный смысл: одни называют структурированным программированием программирование без go to, другие считают, что это нисходящее проектирование программы, третьи настаивают на использовании только трех известных схем конструирования программы (линейная последовательность, ветвление по условию и цикл) и т. д. Все это, конечно, имеет определенное отношение к проблеме структурирования программы, но даже не указывает на существо этой проблемы, на то, что целью является создание программы с четкой и логичной структурой, т. е. организованной так, что ее легче понять, проверить, отладить, использовать.

Избранный нами способ описания архитектуры вполне аналогичен составлению программ. Можно сказать, что в общем это одно и то же, но на разных уровнях иерархии данных и операций. При написании программы осуществляется декомпозиция реализуемых процедур до уровня операций машинного языка или используемого языка программирования. При описании архитектуры компьютера осуществляется такая же декомпозиция операций, зафиксированных в его наборе команд до уровня элементарных операций, производимых над битами. Эта же аналогия имеет место в отношении данных. Структуры данных, используемых программой, строятся на основе единиц, имеющихся в языке команд (байтов, машинных слов), или на основе типов данных в языке программирования высокого уровня. Все виды данных в описываемой архитектуре, а также команды, адреса, указатели и другие кодируемые объекты конструируются из битов путем описания соответствующих структур.

Заметим, что имеет место параллелизм между структурами данных и структурами процедур, манипулирующих данными. Так, простейшей структуре данных — одномерной цепочке (конкатенации) битов прямо соответствует линейная последовательность выполняемых друг за другом операций. Схеме ветвления или выбора одной из предусмотренных в программе процедур в зависимости от результата проверки некоторого условия соответствует табличная организация данных. Аналогами циклов, т. е. замкнутых в кольцо последовательностей операций, являются регистры с циклическим сдвигом, а также массивы, в которых указатель, пробегающий n элементов, работает в режиме счетчика, считающего по модулю n. Этот параллелизм структур представляется еще одним подтверждением фундаментальности и практической важности рассмотренного выше принципа структурирования.

*) Dijkstra E. W. Notes on structured programming. T. H. Report 70-WSK-03, Eindhoven, Netherlands, Technological University, 1970. Русский перевод в кн.: Дал У., Дейкстра Э., Хоар К. Структурное программирование.— М.: Мир, 1975.

§ 3. Операции над битами

Простейшим элементом данных является *бит* — переменная, для которой допустимы только два значения, обозначаемые применительно к компьютерам цифрами 0 и 1. Физически бит реализуется в виде устройства с двумя отчетливо различными устойчивыми состояниями, которым сопоставляются значения 0 и 1. Типичным примером является транзисторный триггер, состояние которого опознается по напряжению на его выходе, скажем, в состоянии 0 это напряжение не превышает 0,3 В, а в состоянии 1 оно не менее чем 3 В. Установка триггера в требуемое состояние производится подачей на его управляющий вход импульса «принять» при наличии на воспринимающем входе напряжения, соответствующего принимаемому состоянию, причем состояние 0 устанавливается, если это напряжение не превышает 1 В, а состояние 1, — если оно не менее чем 2 В. Триггер может обладать также так называемым входом счетчика, при воздействии на который его состояние изменяется на противоположное, т. е. происходит переключение из состояния 0 в состояние 1, а из состояния 1 в состояние 0. На языке, отвлеченном от физической реализации, все это выражается так: 1) биту можно *присваивать* требующееся значение (0 или 1), 2) значение бита можно *инвертировать*.

Конкретный, т. е. выполняющий определенную функцию, бит наделяют собственным символическим именем. Например, бит, фиксирующий цифру переноса из старшего разряда сумматора (carry bit), обозначают обычно C, или Cr, или Cb. Используя принятый в языках программирования символ *присваивания* :=, можно записать:

Cb := 0 — присвоить биту Cb значение 0,
Cb := A — присвоить биту Cb текущее значение бита A.

Заметим, что в последнем случае значение бита A сохраняется неизменным. *Инвертирование* бита Cb задается в виде

Cb := ¬ Cb

где знак ¬ символизирует операцию инвертирования или отрицания НЕ: поскольку возможны только два значения, то НЕ 0 есть 1, а НЕ 1 есть 0.

Операция инвертирования является одной из четырех возможных над двузначной переменной одноместных операций. Три других операции — это: 1) заменить текущее значение переменной нулем (т. е. присвоить 0), 2) заменить единицей, 3) оставить неизменным (повторить).

Наиболее часто используемыми двуместными операциями над битами являются *конъюнкция* и *дизъюнкция*. Всего же возможны 16 различных двуместных операций над двузначными переменными, но любую из них можно реализовать в виде выражения с использованием только инверсии и конъюнкции или инверсии и дизъюнкции. Однако более понятными и легко преобразуемыми являются выражения, построенные на базе трех операций — инверсии, конъюнкции и дизъюнкции, которые вместе с двузначными переменными составляют *булеву алгебру*.

Операция конъюнкции, называемая также операцией И (по-английски AND), отражает действие совмещения, передаваемое в естественном языке при помощи союза И, когда им связываются необходимые признаки одного и того же предмета, атрибуты понятия. Например, делимость на 6 — это четность и делимость на 3. Можно вычислять делимость на 6 по известным делимости на 2 (четности) и делимости на 3. Для этого введем биты D6, D3, D2, условившись, что значение 1 символизирует наличие сопоставленной биту делимости. Желаемое вычисление задается предложением

$$D6 := D2 \wedge D3$$

в котором $\wedge$ — символ конъюнкции. Значение 1 будет присвоено переменной D6 только в том случае, когда оба члена конъюнкции — D2 и D3 — совместно обладают значением 1. Если же значением хотя бы одного из этих членов является 0, то присваиваемое значение их конъюнкции также будет 0.

Операция дизъюнкции *двойственна* конъюнкции в том смысле, что 0 является ее результатом только в случае, когда оба члена имеют значение 0, а конъюнкция дает в результате 1 только тогда, когда значения обоих членов равны 1. Рассмотренное только что вычисление делимости на 6 можно осуществить с помощью дизъюнкции, так сказать, двойственным способом, условившись, что наличию делимости соответствует значение 0, а не 1. При этом в предложении для вычисления $\overline{D6}$ знак конъюнкции надо заменить двойственным ему знаком дизъюнкции $\vee$:

$$\overline{D6} := \overline{D2} \vee \overline{D3}$$

Присваивание переменной $\overline{D6}$ значения 0 производится только в случае, когда значения обоих членов дизъюнкции 0. Достаточно хотя бы одному из членов иметь значение 1, как результат дизъюнкции будет 1.

Можно сказать, что дизъюнкция предпочитает значение 1 (а конъюнкция соответственно — 0): если значения членов дизъюнкции различны, то в качестве результата принимается 1 (а в случае конъюнкции — 0). По отношению к предпочитаемому значению 1 дизъюнкция идентична союзу ИЛИ естественного языка в том неисключающем значении, которое этот союз имеет при перечислении достаточных признаков предмета или достаточных условий события. Например, если источником денег, получаемых студентом, служит стипендия и переводы, присылаемые родителями, то введя соответственно биты Д, С, П и условившись обозначать всякое поступление денег значением 1, имеем предложение, устанавливающее факт получения денег студентом, в виде

$$Д := С \vee П$$

Студент при деньгах, если выдали стипендию или получен перевод, а тем более, если были и стипендия, и перевод, хотя в качестве факта получения достаточно одного из двух. Операцию дизъюнкции обозначают также словом ИЛИ и эквивалентным английским словом OR.

Довольно часто применяется операция *«исключающее* ИЛИ», или «ИЛИ с исключением» (Exclusive OR), которую логики называют *неэквивалентностью*, а проектировщики арифметических устройств — *сложением по модулю два*. В отличие от обычного (неисключающего) ИЛИ, т. е. от дизъюнкции, эта операция в случае, когда значением обоих операндов является 1, дает в результате не 1, а 0. Другими словами, она дает в результате 1 только тогда, когда значения операндов различны (неэквивалентны).

Сложение по модулю два означает, что в качестве результата принимается остаток от деления нацело суммы на 2. Применительно к значениям бита 0 и 1 это выражается в том, что

$$0 \oplus 0 = 1 \oplus 1 = 0,$$
$$0 \oplus 1 = 1 \oplus 0 = 1.$$

В сущности это естественное сложение битов: когда значение суммы $1 + 1$ выходит за пределы интервала допустимых для бита значений, его заменяют сравнимым по модулю 2, т. е. нулем. В качестве знака такого сложения мы используем $\oplus$ (плюс в кружке). Эту операцию обозначают также символами $\neq$, $\forall$, XOR.

Операция, двойственная неэквивалентности, — *эквивалентность* — дает в результате 1 в случае равенства значений ее операндов, т. е. оба значения 0 или оба 1. Эту операцию обозначают знаком $=$ или $\sim$. Ее можно рассматривать как частный случай операции, проверяющей отношение равенства чисел, которая дает в ре-

зультате 1, если сравниваемые числа равны, а если не равны, то дает 0. Соответственно, неэквивалентность является частным случаем операции, устанавливающей неравенство чисел. Впрочем, обе операции фиксируют как равенство, так и неравенство, но одна при неравенстве дает 0, при равенстве — 1, а другая — наоборот, дает 1 при равенстве и 0 при неравенстве. В этом, собственно, и проявляется их двойственность.

Операции эквивалентность и неэквивалентность выразимы посредством операций булевой алгебры — конъюнкции, дизъюнкции и инверсии. Так, неэквивалентность, или сложение битов A и B, представима каждым из выражений

$$(A \vee B) \wedge \neg (A \wedge B), \quad (A \vee B) \wedge (\neg A \vee \neg B),$$
$$(A \wedge \neg B) \vee (\neg A \wedge B).$$

Аналогичными выражениями эквивалентности являются

$$(A \wedge B) \vee \neg (A \vee B), \quad (A \wedge B) \vee (\neg A \wedge \neg B),$$
$$(A \vee \neg B) \wedge (\neg A \vee B).$$

С другой стороны, операции булевой алгебры можно представить полиномами битов в так называемой модулярной алгебре, или алгебре Жегалкина. В этой алгебре операцией сложения является рассмотренная выше операция сложения битов, а операция умножения совпадает с конъюнкцией. Действительно, конъюнкция с арифметической точки зрения представляет собой умножение для чисел 0 и 1 (умножение битов):

$$0 \wedge 0 = 0 \wedge 1 = 1 \wedge 0 = 0,$$
$$1 \wedge 1 = 1.$$

Таким образом, имеем алгебру с операциями сложения и умножения битов, представляющую собой частный случай обычной алгебры чисел, в котором множество чисел ограничено парой 0 и 1. Условившись, что операндами могут быть только биты, можно употреблять общепринятые знаки сложения и умножения, причем знак умножения, как обычно, опускать. В такой алгебре конъюнкция A и B будет просто AB, дизъюнкция выразится полиномом $AB + A + B$, а инверсия $\neg A$ — в виде суммы с константой: $A + 1$. Ясно, что и всякая булева функция может быть представлена полиномом битов, т. е. посредством операций сложения и умножения, а также константы 1.

То, что одна и та же операция над битами имеет разные истолкования (конъюнкция и умножение, которое, впрочем, не умножает; неэквивалентность и сложение, которое не суммирует; инверсия и сложение с 1, также не увеличивающее количества) объясняется

крайней элементарностью бита. В самом деле, два значения — это абсолютный минимум, при котором переменная сохраняет еще способность изменяться. Одно значение — это уже только константа. С увеличением значности переменных число существенно различных операций быстро растет и совпадающие в случае битов представители равных алгебр обретают индивидуальные черты, становятся особенными, отличными друг от друга. Так, даже минимальное усложнение элемента данных, заключающееся в добавлении третьего значения (*трит* вместо бита), приводит к тому, что умножение уже не совпадает с конъюнкцией, сложение не является неэквивалентностью и не совпадает с вычитанием в отличие от того, как это имеет место для битов.

§ 4. Слова и массивы

Операции над битами имеют очевидное значение как средство реализации логики, которая занимает ведущее положение в управляющих и многих других применениях микрокомпьютеров. Но кроме того, эти операции являются той основой, на которой возводится иерархия всех последующих операций и процедур, как осуществляемых аппаратурой, так и создаваемых в виде программ. Ее построение производится параллельно и в неразрывной связи с конструированием все более и более мощных и сложных структур данных.

Самой простой формой организации отдельных битов в упорядоченную совокупность является линейная *цепочка* или *ряд* (часто употребляется также термин строка, представляющий собой неудачный перевод английского string). Операция соединения элементов в цепочку называется *сцеплением*, *сочленением* или *конкатенацией*. Строго говоря, сцепление определено не над элементами, а над цепочками, причем цепочка может состоять из одного элемента и даже не содержать ни одного элемента — быть пустой.

Цепочка, так же как бит, обладает информационной емкостью, которая равна емкости одного элемента, умноженной на число элементов, содержащихся в данной цепочке. Число элементов в цепочке называется длиной цепочки. Длина цепочки, вообще говоря, может изменяться. Цепочка фиксированной длины называется *словом*. Цепочка битов фиксированной длины называется *двоичным словом*.

Значением цепочки является совокупность значений ее элементов, просматриваемых в порядке расположения этих элементов друг за другом. Число различных значений, принимаемых цепочкой, равно p^n, где p — *значность* элементов, составляющих цепочку, n — *длина* цепочки. Двоичное слово длины n принимает 2^n различных значений. Действительно, один бит принимает 2 различных значе-

ния, а с каждым добавлением еще одного бита число значений удваивается.

Память микрокомпьютера, как главную, так и в большей части собственную память процессора, организуют в форме массивов слов, которые в свою очередь представляются массивами битов. *Массив* — это такая организация совокупности элементов, при которой доступ к любому элементу обеспечивается заданием соответствующего этому элементу значения указателя (индекса), определяющего положение элемента в массиве. Примером одномерного массива служит вектор, двумерного — матрица, но возможны массивы и большей размерности.

Одномерный массив представляет собой цепочку элементов, которым сопоставлены последовательные целые числа — значения указателя. Допустима как восходящая (при продвижении слева направо или снизу вверх), так и нисходящая нумерация элементов. В последнем случае значение индекса увеличивается при продвижении справа налево или сверху вниз. Описание массива наряду с наименованием типа элементов и именем массива содержит значения индекса, соответствующие крайним элементам в виде так называемой граничной индексной пары (для n-мерного массива в виде n пар). Например, одномерные массивы битов могут быть описаны так:

bit array BYTE (1:8), R (15:0), FXP (0:—31), Zb (1:1)

Массив BYTE содержит 8 битов, пронумерованных слева направо, начиная с 1. Массив R 16-битный, значение индекса возрастает справа налево, начиная с 0. В массиве FXP 32 бита с убывающим, начиная с нуля, индексом. Массив Zb состоит из одного бита, индекс которого равен 1.

Ссылкой на массив в целом служит его имя, употребленное без индекса. Ссылка на некоторый элемент массива состоит из имени этого массива, за которым следует индекс в виде заключенного в скобки номера требуемого элемента. Ссылка на ряд элементов (*поле*) — это имя массива в сопровождении заключенной в скобки индексной пары, которая задает номера начального и концевого элементов адресуемого поля. Например, присваивание массиву BYTE значения, представленного младшими 8-ю битами массива R, записывается в виде

BYTE := R(7:0);

Присваивание однобитному массиву Zb значения нулевого бита массива FXP выражается так:

Zb := FXP(0);

Присваивание значения массива BYTE старшим 8-ми битам массива R, а младшим 8-ми битам — значения 0 осуществляется парой предложений:

R := 0; R(15:8) := BYTE;

Для описания главной памяти микрокомпьютера в виде массива битов необходим массив размерности два или более. Например, побайтно адресуемую память емкостью 16 КВ можно представить массивом m, описанным так:

bit array m(0:16383, 0:7);

Это исчерпывающее с точностью до бита описание полностью отражает структуру памяти, но с точки зрения практического использования компьютера представляется излишне подробным и даже в известной степени неверным, поскольку прямого доступа к отдельным битам запоминающее устройство не обеспечивает — адресовать можно только целые байты. Поэтому для пользователей память компьютера правильнее будет представить массивом прямо адресуемых элементов, охарактеризовав, если надо, эти элементы отдельным описанием. Рассматриваемую память можно задать так:

byte array m(0:16383);

Однако применительно к детальному описанию архитектуры компьютера представление структур данных с точностью до бита вполне оправдано и в тех случаях, когда прямой доступ к битам физически не обеспечен. Например, чтобы исчерпывающе описать операции, выполняемые компьютером над словами, приходится рассматривать слово как массив битов, манипулируя произвольными его полями и отдельными битами. Как правило, набор команд компьютера, и даже микрокомпьютера, не обеспечивает возможности столь свободного обращения с битами — командам соответствуют сравнительно сложные операции над словами, а к тому, как они выполняются на уровне битов, доступа нет.

Таким образом, следует иметь в виду, что вводимые в процессе описания архитектуры компьютера детальные структуры данных и операции над ними не всегда доступны (и скорее недоступны) пользователю компьютера, а являются лишь средством описания того, что доступно. Например, описывая операции, определенные над регистром-аккумулятором, приходится представить этот регистр как массив битов, скажем Ac(15:0), и дать алгоритм выполнения каждой операции в терминах операций над битами. Программист же воспринимает регистр Ac как единое слово, а относящиеся к этому регистру операции — как операции над словами.

Заметим, что ссылка на массив или на любую другую совокупность данных, взятую в целом, по имени вполне аналогична обращению по имени к процедуре. В обоих случаях имя представляет то, что реализовано в деталях на нижних уровнях иерархии.

§ 5. Числовая интерпретация слов

Двоичное слово длины n принимает 2^n различных значений, которые можно сопоставить объектам того или иного рода, т. е. интерпретировать (истолковывать) в том или ином смысле. Например, значения 8-битных слов (байтов) интерпретируются как коды литер (букв, цифр и других печатных знаков), а с другой стороны, как коды команд в 8-битных микрокомпьютерах, 16-битные слова используются как коды команд и как адреса ячеек памяти и т. д. При этом одно и то же значение выступает в различных смыслах и в каждом конкретном случае его смысл устанавливается в зависимости от выполняемой операции. Иначе говоря, интерпретация слов в компьютере осуществляется выполнением над ними соответствующих операций. Так, если над парой слов выполняется операция сложения, то тем самым эти слова интерпретируются как числа, а если к ним применена команда «Напечатать», то они будут истолкованы как коды печатных знаков.

Важнейшей является интерпретация слова как целого числа, лежащая в основе всех числовых и ряда других интерпретаций, а в микрокомпьютерах также определяющая упорядоченность значений слова и, пожалуй, преобладающую часть возможностей их преобразования, поскольку, как правило, имеется только целочисленная арифметика. Коды операций, алфавитных знаков и прочих нечисловых объектов рассматриваются обычно в виде их числовых эквивалентов.

Простейшая числовая интерпретация двоичного слова — это представление целого без знака (*натуральный код*). Биты слова рассматриваются как разряды двоичного целого числа, т. е. наделяются весами, которые возрастают справа налево по степеням двойки: $2^{n-1}, \ldots, 2^3, 2^2, 2^1, 2^0$, и значение слова A полагается равным

$$A = A_{n-1} \cdot 2^{n-1} + A_{n-2} \cdot 2^{n-2} + \ldots + A_i \cdot 2^i + \ldots + A_2 \cdot 2^2 + A_1 \cdot 2 + A_0,$$

где A_i — значение i-го бита, т. е. 0 или 1. Слово A принимает минимальное значение, равное 0, когда значения всех его битов равны 0. Максимальное значение $A = 2^n - 1$ соответствует случаю, в котором значения всех битов равны 1.

Арифметические операции над представленными таким образом числами выполняются по правилам, обычным для позиционных систем счисления с неотрицательными значениями цифр: вычисление производится поразрядно, причем если полученный в данном разряде результат не является значением одной из цифр, то происходит перенос единицы в соседний старший разряд или заем из этого разряда. Так, выполняемая двоичным счетчиком операция прибавления единицы к имеющемуся на счетчике числу A выражается в последовательной обработке битов слова A, начиная с A_0. Обработка каждого бита A_i заключается в том, что определяется значение переноса C_{i+1} из i-го разряда в $(i+1)$-й разряд:

$$C_{i+1} := A_i \wedge C_i;$$

и модифицируется значение i-го бита сложением по модулю два с переносом C_i, поступившим из предыдущего бита:

$$A_i := A_i \oplus C_i;$$

Переносом в нулевой разряд счетчика служит единица, прибавляемая к числу A, т. е. $C_0 = 1$.

Пример. Пусть $n = 4$ и значение счетчика A равно нулю, т. е. 0000. Первое прибавление единицы изменит только значение бита A_0:

$$A_0 := 0 \oplus 1; \quad C_1 := 0 \wedge 1; \quad A_1 := 0 \oplus 0;$$
$$C_2 := 0 \wedge 0; \quad \ldots \quad A_3 := 0 \oplus 0; \quad C_4 := 0 \wedge 0;$$

Второе прибавление изменит A_0 и вызовет перенос единицы в A_1:

$$A_0 := 1 \oplus 1; \quad C_1 := 1 \wedge 1; \quad A_1 := 0 \oplus 1;$$
$$C_2 := 0 \wedge 1; \quad \ldots A_3 := 0 \oplus 0; \quad C_4 := 0 \wedge 0;$$

Третье прибавление единицы изменит только значение A_0. Счетчик последовательно принимает значения 0000, 0001, 0010, 0011 и т. д.

В том случае, когда значение числа A оказывается максимальным для данной длины слова n, т. е. когда значения всех битов слова равны 1, прибавление единицы вызывает сквозной перенос по всей длине и возникновение переноса $C_n := 1$ из $(n-1)$-го в несуществующий n-й бит. Эта ситуация называется *переполнением* счетчика. С сигналом $C_n := 1$ можно связать выполнение служебной процедуры, фиксирующей факт переполнения или осуществляющей, например, прибавление единицы к слову, составляющему дополнительные старшие разряды счетчика. Но можно и попросту игнорировать этот сигнал. В последнем случае счет будет производиться по модулю 2^n: вместо значения 2^n, следующего по порядку за максимальным, представимым n битами, значением $2^n - 1$, счетчик будет принимать сравнимое с 2^n по указанному модулю и представимое значение 0, за которым последуют затем 1, 2, 3, . . ., $2^n - 2$, $2^n - 1$, 0, 1, 2 и т. д.

В условиях обычной для микрокомпьютеров целочисленной арифметики с ограниченной длиной слов счет по модулю 2^n явле-

ние естественное, причем не только для счетчиков, но и для других арифметических механизмов. Вследствие фиксированной длины слова, всякий результат, если не приняты особые меры, оказывается представленным только его младшими n битами, а значения вышедших за пределы слова старших разрядов утрачиваются. Таким образом, результаты арифметических операций достоверны лишь при условии непереполнения слов, используемых для представления чисел.

Операция сложения двух чисел, представленных двоичными словами A и B длины n, заключается, подобно операции счета, в последовательной справа налево обработке битов слагаемых с учетом возникающих переносов. Перенос C_{i+1} в $(i+1)$-й разряд и значение i-го бита суммы S_i определяются как результаты присваиваний

$$C_{i+1} := A_i \wedge B_i \vee A_i \wedge C_i \vee B_i \wedge C_i;$$
$$S_i := A_i \oplus B_i \oplus C_i;$$

Перенос в нулевой разряд, если не предписано иное, полагается равным нулю: $C_0 = 0$. Возникновение переноса $C_n := 1$ из $(n-1)$-го разряда означает переполнение сумматора.

Пример. Пусть $n = 8$, A = 00101011, B = 00011010. Процесс сложения можно представить следующим столбцом, в котором C — цепочка переносов, S — полученная сумма:

```
A   00101011
B   00011010
C  001110100
-------------
S   01000101
```

Натуральный двоичный код — это представление целых чисел без знака. Он недостаточен даже для целочисленной арифметики микрокомпьютеров, которая включает операцию вычитания и отрицательные числа, т. е. определена над числами со знаком. В связи с этим в микрокомпьютерах используют по меньшей мере два представления чисел: натуральный код для адресов памяти, которые по традиции принимают целые неотрицательные значения, и так называемый дополнительный код для числовых данных.

Дополнительный код заключается в том, что из 2^n значений двоичного слова длины n только первые $2^n/2$ числовых значений интерпретируются как неотрицательные, в то время как остальные $2^n/2$ значения представляют собой отрицательные числа, сравнимые по модулю 2^n с теми положительными числами, которые представлены этими значениями слова в натуральном коде. Таким образом, если слово A в натуральном коде является числом, удовлетворяющим отношению $A < 2^{n-1}$, то оно в дополнительном коде сохраняет свое натуральное числовое значение, в противном случае его значение в дополнительном коде получается вычитанием из натурального значения величины 2^n. Другими словами, интервал числовых значений слова, простирающийся в случае натурального кода от 0 до 2^n-1, в дополнительном коде складывается из двух половинных интервалов: неотрицательные числа от 0 до $2^{n-1}-1$, а далее отрицательные числа от -2^{n-1} до -1, отстоящие от своих прототипов из второй половины интервала чисел, представленных в натуральном коде, точно на 2^n. Значениями слова покрыта половина расположенного вправо от нуля интервала, покрываемого в натуральном коде, а затем такой же участок, примыкающий к нулю слева.

Код называется дополнительным, потому что отрицательные числа в нем представлены значениями слова, которые в натуральном коде обозначают числа, дополняющие абсолютные величины соответствующих отрицательных чисел до 2^n. Иначе: абсолютная величина заданного в дополнительном коде отрицательного числа получается вычитанием натурального числового значения данного слова из 2^n. Например, четырехбитное слово 1001 в натуральном коде значит 9, а в дополнительном: $9-2^4 = -7$. Ясно, что абсолютная величина этого значения может быть получена вычитанием из 2^4 значения слова в натуральном коде: $2^4-9 = 7$.

К понятию дополнительного кода, как видно, весьма неудобному для человеческого восприятия, можно подойти естественным путем, выполняя в натуральном коде вычитание из меньшего числа большего. Этот подход не избавляет от неудобств, но может служить определенным утешением, так как показывает, что неудобства не придуманы людьми, а являются неотъемлемым свойством двоичного представления чисел. Так, при вычитании единицы из нуля, который представлен словом A, все n битов которого равны нулю, обрабатывая крайний справа бит A_0, мы вынуждены сделать заем из следующего по старшинству бита A_1, поскольку попытка вычитать без заема приводит к не представимому в рамках бита отрицательному результату. Значение бита A_1 также равно 0, а заем равносилен вычитанию 1, поэтому возникает необходимость заема из A_2, и далее этот процесс распространяется по всей длине слова A и даже за его пределы — возникает заем из несуществующего n-го бита. В записи столбцом это выглядит так:

```
  00 . . . 000
—
            1
-------------
(. . . 11)11 . . . 111
```

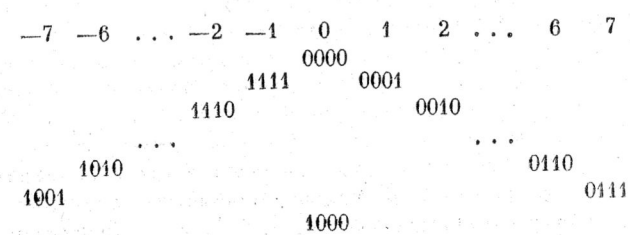
Единицы и многоточие в скобках являются обозначением того, что получаемая в результате вычитания последовательность единиц

продолжается влево бесконечно. Это и есть натуральное представление минус единицы в двоичной системе счисления. Последовательно модифицируя его вычитанием единицы, получим представления других отрицательных чисел: ...111...110 = —2, ...111...101 = = —3 и т. д.

Ясно, что в отличие от неотрицательных чисел, в представлении которых подразумевается не ограниченная влево последовательность ведущих нулей, отрицательным числам свойственна аналогичная последовательность ведущих единиц. Пока длины этих последовательностей не ограничены, количества представимых значений чисел, как отрицательных, так и неотрицательных, бесконечны. Когда же надо ограничиться словом конечной длины, количество ведущих нулей/единиц сводится к минимуму, но должен быть по крайней мере один ведущий нуль (ведущая единица), чтобы можно было различать отрицательные и неотрицательные числа. Таким образом, в слове должен сохраняться как минимум один ведущий бит, имеющий значение 0, если представленное словом число неотрицательно, и значение 1, если оно отрицательно.

Этот начальный бит слова принято называть *битом знака* чисел, что расходится с пониманием знака числа в математике и технических науках как трехзначной функции, принимающей значения «плюс», «минус» и «нуль» или «плюс», «минус» и «нет знака». Надо бы подчеркнуть, что в зависимости от значения так называемого бита знака осуществляется разбиение чисел на отрицательные и неотрицательные, но обычно считают, что значению бита знака 0 соответствует знак «плюс», а значению 1 — знак «минус». Таким образом, числа в зависимости от бита знака разбивают на положительные и отрицательные, причем число нуль относится к положительным и сам термин «положительное число» становится неоднозначным: то ли имеется в виду строго положительное число, то ли неотрицательное (положительное или равное нулю).

Рассмотрим структуру множества значений, принимаемых двоичным словом при числовой со знаком интерпретации, на конкретном примере четырехбитного слова.



За исключением 0000 и 1000, рассматриваемые значения группируются в пары относительно операции изменения знака (каждая такая пара расположена на отдельной строке). Члены пары равны по абсолютной величине, но противоположны по знаку. Операция изменения знака числа A, представленного в дополнительном коде, может быть реализована как побитная инверсия слова A, т. е. инверсия значений всех его битов с 0 на 1 и с 1 на 0, с последующим прибавлением к инвертированному значению единицы:

$$A := \overline{A} + 1;$$

Например, слово 1011, представляющее число —5, в результате инвертирования превращается в 0100, а после прибавления 1 — в 0101, что представляет собой число 5. Обратно, инвертируя 0101, получим 1010, а затем добавив 1, имеем 1011.

Доказать правильность данной формулы в общем случае можно, исходя из того, что побитная инверсия слова равносильна дополнению его значения как числа без знака до $2^n - 1$. (Это следует в свою очередь из того, что сумма весов всех битов слова — числа без знака — равна $2^n - 1$.) Прибавление 1 к результату инвертирования дает дополнение до 2^n, которое, как было разъяснено выше, является представлением в дополнительном коде числа, равного исходному числу по абсолютной величине, но противоположного по знаку.

Другой способ реализации операции изменения знака числа сводится к последовательной обработке битов слова A, начиная с A_0, заключающейся в следующем. Значения обрабатываемых битов не изменяются, пока не встретится бит, равный 1 (т. е. не изменяются хвостовые нули). Не изменяется и встретившаяся первой 1, но вся последующая часть слова инвертируется. Например, в слове 0110, представляющем 6, сохраняется хвостовой нуль и следующая за ним первая 1, а остальные биты инвертируются. В результате имеем 1010, т. е. —6. Обратно, обрабатывая тем же способом слово 1010, получим 0110.

Обосновать этот способ можно, рассматривая его как видоизменение предыдущего способа. Действительно, в случае инвертирования и прибавления 1 хвостовые нули превращаются в последовательность единиц, которые после прибавления 1 снова становятся нулями вследствие распространения по ним переноса 1. Этот перенос прервется на первом же бите, исходным значением которого была 1, перешедшая в результате инвертирования в 0, причем поступающая в этот бит единица переноса восстанавливает в нем значение 1. Далее перенос не распространяется, поэтому все последующие биты сохраняют значения, установленные при инвертировании.

Иногда дополнительный код истолковывают так, будто биту знака приписан отрицательный вес -2^{n-1}, т. е. принимают, что

числовое значение слова определяется формулой

$$A = -A_{n-1}\cdot 2^{n-1}+A_{n-2}\cdot 2^{n-2}+\ldots+A_i\cdot 2^i+\ldots+A_2\cdot 2^2+A_1\cdot 2+A_0.$$

При таком истолковании операцию изменения знака числа можно представить как вычитание в натуральном коде из слова, содержащего бит знака с нулями в качестве остальных битов, слова, имеющего нуль в бите знака при сохранении значений прочих битов:

$$A := A_{n-1}00\ldots00-0A_{n-2}\ldots A_2A_1A_0.$$

Наконец, самым очевидным способом было бы вычитание данного слова как числа в натуральном коде из 2^n или из равного 2^n по модулю 2^n нуля: изменение знака по существу и есть вычитание из нуля. Однако сама операция вычитания реализуется в дополнительном коде как прибавление вычитаемого с предварительно измененным знаком. Поэтому в компьютере изменение знака числа осуществляется инвертированием с прибавлением 1. При этом нет необходимости создавать «вычитатель», а сложение с учетом знаков чисел осуществляет рассмотренный применительно к натуральному коду двоичный сумматор, если обрабатываемые им слова интерпретировать как числа, представленные в дополнительном коде.

Таким образом, один и тот же сумматор в зависимости от того, как интерпретируются обрабатываемые им двоичные слова, выполняет сложение и чисел без знака, и чисел со знаком. Различие заключается в упорядоченности значений слова и в признаках переполнения. В случае натурального кода (чисел без знака) значения возрастают от 000...00 до 111...11 и признаком переполнения является перенос 1 из (n — 1)-го в несуществующий n-й бит или заем 1 из n-го бита. В случае дополнительного кода (чисел со знаком) значения возрастают от 100...00 до 111...11 и далее от 000...00 до 011...11, причем возникновение единицы переноса/заема при переходе от 111...11 к 000...00 и обратно не означает переполнения.

Переполнением слова, интерпретируемого как число со знаком, является получение в результате операции положительного числа, превышающего 011...11, или отрицательного числа, оказывающегося за нижней границей (100...00) интервала представимых словом отрицательных чисел. Понятно, что переполнение может произойти при получении суммы чисел одинакового знака, причем в качестве вышедшего за пределы представимых словом значений положительного результата получается значение, обозначающее отрицательное число, а в качестве отрицательного — положительное. Иначе: признаком переполнения может служить несоответствие значения бита знака суммы значению битов знака слагаемых.

Однако более удобным для машинной реализации является другой способ обнаружения переполнения в дополнительном коде,

а именно — по несовпадению значений битов знака и переноса в полученном результате. Перед выполнением операции к каждому слову-операнду добавляется слева n-й бит (*«бит переноса»*), значение которого полагают равным значению бита знака. (Иногда это называют модифицированным дополнительным кодом.) Теперь в представимом результате значения битов знака и переноса будут одинаковыми, а в случае переполнения — разными. Проиллюстрируем сказанное на примерах с четырехбитными словами, к которым слева добавлен пятый бит, дублирующий значение четвертого.

Примеры, в которых переполнение не происходит:

11101	00101	00011	11011
+ 00110	+ 00010	+ 11010	+ 11110
00011	00111	11101	11001

Примеры, в которых переполнение происходит:

00101	00110	11001	11101
+ 00011	— 11101	+ 11011	— 00110
01000	01001	10100	10111

Возвращаясь к множеству числовых значений слова в дополнительном коде, обратим внимание на то, что имеется два значения, не изменяемые операцией изменения знака: соответствующее нулю значение 000...00, у которого нет знака, и значение 100...00, соответствующее отрицательному числу -2^{n-1}, у которого нет представимой словом длины n положительной пары. Действительно, применяя к этому значению операцию изменения знака в любом из рассмотренных вариантов ее реализации, получаем неизменно все то же значение. Например:

$$000\ldots00-100\ldots00=100\ldots00$$

В конкретном случае четырехбитного слова это равносильно

$$0-(-8) = -8.$$

Такая «арифметика» может привести к самым нежелательным и опасным последствиям. Дефект этот составляет неотъемлемую принадлежность дополнительного кода в системах счисления с четным основанием, и не видно эффективного способа его устранения. В аккуратно построенных архитектурах предусматривают сигнализацию, срабатывающую при появлении значения 100...00 в качестве результата арифметической операции. Поскольку это значение находится на границе интервала представимых отрицательных чисел и парное ему положительное число непредставимо (вызывает пере-

полнение), то естественно было бы принять, что оно представляет не отрицательное число, а еще один случай переполнения.

С числовой интерпретацией слова связана также операция *арифметического сдвига*. При сдвиге влево каждый бит слова, начиная с крайнего левого, принимает значение соседнего справа бита, а последний бит принимает значение 0. Если при этом не происходит переполнение (не изменяется значение бита знака), то числовое значение слова увеличивается в 2 раза, т. е. сдвиг влево равносилен умножению числа на 2. Например:

до сдвига	0011 (3)	1101 (—3)	1010 (—5)
после сдвига	0110 (6)	1010 (—6)	0100 (переполнение)

При сдвиге вправо каждый бит, начиная с крайнего правого, принимает значение соседнего слева бита, а значение бита знака сохраняется неизменным. Числовое значение слова в результате такого сдвига делится на 2 и полученное частное заменяется ближайшим целым, не превосходящим его точного значения. Примеры:

до сдвига	0001 (1)	1111 (—1)	0011 (3)	1101 (—3)
после сдвига	0000 (0)	1111 (—1)	0001 (1)	1110 (—2)

Формально арифметический сдвиг вправо в дополнительном коде равносилен делению нацело на 2 при указанном только что понимании целой части числа как ближайшего не превосходящего это число целого значения. С точки зрения теоретиков, эта алгольная трактовка вполне удовлетворительна, но практики считают, что арифметический сдвиг вправо в дополнительном коде не является делением нацело, потому что значение —1 в результате сдвига не изменяется, а результаты сдвига равных по абсолютной величине, но различных по знаку чисел оказываются не равными друг другу по абсолютной величине. Другими словами, практикам удобнее понимать деление нацело в духе фортрана, чтобы деление и сдвиг чисел, равных по абсолютной величине, давали равные по абсолютной величине результаты.

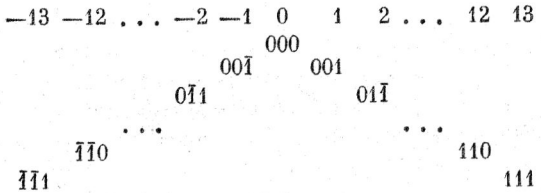
Несмотря на отмеченные недостатки и неудобства, к которым можно присовокупить проблемы округления, варьируемой длины слова и др., дополнительный код является, по-видимому, наиболее удовлетворительным представлением чисел со знаком в двоичной машине: по крайней мере, другие известные альтернативы (прямой и обратный коды, система с основанием —2) в микрокомпьютерах практически не привились. Вместе с тем существует альтернатива, полностью свободная от недостатков, проблем и неудобств. Она состоит в том, что надо продвинуться на один шаг вперед — перейти от системы с двумя цифрами к системе с тремя цифрами, от двоичного

представления чисел к троичному. В троичной системе счисления имеется возможность натурального кода для чисел со знаком.

Эта возможность реализуется выбором в качестве числовых значений трита (трехзначного аналога бита) чисел 0, 1 и —1, т. е. добавлением к двоичным цифрам 0 и 1 «отрицательной цифры» —1. Числовое значение троичного слова длины n (состоящего из n тритов) выражается при этом применительно к целым числам в виде, совершенно аналогичном выражению числа в натуральном двоичном коде, а именно:

$$A = A_{n-1}\cdot 3^{n-1} + A_{n-2}\cdot 3^{n-2} + \ldots + A_i\cdot 3^i + \ldots + A_2\cdot 3^2 + A_1\cdot 3 + A_0,$$

где A_i — значение i-го трита, т. е. 0, или 1, или —1. В данной интерпретации из 3^n значений, принимаемых троичным словом длины n, имеем $(3^n - 1)/2$ положительных, ровно столько же отрицательных и одно равное нулю. Знак числа A задается старшим из не равных нулю тритов: число положительно, если этот трит равен 1, отрицательно, если он равен —1, равно нулю, если все триты слова равны нулю. Так, при n = 3 множество представимых чисел выглядит следующим образом:



Цифра $\bar{1}$ (единица с чертой сверху) означает —1.

Нетрудно заметить, что данная система чисел со знаком несравнимо естественнее и проще дополнительного кода, а досадные нерегулярности последнего в ней полностью исключены. Число значений слова нечетно, поэтому имеется один нуль и $(3^n-1)/2$ пар чисел, противоположных по знаку и равных по абсолютной величине. Операция изменения знака числа состоит в том, что в тритах, не равных нулю, изменяется знак цифры, т. е. 1 заменяется на $\bar{1}$, а $\bar{1}$ — на 1.

Таким образом, значения троичного слова длины n интерпретируются как наименьшие по абсолютной величине вычеты по модулю 3^n, как ближайшие к нулю и симметрично расположенные вокруг нуля целые числа, покрывающие интервал от $-(3^n-1)/2$ до $(3^n - 1)/2$. Разумеется, сумматор в такой системе осуществляет сложение с учетом знаков слагаемых, а счетчик является реверсивным — обеспечивающим счет как в положительном, так и в отрицательном направлении, в зависимости от знака поступающих на его

вход единиц: 1 или $\bar{1}$. Переносы принимают, естественно, три значения: 0, 1, $\bar{1}$ (последнее можно рассматривать как заем 1). Признаком переполнения слова является ненулевой перенос в несуществующий n-й трит. Существенно, что «расширение» (увеличение длины) слова производится добавлением нулевых тритов в случае как положительного, так и отрицательного знака. Это коренным образом упрощает реализацию операций с операндами неодинаковой длины и изменение длины операндов.

Соответственно упрощаются и оптимизируются операции сдвига и деления нацело. Во-первых, нет необходимости в специальной операции арифметического сдвига, поскольку обычный «логический» сдвиг с заполнением освобождающихся тритов нулями равносилен умножению/делению числа со знаком на 3. Во-вторых, результатом реализуемого сдвигом вправо деления нацело является целое, ближайшее к точному значению частного по абсолютной величине.

Например:

до сдвига	1$\bar{1}\bar{1}$ (5)	$\bar{1}$11 (—5)	101 (10)	10$\bar{1}$ (—10)
после сдвига	01$\bar{1}$ (2)	0$\bar{1}$1 (—2)	010 (3)	0$\bar{1}$0 (—3)

Это означает также, что отсечение хвостовых тритов слова представляет собой правильное округление троичного числа.

Таким образом, троичный код с цифрами 0, 1, $\bar{1}$ является поистине безукоризненным и, вместе с тем, столь же естественным и простым, как натуральный двоичный код неотрицательных чисел, представлением чисел со знаком.

§ 6. Нечисловая интерпретация слов

Рассмотренные выше интерпретации слова как целого числа без знака и со знаком лежат в основе других числовых, а также и нечисловых интерпретаций слова. Многообразие интерпретаций порождается главным образом вследствие разбиения слова всевозможными способами на группы битов, называемые полями, значения которых истолковываются в том или ином смысле. Например, слово, представляющее десятичное число, разбивают на четверки битов и каждую четверку интерпретируют как целое без знака, обладающее значением десятичной цифры. Так, двоично-десятичный код числа 1234 будет 0001 0010 0011 0100. Аналогичным образом кодируются буквы, математические знаки и другие литеры алфавита, но ширина поля устанавливается большей — обычно равной 8-ми битам (байту).

Байт является основной единицей для двоичного кодирования литер не только в компьютерах, но и в системах передачи данных.

Соответствие двоичных значений байта литерам или управляющим сигналам, как правило, жестко фиксируется в аппаратуре, используемой для передачи и воспроизведения данных (в клавиатурах, в печатающих устройствах), поэтому крайне важно, чтобы оно было стандартным, ибо в противном случае аппаратура будет несовместимой: передаваемое с одного аппарата будет неверно воспроизводиться или не сможет быть воспроизведено другим.

Существуют стандарты кодов для обмена информацией, разработанные на основе международного стандарта, установленного международной организацией стандартов ISO — International Standard Organization. В СССР стандартные коды для вычислительных машин и аппаратуры передачи данных определены ГОСТ 13052-74, которым установлены два набора алфавитно-цифровых и управляющих символов 7-битного кода для обмена информацией — КОИ-7$Н_0$ и КОИ-7$Н_1$. Оба набора представлены в таблице, строки которой соответствуют всевозможным комбинациям значений четырех младших битов б4 — б1, а столбцы — трех старших б7 — б5. Восьмой бит байта используется для контроля по четности/нечетности: принимает значение 0 или 1 в зависимости от того, четна или нечетна сумма значений остальных семи битов.

Набор КОИ-7$Н_0$ включает строчные и прописные буквы латинского алфавита, набор КОИ-7$Н_1$ — русского. Кроме того, в оба набора входят цифры, математические и разделительные знаки, а также символы управления передачей данных, например: НЗ — начало заголовка, НТ — начало текста, КТ — конец текста, КП — конец передачи, ПС — перевод строки, ВК — возврат каретки и др. Управляющие символы ВХ и ВЫХ (вход и выход) осуществляют переключение соответственно в набор КОИ-7$Н_0$ и в набор КОИ-7$Н_1$. Пока не появился ни тот, ни другой из этих символов, по умолчанию действует набор КОИ-7$Н_0$.

Каждой кодовой комбинации в данном наборе соответствует печатный знак (литера) или управляющий символ и их «десятичный эквивалент» — числовое без знака значение кода. Например, комбинация 1011111, десятичный эквивалент которой равен 95, в наборе КОИ-7$Н_0$ означает подчеркивание, т. е. печатание черты под знаковой позицией, возможно, без перемещения каретки в следующую позицию, а в наборе КОИ-7$Н_1$ — строчную литеру ъ (твердый знак).

В числе наборов символов, предусмотренных ГОСТ 19767-74, имеется также включающий как латинские, так и русские прописные буквы, но не содержащий строчных букв. Этот набор, получаемый совмещением наборов КОИ-7$Н_0$ и КОИ-7$Н_1$ с предварительным изъятием из них строчных букв, весьма удобен в применениях, не вяза ных с высококачественной распечаткой текстов, и получил

Кодовая таблица КОИ-7$Н_0$/КОИ-7$Н_1$

					б7	0	0	0	0	1	1	1	1
					б6	0	0	1	1	0	0	1	1
					б5	0	1	0	1	0	1	0	1
б4	б3	б2	б1		0	1	2	3	4	5	6	7
0	0	0	0	0	ПУС	АР1	Пробел	0	@ / ю	P / п	` / Ю	p / П
0	0	0	1	1	НЗ	(СУ1)	!	1	A / а	Q / я	a / А	q / Я
0	0	1	0	2	НТ	(СУ2)	"	2	B / б	R / р	b / Б	r / Р
0	0	1	1	3	КТ	(СУ3)	#	3	C / ц	S / с	c / Ц	s / С
0	1	0	0	4	КП	СТП	¤	4	D / д	T / т	d / Д	t / Т
0	1	0	1	5	КТМ	НЕТ	%	5	E / е	U / у	e / Е	u / У
0	1	1	0	6	ДА	СИН	&	6	F / ф	V / ж	f / Ф	v / Ж
0	1	1	1	7	ЗВ	КБ	'	7	G / г	W / в	g / Г	w / В
1	0	0	0	8	ВШ	АН	(	8	H / х	X / ь	h / Х	x / Ь
1	0	0	1	9	ГТ	КН	)	0	I / ш	Y / ы	i / И	y / Ы
1	0	1	0	10	ПС	ЗМ	*	:	J / й	Z / з	j / Й	z / З
1	0	1	1	11	ВТ	АР2	+	;	K / к	[/ ш	k / К	{ / Ш
1	1	0	0	12	ПФ	РФ	,	<	L / л	\ / э	l / Л	| / Э
1	1	0	1	13	ВК	РГ	–	=	M / м	] / щ	m / М	} / Щ
1	1	1	0	14	ВЫХ	РЗ	.	>	N / н	^ / ч	n / Н	~ / Ч
1	1	1	1	15	ВХ	РЗ	/	?	O / о	_ / ъ	o / О	ЗБ

широкое распространение, в частности, на микрокомпьютерах, для которых упрощение и удешевление устройств ввода/вывода весьма существенно.

Поля, на которые разбивается слово при той или иной интерпретации, могут быть как одинаковой, так и разной ширины. Например, слово, интерпретируемое как код команды, содержит обычно поля кода операции, операнда или операндов, а также каких-либо признаков. При этом поле кода операции может занимать от нескольких битов и даже одного бита до полной длины слова в случае безадресной команды. Ширина полей операндов варьируется еще в большей степени: в случае неявной адресации поля операнда просто нет, а в случае задания полного абсолютного адреса оно может состоять из целого слова, а то и из двух слов, например, в 8-битном микрокомпьютере.

Вообще, традиционное понимание поля как части слова, удовлетворительное в условиях архитектуры компьютеров с большой длиной слова, применительно к микрокомпьютерам нуждается в переосмыслении. Поля следует понимать не как части слова, а как звенья кодируемого словом или несколькими словами объекта программы или данных (команды, описателя, структуры данного). При этом речь должна идти не о разбиении слова, а скорее о покрытии машинными словами указанных объектов или об отображении этих объектов на последовательность слов. Так, и в 16-битной архитектуре (не говоря уж о 8-битной), наряду с однословными командами, имеют место кодируемые двумя и тремя словами. Для представления числа с плавающей запятой также используют два или три 16-битных слова, в одном из которых выделяются поля порядка и знака числа, а оставшаяся его часть вместе со вторым или со вторым и третьим словом составляет поле мантиссы.

Интерпретация подобных многословных объектов осуществляется либо программным путем, либо при помощи специально добавляемой к базовой конфигурации компьютера аппаратуры. Типичным примером служат устройства расширенной арифметики и спецпроцессоры с плавающей запятой, подключаемые к основному процессору на правах периферийных устройств. Впрочем, давно наметилась и неуклонно, хотя и медленно, усиливается тенденция ориентировать базовый процессор на работу с командами и данными переменной длины. Это проявляется в использовании уже упоминавшихся многословных команд, укороченных и полных адресов, а также операций с операндами удвоенной и учетверенной длины, с операндами-байтами и операндами-словами.

Самой элементарной интерпретацией слова является так называемая *логическая интерпретация*, т. е. истолкование значений сло-

ва, подобно значениям бита, в смысле «ложь» и «истина», или «нет» и «да», или любой другой двузначной противоположности. Короче говоря, слово выступает в качестве бита, рассматривается как принимающее одно из двух значений, над которыми определены операции двузначной логики — отрицание (инверсия), конъюнкция, дизъюнкция, неэквивалентность и др.

Использование многозначного слова в качестве двузначного бита осуществимо разными способами. Например, можно отождествить все слова, числовые значения которых сравнимы по модулю два, сопоставив четным значениям логический 0, и нечетным — 1. Технически это равносильно различению слов по их младшему биту, независимо от значений остальных битов. Другая возможность — принять в качестве логического нуля числовой нуль, а все ненулевые значения слова отождествить с логической единицей. Но сократить число значений слова до двух можно как путем отождествления, так и путем ограничения множества принимаемых им значений. Естественно принять в качестве логических 0 и 1 числовые 0 и 1, а все другие значения слова просто не использовать. Иначе: слово в роли логического операнда может иметь значение 0 или 1, другие значения недопустимы.

Таким образом, словами можно манипулировать как битами, ограничив множество допустимых значений числами 0 и 1, т. е. 00...00 и 00...01. Однако логические операции, выполняемые компьютером, интерпретируют слово не как бит, а как n независимых битов — *булевский* n-компонентный вектор. Они производятся побитно, т. е. над каждым битом слова в отдельности или в случае двухместной операции над парами одноименных битов слов-операндов. В сущности логическими эти операции являются только по отношению к отдельным битам слова, а по отношению к слову в целом они служат средством выделения, очистки, наложения, сочленения и других неарифметических преобразований его частей (полей).

Например, выделение части слова можно осуществить логическим умножением (конъюнкцией) на слово-константу (называемое обычно маской), которое содержит единицы в битах, соответствующих выделяемой части, и нули в остальных битах. Чтобы сочленить части слов, их предварительно выделяют, а затем выполняют дизъюнкцию, возможно, в сочетании со сдвигом.

В случае, когда значениями слов-операндов могут быть только 00...00 и 00...01, операции побитной конъюнкции, побитной дизъюнкции и побитной неэквивалентности равносильны соответствующим операциям с однобитными операндами при условии, что числовые значения операндов-слов равны значениям операндов-битов. Например, нетрудно убедиться, что побитная неэквивалентность слов

равносильна в указанном смысле неэквивалентности битов:

$$00...00 \oplus 00...00 = 00...00, \text{ т. е. } 0 \oplus 0 = 0,$$
$$00...01 \oplus 00...01 = 00...00, \text{ т. е. } 1 \oplus 1 = 0,$$
$$00...00 \oplus 00...01 = 00...01, \text{ т. е. } 0 \oplus 1 = 1.$$

Тем не менее побитные логические операции над словами, принимающими только значения 00...00 и 00...01, не являются полным аналогом соответствующих операций над битами. Так, побитная инверсия слова 00...00 дает в результате не 00...01, как то следовало бы по аналогии с инверсией бита, а 11...11, что в дополнительном коде значит —1. Соответственно побитная инверсия слова 00...01 дает не 00...00, а 11...10, т. е. —2. Точно так же ведут себя двухместные побитные операции, выражающиеся через конъюнкцию (или дизъюнкцию) с инверсией. Например, операция эквивалентности

$$00...00 \sim 00...00 = 11...11$$
$$00...01 \sim 00...01 = 11...11$$

дает в результате — 1 вместо ожидаемой 1.

Как видно, побитные логические операции над словами, принимающими, подобно битам, числовые значения 0 и 1, нельзя безоговорочно употреблять в качестве операций двузначной логики. Даже ограничившись операциями булевой алгебры — конъюнкцией, дизъюнкцией и инверсией, приходится принимать меры, чтобы реализация их была корректной. Это касается инверсии, которая в отличие от побитной инверсии, инвертирующей значения всех битов слова, должна инвертировать только значение младшего бита. Поскольку значения всех битов, кроме самого младшего, для логических операндов должны быть равными нулю, то желаемая операция инверсии реализуется как побитная инверсия с последующим обнулением всех битов, за исключением самого младшего, которое осуществляется конъюнкцией с константой, содержащей 1 в младшем бите и 0 в остальных.

Набор команд компьютера может включать, наряду с побитной инверсией, команду булевской инверсии (отрицания), выполнение которой преобразует 00...00 в 00...01, а 00...01 в 00...00. Естественно, однако, потребовать, чтобы предписываемая этой командой операция была определена не только для значений 0 и 1, но и для всех других значений слова. Например, можно определить ее как взятие дополнения до 1 по аналогии с тем, что побитная инверсия представляет собой взятие дополнения до —1, а операция изменения знака числа — взятие дополнения до 0.

Заметим, что требование выполнимости операций по возможности для любых значений слов-операндов прямо противоположно принятой в большинстве языков программирования идее типов данных, согласно которой каждая операция определена и выполнима только для данных соответствующего типа. Например, логические операции могут выполняться только с данными булевского типа, принимающими значения «истина» и «ложь». Компьютер же все операции выполняет над словами длины n, принимающими каждое 2^n различных значений. В зависимости от выполняемой операции значениям слова приписывается тот или иной смысл, причем осмысленными могут быть не все возможные значения, но это не значит, что для не осмысленных значений операция не должна выполняться. Чтобы при минимальном наборе команд процессора обеспечивалась высокая эффективность обработки различного вида данных, предписываемые командами операции следует определять по возможности шире, имея в виду, что операции, подобно данным, истолковываются и используются не единственным образом.

ГЛАВА 3

АРХИТЕКТУРА МИКРОКОМПЬЮТЕРОВ: АДРЕСАЦИЯ И УПРАВЛЕНИЕ

§ 1. Прямая и непосредственная адресация

В ходе выполнения программы процессор осуществляет выборку из памяти команд и данных, запись результатов. При этом он адресуется к ячейкам памяти по их номерам. Ячейки пронумерованы последовательными целыми числами, обычно начиная с нуля, т. е. память представляет собой одномерный массив m(0:k—1), содержащий k элементов-ячеек. Совокупность значений, пробегаемых индексом этого массива (адресом памяти), называют *адресным пространством*.

Способы задания адреса в командах и соответствующие механизмы доступа к ячейкам памяти составляют важную часть архитектуры компьютера — *систему адресации*. Эта система, наряду с адресацией главной памяти, адресует также собственные регистры процессора и регистры периферийных устройств. Регистры процессора, как правило, имеют отдельную нумерацию. Периферийные регистры либо имеют отдельную нумерацию, либо включены в единое с главной памятью адресное пространство.

Номер ячейки, сообщаемый запоминающему устройству при обращении к этой ячейке, (индекс массива m) называется *абсолютным*, или *исполнительным*, или *эффективным* адресом. Обозначим его EA. Адресация заданием в команде исполнительного адреса EA называется *абсолютной адресацией*. Адресуемый операнд — ячейка m(EA). Например, команда LDA EA загрузки аккумулятора значением, хранящимся по адресу EA, выполняется как Ac:= m(EA), а команда STA EA записи текущего значения аккумулятора по адресу EA — соответственно как m(EA):=Ac. Это весьма простой, но не очень гибкий способ адресации.

Альтернативу абсолютной адресации составляет задание в команде не готового исполнительного адреса, а исходных данных для его вычисления. Чаще всего это *смещение*, т. е. величина (обозначим ее D), прибавление которой к некоторому *базовому адресу* BA дает исполнительный адрес: EA:=BA + D. Обычно смещение является числом со знаком. В сущности это адресация относительно базы BA, однако *относительной* ее принято называть только в том случае, когда базой служит текущее значение счетчи-

ка команд PC и абсолютный адрес получается так: EA:=PC + D. Когда же в качестве базы используется другой регистр, то говорят об *адресации по базе*, или о *базовой адресации*. При этом, если в составе процессора имеется несколько регистров, относительно которых возможна базовая адресация, то команда помимо смещения должна содержать также номер регистра-базы.

Относительная адресация часто применяется в командах переходов. Использование ее позволяет обходиться, как правило, коротким адресом и получать перемещаемые в адресном пространстве программы. Базовая адресация используется для доступа к данным и адресам в позиционно-независимых программах, а также при передаче подпрограмме группы параметров.

Широко распространена так называемая *индексная адресация*, при которой исполнительный адрес получается также сложением смещения и базы, но базовый адрес содержится в команде, а смещение, называемое в этом случае индексом, помещается в регистр (индексный регистр). Варьируя значение регистра, можно одной и той же последовательностью команд обрабатывать все элементы массива или таблицы. Индексация возможна одновременно по нескольким индексным регистрам. Так, существуют системы *двойной индексации*, в которых исполнительный адрес получается как сумма значений двух регистров, скажем, RI, RJ и содержащегося в команде базового адреса BA:

$$EA:=RI + RJ + BA$$

Команда в такой системе должна наряду с BA содержать номера I и J вовлекаемых в вычисление EA регистров.

Аналогом адресации смещением является *страничная адресация*. Она выражается в том, что абсолютный адрес представляется в виде сочленения (конкатенации) двух частей: старшая часть рассматривается как номер страницы памяти, а младшая — как номер ячейки в пределах этой страницы. Таким образом, память мыслится организованной как двумерный массив m(0:p—1, 0:s—1), причем число страниц $p=2^r$, а число ячеек на странице $s = 2^{n-r}$, где n — длина абсолютного адреса в битах.

Основное назначение страничной адресации — сократить длину адресной части команды. Команда должна содержать адрес в пределах страницы, длина которого составляет обычно 8—10 битов, и номер регистра — указателя страницы, занимающий 1—3 бита. В простейшем случае адресоваться можно только к нулевой и к текущей (содержащей выполняемую команду) страницам, т. е. номером страницы может быть либо 0, либо старшая часть абсолютного адреса, находящегося в программном счетчике. Для выбора одной из этих двух возможностей в команде необходим бит —

указатель страницы: «нулевая/текущая». В более развитой системе имеются специальные регистры, в которых содержатся номера открытых в данный момент страниц, а в команде должно быть поле битов, указывающих требуемый регистр. Кроме того, необходима особая команда для засылки в регистры-указатели номеров открываемых страниц.

На таком же принципе основаны схемы *расширения адресного пространства* по сравнению с обеспечиваемым путем абсолютной адресации при данной длине адреса. Старшие биты адреса используются для указания регистра, содержащего номер блока памяти, к которому производится обращение, а остальная часть адреса составляет номер ячейки в пределах блока. Если для указания регистра выделить n1 битов, а номера блоков (длины регистров) сделать n2-битными, то адресное пространство расширится в 2^{n2-n1} раз.

До сих пор речь шла о *прямой адресации*, т. е. такой, при которой адрес прямо указывает ячейку, содержащую значение операнда, а точнее, при которой содержимое ячейки, обладающей данным адресом, рассматривается как значение операнда. Адрес ячейки, одержащей значение операнда, называется также *прямым* или *адресом 1-го ранга*, *ссылкой 1-го ранга*. Существуют адреса или ссылки более высоких рангов, а также нулевого ранга.

Ссылка нулевого ранга — это непосредственное задание значения операнда в самой команде. Заданное таким образом значение называют *литералом*, а способ задания — *непосредственной адресацией*. Непосредственная адресация характеризуется минимальной гибкостью: в отличие от прямого адреса, указывающего ячейку, значение которой является переменным, может быть изменено путем присваивания иного значения, литерал представляет собой константу, замена которой связана с необходимостью изменения самой команды.

§ 2. Косвенная адресация

Адресация с использованием адреса, обладающего рангом выше 1-го, называется *косвенной*, как и сам такой адрес. Сущность косвенной адресации в том, что заданный в команде адрес не является прямой ссылкой на текущее значение операнда, а указывает ячейку, в которой опять-таки содержится адрес, но на 1 меньшего ранга. Как правило, применяется однократная косвенность (максимальный ранг адреса равен 2), но существуют системы и с многократной косвенной адресацией.

Для обозначения косвенной и непосредственной адресации в языке ассемблера используются специальные знаки @ и #, смысл которых можно уяснить на примере команды LDA — «загрузить

аккумулятор». Записанная в виде LDA EA эта команда вызывает присваивание AC:=m(EA). Если же перед EA поставить знак @, то получим команду LDA @ EA с адресом 2-го ранга, вызывающую присваивание Ac:=m(m(EA)). Другой способ обозначения косвенности — заключение адреса в круглые скобки, т. е. вместо @ EA пишут (EA). Знак # указывает на то, что величина, перед которой он поставлен, является литералом. Так, команда LDA # EA вызовет присваивание Ac:=EA. В коде команды для указания способа адресации отводится поле, значениям которого соответствуют приведенные и подобные им знаки.

Косвенность сообщает адресации наибольшую гибкость. Если использование прямого адреса позволяет осуществить переменную величину, то с косвенным адресом связана возможность применения данной команды или последовательности команд (процедуры) к любой из рассматриваемых переменных, например, к любому элементу некоторого массива. Достигается это путем замены или модификации адреса 1-го ранга, хранящегося в ячейке, на которую указывает косвенный адрес. Так, одна и та же подпрограмма может использоваться для обработки данных то в одной, то в другой области памяти, если эти данные адресуются в ней косвенно, и в ячейки, хранящие адреса 1-го ранга, при каждом обращении к этой подпрограмме производится засылка фактических адресов применительно к обрабатываемой области.

Возможности, обеспечиваемые косвенной адресацией, приобретаются ценой очевидного усложнения архитектуры и соответственно увеличения трудности ее освоения и использования. Кроме того, для реализации косвенной ссылки требуется два обращения к памяти, так что по сравнению с прямой косвенная адресация в два раза медленнее. Впрочем, последнее верно только в том случае, когда и значения операндов, и прямо ссылающиеся на них адреса (адреса 1-го ранга) хранятся в главной памяти, что типично для миникомпьютеров 60-х годов. В современных микрокомпьютерах имеет место *регистровая косвенная адресация*, при которой адреса хранятся не в ячейках главной памяти, а в быстродействующих регистрах процессора, благодаря чему замедление отсутствует. Оно возникает лишь с адресами 3-го ранга, при двукратной косвенности.

Регистровая адресация (не обязательно косвенная), т. е. адресация с участием регистров процессора, является в микрокомпьютерах главной. В условиях короткого машинного слова манипулировать номерами регистров, содержащих адреса, выгоднее, чем непосредственно самими адресами. Поэтому в командах, как правило, содержатся не адреса (для представления которых приходится удлинять команду добавлением к основному слову дополнительных слов), а номера регистров, содержащих адреса или значения операндов.

В случае прямой адресации номер регистра в коде команды указывает на содержащееся в этом регистре значение операнда или на сам регистр как приемник значения, а в случае косвенной — на содержащийся в регистре адрес ячейки главной памяти. При записи на языке ассемблера прямая регистровая адресация операнда обозначается обычно буквой R, сопровождаемой номером регистра, например: R2, R5. Косвенность символизирует знак @ или заключение имени регистра в круглые скобки, например: @ R2 или (R2). Смысл этих обозначений поясним на примере операций, соответствующих вариантам команды MOV с различными сочетаниями способов адресации операндов.

Команда				Операция
MOV R3,	R4			R4:= R3
MOV @R3,	R4	или	MOV (R3), R4	R4:= m(R3)
MOV R3,	@R4	или	MOV R3, (R4)	m(R4):= R3
MOV @R3,	@R4	или	MOV (R3), (R4)	m(R4):= m(R3)

Множество комбинированных способов адресации порождается в результате сочетания косвенности с относительной, страничной, базовой и индексной адресацией. Например, в системе, допускающей прямую адресацию к нулевой и текущей страницам, с помощью косвенной адресации можно обращаться к любой ячейке памяти при наличии ее абсолютного адреса в одной из ячеек нулевой или текущей страницы. Индексируя этот адрес, можно осуществить перебор ряда ячеек — элементов массива или таблицы.

В условиях косвенной адресации индексировать можно как отправной, заданный в команде адрес (при однократной косвенной адресации адрес 2-го ранга), так и любой из адресов меньшего ранга, в частности, завершающий цепочку адрес 1-го ранга. Теоретически в случае многократной косвенности возможно множество вариантов индексирования, различающихся по числу и по рангам индексируемых адресов. Но на практике, как правило, ограничиваются однократной косвенностью и индексируют либо только отправной, либо только завершающий адрес. В первом случае адресация называется *преиндексной*, а во втором — *постиндексной*.

Наряду с косвенной индексной адресацией, широко применяется так называемая *автоиндексная*, заключающаяся в том, что значение указываемой косвенным адресом ячейки или регистра при каждом обращении автоматически увеличивается или уменьшается на некоторую фиксированную величину (обычно на 1 или на 2). Автоиндексация прибавлением называется *автоинкрементной*, автоиндексация вычитанием — *автодекрементной* адресацией. За-

метим, что в отличие от индексной адресации, которая заключается в использовании суммы «адрес плюс индекс» при сохранении адреса неизменным, автоиндексация представляет собой изменение адреса, указываемого косвенным адресом. Это изменение может производиться либо до, либо после того, как по данному адресу будет произведено обращение. Если изменение адреса предшествует использованию его для обращения, то адресация называется *преавтоиндексной* (*преавтоинкрементной*, *преавтодекрементной*). Если же адрес модифицируется после того, как произведено обращение, адресацию называют *поставтоиндексной* (*постинкрементной*, *постдекрементной*). Обратим внимание на то, что пре- и поставтоиндексация имеют совсем иной смысл, чем рассмотренные ранее пре- и постиндексация. Так, если команда содержит косвенный адрес A и индексация осуществляется с помощью регистра RI, а автоиндексация — добавлением или вычитанием единицы, то вычисление исполнительного адреса EA применительно к перечисленным видам адресации будет следующим:

преиндексация: EA:=m(m(A+RI));
постиндексация: EA:=m(m(A)+RI;
преинкрементная адресация: m(A):=m(A)+1; EA:=m(A);
предекрементная адресация: m(A):= m(A)—1; EA:=m(A);
постинкрементная адресация: EA:=m(A); m(A):=m(A)+1;
постдекрементная адресация: EA:=m(A);m(A):=m(A)—1;

Автоиндексация служит средством последовательного перебора элементов массивов, а также реализации стеков и очередей.

§ 3. Неявная адресация

Помимо *явной адресации*, т. е. такой, при которой операнд задан в команде адресом или данными для вычисления адреса, а также указанием способа доступа к операнду, широко применяется *неявная* (подразумеваемая) адресация. Наиболее известным примером неявной адресации служит умалчивание адреса одного из операндов, а также приемника значения результата; в одноаккумуляторной архитектуре подразумевается, что источником значения неуказанного операнда и приемником значения результата операции служит аккумулятор. По принципу неявной автоинкрементной адресации работает программный счетчик.

Воплощением более развитого и весьма важного вида неявной адресации является *стек* (магазин) — память, организованная по принципу «последним вошел — первым вышел». В противоположность массиву, элементы которого пронумерованы подобно местам в кинозале и доступны по их номерам для присваивания и копирования значений, стек представляет собой не совокупность фиксированных мест (ячеек), а динамичную последовательность безымянных элементов, уложенных друг на друга, подобно патронам в магазине пистолета-автомата. Из всех находящихся в стеке элементов доступен всегда только самый верхний, называемый *вершиной стека*. Доступ к элементу, расположенному в глубине стека, возможен обычно лишь путем удаления всех поступивших после него и находящихся над ним элементов, так чтобы этот элемент стал вершиной. При удалении из стека верхнего элемента вершиной становится элемент, являвшийся до этого *подвершиной*, т. е. располагавшийся непосредственно под вершиной, и общая глубина стека (число содержащихся в стеке элементов) уменьшается на 1. При занесении элемента в стек этот элемент занимает положение вершины, прежняя вершина переходит в подвершину, а общая глубина стека увеличивается на 1.

Важным преимуществом стека по сравнению с адресной организацией памяти является то, что его элементами можно манипулировать, не адресуясь к ним явно, не называя их имен. Например, процедуру сложения двух чисел применительно к стеку, элементами которого являются числа, можно организовать следующим образом:

— взять из стека 1-е слагаемое;
— взять из стека 2-е слагаемое;
— сумму заслать в стек.

Замечательно, что значение суммы, автоматически заместившее в стеке значения слагаемых, можно, также не именуя, употребить в качестве операнда какой-либо последующей операции. Развитие этой возможности приводит к весьма экономной бесскобочной *постфиксной записи* выражений, называемой по-другому польской инверсной записью (ПОЛИЗ), которая непосредственно выполнима на стековом процессоре, является для него готовой программой.

Стековый процессор можно рассматривать как усовершенствование аккумуляторного процессора путем замены аккумулятора стеком или добавления к аккумулятору, используемому в качестве вершины стека, подвершины и ряда более глубоких ступеней для автоматического запоминания значений, вытесняемых из вершины при засылке в нее новых операндов. В обычном аккумуляторе засылка нового значения уничтожает старое, поэтому его, если нужно, сохраняют, предусматривая перед засылкой команду копирования аккумулятора в ячейку памяти.

Выборку команд и исходных данных стековый процессор производит из обычной (адресуемой) памяти, но значения промежуточных результатов автоматически сохраняются и реализуются в стеке. При этом оказывается возможным и целесообразным расще-

пить традиционные команды с их полями операции и операндов на элементарные операционные и адресные команды. Адресная команда содержит только адрес и предписывает заслать в стек указываемое этим адресом значение. Операционная команда представляет собой код операции, которую надлежит выполнить процессору над стеком.

Проиллюстрируем работу такого процессора на примере выражения (A + B) * (C—D). Условимся, что адресным командам соответствуют буквы, а операционным командам — знаки операций. Программой вычисления данного выражения является его постфиксная запись:

A B + C D — *

Реализуя эту запись, процессор зашлет в стек значение A, затем — значение B, затем выполнит над стеком операцию +. В ходе выполнения операции + из стека будут взяты значения A и B, а значение их суммы (A + B) будет заслано в стек. Далее в стек поступят значения C и D, а затем операция — (вычитание) извлечет их из стека и зашлет в него значение разности (C — D). После этого в стеке будет два значения: (A + B) и (C — D). Операция * (умножение) удалит их из стека, перемножит и зашлет в стек полученный результат.

Достоинства стекового процессора далеко не исчерпываются экономностью реализации выражений. Не менее важна, например, возможность передавать через стек аргументы процедур и таким образом пользоваться процедурами, как операциями, обращаясь к ним по именам без параметров. Вместе с тем, стек представляет собой идеальный механизм *гнездования* (осуществления *вложенности*) процедур, применяемый сегодня во всех процессорах, не только стековых. При каждом обращении к процедуре адрес непосредственно следующей за этой процедурой команды («адрес возврата») запоминается в стеке, а по окончании процедуры передается в программный счетчик. В случае многократной вложенности, т. е. когда в теле процедуры содержится процедура, также вызывающая процедуру, и т. д., в стеке возникает последовательность адресов возврата, передаваемых затем в программный счетчик в обратном порядке по мере завершения выполнения соответствующих тел. Аналогично обрабатывается *рекурсивная* процедура, обращающаяся к самой себе.

Особенно эффективной оказывается двухстековая архитектура процессора, располагающая отдельными стеками операндов и адресов возврата. Эта архитектура составляет благоприятную основу для реализации структурированного программирования в режиме диалога, чему будет посвящена одна из дальнейших глав.

Технически механизм стека можно осуществить двумя путями. Первый путь буквально повторяет конструкцию пистолетного магазина: несколько регистров соединяются в цепочку, по которой содержимое регистров можно передавать (сдвигать) в прямом и обратном направлении. При засылке значения в стек производится сдвиг на одно звено цепочки в направлении от вершины, а при удалении значения — такой же сдвиг в направлении к вершине.

Второй путь состоит в использовании ряда ячеек адресуемой памяти с доступом посредством *указателя стека*. Указателем обычно служит регистр, принимающий в качестве значений номера используемых в стеке ячеек. Засылаемое в стек значение записывается в указываемую указателем ячейку и сохраняется в этой ячейке «неподвижно», пока не будет удалено,— вместо перемещения элементов изменяется значение указателя. В зависимости от того, увеличивается или уменьшается значение указателя по мере заполнения стека, а также от того, когда модифицируется указатель — до или после записи заносимого в стек значения, возможны четыре разновидности стека. Если значение указателя увеличивается при занесении элемента в стек, то стек растет в направлении возрастания адресов, т. е. является восходящим, в противном случае он будет нисходящим, т. е. растущим в направлении убывания адресов ячеек. Если указатель стека модифицируется перед записью заносимого значения, то он указывает на вершину стека, т. е. на ячейку, содержащую последнее занесенное в стек значение. Если же модификация указателя производится после записи занесенного значения, то он указывает надвершину — первую из еще не занятых стеком ячеек.

Процедура PUSH R занесения значения регистра R в стек с текущим значением указателя P для данных четырех вариантов выражается так:

1) P:=P+1; m(P):=R;
2) P:=P—1; m(P):=R;
3) m(P):=R; P:=P+1;
4) m(P):=R; P:=P—1;

Обратная процедура POP R изъятия из стека элемента и присваивания его значения регистру R для тех же четырех вариантов выражается следующим образом:

1) R:=m(P); P:=P—1;
2) R:=m(P); P:= P+1;
3) P:=P—1; R:=m(P);
4) P:=P+1; R:=m(P);

Видно, что как восходящий, так и нисходящий стек можно осуществить, используя преинкрементную и постдекрементную или предекрементную и постинкрементную адресацию.

Механизмом, родственным стеку, но работающим по правилу «первым прибыл — первым выбыл», является *очередь*. Основное назначение очереди — сглаживание неравномерного потока заявок или данных на входе обслуживающего или обрабатывающего устройства (процесса).

Очередь представляет собой массив ячеек памяти с двумя указателями. По указателю P1 производится запись в очередь, по указателю P2 — обслуживание или обработка. Обычно указатели работают как счетчики по модулю 2^k и соответственно используемый массив содержит 2^k ячеек. Указатели могут работать в автоинкрементном или в автодекрементном режиме, но их режимы должны быть одинаковыми.

При P2=P1 очередь пуста, необработанных заявок (данных) нет. В этом состоянии доступ по P2 запрещен, возможна только запись в очередь по P1. В случае инкрементного режима, полагая, что источником очередников является регистр R1, имеем процедуру записи в виде

m(P1):= R1; P1:=P1+1;

Прием из очереди на обслуживание производится в регистр R2 при условии P2 ≠ P1:

if P2 ≠ P1 **then begin** R2:=m(P2); P2:=P2+1 **end**

Если интенсивность потока заявок в среднем превышает интенсивность обслуживания, то возможно переполнение очереди.

Переполнение наступает при P2=P1+1.

§ 4. Способы и средства управления последовательностью операций

Процедура является важнейшим средством организации и структурирования деятельности. В простейшем случае она представляет собой четко описанную цепочку операций, снабженную собственным именем, которое обозначает реализованное в виде этой цепочки составное действие. Употребление имени процедуры равносильно применению данного действия в том контексте, в котором употреблено имя. Правда, в языках программирования, как правило, имя процедуры употребляют в сочетании с ключевым словом, символизирующим так называемое обращение к процедуре или ее вызов (call), но это уже технические детали. Существо же дела в том, что сколь угодно большую и сложную совокупность операций можно обозначить именем и мыслить ее укрупненно как

результирующее действие, отвлекаясь от подробностей реализации последнего. Употребляющему процедуру важно знать, *что* она делает, но обычно неважно знать, *как* делает.

В общем случае описание процедуры не сводится к перечислению операций, образующих линейную цепочку. Процедура может быть ветвящейся, содержать циклически повторяющиеся операции или последовательности операций. Наряду с элементарными (базовыми) операциями в описание процедуры могут входить процедуры (вложенные процедуры, описания которых в свою очередь могут содержать вложенные процедуры, и т. д.). Именно вложенность (называемая также гнездованием) процедур составляет основу структурирования.

Линейная последовательность действий описывается в виде соответствующей последовательности предложений или команд, отделяемых друг от друга точкой с запятой, а в языке ассемблера — литерой «возврат каретки». Особый случай представляет последовательность вычислительных операций, задаваемая в форме арифметического выражения. Такое задание не всегда однозначно. Например, включающее выражение суммы трех переменных A, B, C предложение

S:=A+B+C;

с учетом коммутативности и ассоциативности сложения может быть реализовано различным образом, в частности, в следующих вариантах:

1) S:=A; S:=S+B; S:=S + C;
2) S:=B; S:= S + C; S:= S + A;
3) S:=C; S:=S + A; S:=S + B;

Ввиду того, что значения слагаемых могут быть разных знаков, а длина регистра S конечна, и поэтому не исключено переполнение, нельзя гарантировать, что значение суммы во всех вариантах будет одним и тем же. Чтобы устранить подобную неоднозначность, устанавливают определенную последовательность выполнения операций одинакового старшинства, например, в порядке их появления при просмотре выражения слева направо. Верным средством задания желаемой последовательности операций в выражении являются скобки.

Замечательной принадлежностью компьютера является то, что он не только обеспечивает строгое соблюдение предписанной последовательности операций, но и позволяет осуществить управление этой последовательностью с учетом условий, складывающихся в ходе ее выполнения. Способ задания и реализации этого управления оказывает решающее влияние на структуру программ и бук-

вально на характер мышления людей, занимающихся их созданием и использованием.

Традиционный способ управления последовательностью выполняемых операций заключается в использовании *меток* и *команд перехода* по данной метке (на данную метку). На участках программы, не содержащих команд перехода, команды выполняются друг за другом в порядке их расположения в тексте программы (в линейной последовательности). Команда перехода позволяет прекратить дальнейшее выполнение программы в линейном порядке и переключиться на команду, указанную при помощи метки. Метка — это число или сочетание литер, поставленное перед командой, на которую может быть произведен переход (такая команда называется помеченной), и используемое в команде перехода в качестве указателя, куда переходить. Разумеется, не должно быть двух команд, помеченных одной и той же меткой, но может быть несколько команд перехода на одну и ту же метку. В машинном коде команда перехода содержит не метку, а адрес команды, на которую она производит переход, или данные для вычисления этого адреса.

Чтобы управлять процессом выполнения программы с учетом его текущего состояния, команды перехода тестируют тот или иной параметр этого процесса в отношении заданных в них условий. Такой условный переход выполняется только в случае, когда значение тестируемой величины удовлетворяет заданному условию, а если условие не соблюдено, то переход не производится и в качестве очередной выполняется команда, расположенная непосредственно после команды перехода, т. е. продолжается линейная последовательность. Условие задается в виде отношения, выполнение которого проверяется при текущих значениях его членов. Например, команда, производящая переход по метке L5, если значение переменной X отрицательно, записывается в виде

if X $<$ 0 **go to** L5 — если X $<$ 0, перейти на L5.

Наряду с командами условного перехода имеется также команда безусловного перехода, предписывающая переход без условий. Например, безусловный переход по метке LABLE:

go to LABLE — перейти по метке LABLE.

Команды переходов являются универсальным (обеспечивающим возможность построения произвольного алгоритма) и эффективным в отношении компактности и быстродействия программ средством управления. Однако именно с критики **go to** началось становление структурированного программирования. Было показано, что бесконтрольное применение переходов является причиной запутанности и ненадежности программ, а также главным препятствием в решении таких актуальных проблем как повышение продуктивности программистского труда и обеспечение удовлетворительной достоверности больших программных систем.

В условиях управления при помощи переходов, да к тому же стремясь достичь максимума машинной эффективности, программист легко приучается рассматривать программу с точки зрения процессора, который будет эту программу выполнять. Программа составляется не так, чтобы ее было проще понять человеку, а так, чтобы компьютер мог быстрее ее выполнить. Получается, что человек идет на поводу у машины. Такое положение можно было в какой-то мере оправдывать, пока компьютеры были очень дорогими, а объем программ сравнительно невелик — уступка машине выходила экономически целесообразной и посильной для человека. Но с удешевлением компьютеров, увеличением размеров программ и расширением круга пользователей бесструктурные программы с управлением командами переходов стали тормозом дальнейшего прогресса.

§ 5. Средства структурированного управления

В качестве средств управления, призванных обеспечить построение структурированных программ, обычно указывают три схемы задания последовательности операций: *линейную последовательность*, *ветвление по условию* и *цикл с выходом по условию*.

Линейная последовательность — это то же, что и в обычной программе, выполнение команд в том порядке, в котором они расположены в тексте программы.

Схема ветвления или выбора задается в виде

if ⟨отношение⟩ **then** ⟨последовательность команд S1⟩ **else** ⟨последовательность команд S2⟩ **endif**

Если помещенное между **if** и **then** отношение удовлетворено, то выполняется последовательность команд S1, в противном случае — последовательность S2. Слова **then**, **else** и **endif** играют роль скобок, выделяющих выбираемые по условию последовательности команд («ветви»).

Имеется сокращенный вариант этой конструкции:

if ⟨отношение⟩ **then** ⟨последовательность команд⟩ **endif**

Заключенная между **then** и **endif** последовательность команд выполняется в случае, когда отношение удовлетворено, и игнорируется в противном случае. Оба варианта употребляются также в форме без **endif**, которая требует введения скобок **begin**, **end**, если последовательность включает более одной команды.

Часто употребляемой схемой цикла является **while-do** — «делай, пока»:

while ⟨отношение⟩ **do** ⟨последовательность команд⟩ **enddo**

Последовательность команд, заключенная между **do** и **enddo** (тело цикла), выполняется снова и снова, пока удовлетворено заданное отношение. Проверка отношения предшествует каждому возможному выполнению тела цикла. Если отношение оказалось не соблюденным при первой проверке, то тело цикла не будет выполнено ни разу. В записи без **enddo** необходимы скобки, если тело содержит более одной команды.

Другая широко применяемая схема цикла — итеративная:

for ⟨переменная⟩ := ⟨выражение E1⟩ **to** ⟨выражение E2⟩ **do** ⟨последовательность команд⟩ **enddo**

Расположенной после **for** так называемой управляющей переменной присваивается в качестве начального значение выражения E1, а значение выражения E2 является ограничителем интервала значений, принимаемых этой переменной в результате прибавления к ней единицы после каждого выполнения тела цикла. Таким образом, тело цикла выполняется для всех последовательно возрастающих с шагом 1 значений управляющей переменной, не превышающих граничного. В записи без **enddo** тело цикла, содержащее две или более команды, надо заключать в скобки **begin, end.**

Примером использования цикла **for** для описания архитектуры компьютера служит процедура «обнуления», т. е. присваивание значения 0 регистру-переменной R, рассматриваемой как массив битов R (15 : 0). Значение 0 присваивается последовательно каждому биту массива:

for i := 0 **to** 15 **do** R(i) := 0 **enddo**

В рассмотренных схемах управления нет ни меток, ни переходов — в них речь идет непосредственно о выполнении или невыполнении той или иной последовательности команд, т. е. тех или иных действий. Несмотря на то, что реализация этих схем в условиях современного состояния архитектуры компьютеров сводится в конечном счете к переходам, программист может об этом не думать и даже не знать, конструируя свою программу путем линейного сочленения данных схем (каждая схема имеет один вход и один выход) и вложения их друг в друга (имеющиеся в составе схем «последовательности команд» могут включать в качестве своих членов конкретные экземпляры схем, являющиеся в сущности командами управления).

В результате ведущейся с конца 60-х годов пропаганды структурированного программирования приведенные выше и подобные им схемы управления получили широкую известность как средство, открывающее возможность коренного улучшения практики создания и использования программ — существенного увеличения производительности труда программистов, повышения надежности программ, снижения затрат на их эксплуатацию и т. д. Соответствующие предложения или команды управления имеются сегодня практически во всех языках программирования, как во вновь созданных, так и в старых, которые так или иначе дополнены ими (например, в виде макросов структурированного программирования в языке макроассемблера). Перестройке подверглись программы и методика обучения программистов, модернизированы и написаны заново учебники.

Однако ожидавшегося эффекта эти мероприятия пока не дали. Трудозатраты на разработку и сопровождение программ, если и уменьшились, то незначительно. Надежность по-прежнему остается острейшей проблемой. Даже рьяные поборники идеи структурирования признают, что революция не удалась. Это надо понимать так, что преследуемые цели не достигнуты и проблема остается нерешенной. Данная проблема является, пожалуй, важнейшей из проблем развития компьютерной техники и особенно развития микрокомпьютеров, широчайшие и разнообразнейшие применения которых требуют общедоступной, надежной и эффективной технологии создания и применения программ.

Можно указать ряд причин неудачи структурированного программирования. Прежде всего говорят, что слабой была организация внедрения, т. е. с недостаточной энергией понуждали программистов переходить на новый стиль, и поэтому все осталось по-старому.

Другая, более серьезная причина заключается в том, что «смена стиля» — не такое простое дело, как полагают. Требуется полностью изменить характер мышления, выработанный с большим трудом и закрепленный длительной практикой. Опыт показывает, что люди, мастерски владеющие программированием с переходами, оказываются просто неспособными действовать новым способом, несмотря на то, что теоретически будто бы постигли его сущность и поняли связанные с ним преимущества. Во всяком случае, не знакомых с программированием научить конструированию структурированных программ несравнимо легче, чем уже программировавших в традиционной манере.

Но и с неискушенными не просто: структурированное программирование еще не господствует в этом мире, архитектура компьютеров и программирование на языке ассемблера базируются на уп-

равлении с помощью переходов, книги и журналы изрисованы хитроумными блок-схемами машинно-эффективных алгоритмов и программ — короче говоря, программисту нельзя еще не знать старой техники. Правда, если новое уже привилось, его шансы больше, хотя не все, конечно, устоят перед соблазном пожертвовать структурой ради экономии ресурсов или какой-нибудь другой заманчивой выгоды.

Наконец, самая значительная причина неудачи «структурированной революции» состоит в том, что средства управления, призванные на смену переходам, по существу не соответствуют преследуемой цели, не обеспечивают эффективного построения программ с желаемой структурой. Достаточно заметить, что в предложенных схемах отсутствует главное орудие структурирования — процедура.

Конечно, процедуры и без того имеются в языках программирования — сам термин происходит из алгола-60, впрочем, как и конструкции **if-then-else** и циклов. Но по традиции процедура или подпрограмма — это средство сокращения текста программы: последовательность команд, неоднократно встречающуюся в программе, обозначают именем, которое используется вместо копирования этой последовательности во всех местах ее вхождения. Стандартные или библиотечные процедуры представляют собой лишь дальнейшее развитие этой же идеи экономии — аналогичным образом употребляются имена имеющихся в системе программирования готовых программных модулей. И обычно рекомендуется не злоупотреблять процедурами, поскольку они существенно замедляют выполнение программы, так сказать, сопряжены с большими накладными расходами.

Такое представление о роли процедур, сохраняющееся и поныне, с точки зрения структурированного программирования является дезориентирующим. Экономия бумаги, компьютерной памяти и даже трудозатрат на копирование повторяющихся фрагментов текста — все это не должно заслонять того главного, что вообще упущено из виду: процедура является важнейшим средством структурирования программы, построения ее в форме иерархии постепенно разукрупняемых частей.

Программист будет вынужден мыслить в терминах процедур и соответствующим образом строить программу, если адекватно истолкованная категория процедуры займет надлежащее место в языке программирования. Другими словами, язык программирования должен понуждать (а лучше — вынуждать) программиста к построению программы в виде иерархии процедур.

Легко видеть, что схемы управления, рекомендованные в качестве базы структурированного программирования, не соответствуют данному требованию. Процедур в них просто нет, хотя выбираемые или повторяемые по условию «последовательности команд» только по форме не являются процедурами. Кроме того, последовательность команд, в частности, может состоять из одной-единственной команды вызова процедуры. Последнее значит, что возможность структурирования программы не исключена. Однако по привычке и в погоне за эффективностью программисты предпочитают другую возможность — возможность написать длинную последовательность команд, которая исчерпывала бы проблему в один прием.

Ясно, что в языке, ориентированном на структурированное программирование, эта вторая возможность должна быть либо ограничена, либо вовсе исключена. Исключить ее из рассматриваемых схем управления можно, сократив «последовательность команд» до одной команды, которая, в частности, может быть командой вызова процедуры, и запретив использование скобок **begin-end**. В результате получим:

if ⟨отношение⟩ **then** ⟨команда⟩ **else** ⟨команда⟩;
while ⟨отношение⟩ **do** ⟨команда⟩;
for ⟨переменная⟩ := ⟨выражение⟩ **to** ⟨выражение⟩ **do** ⟨команда⟩.

Менее суровой мерой может быть ограничение длины последовательностей команд, скажем, 3 — 5 командами, но во всяком случае ограничение необходимо и должно быть строгим. Это одно из главных проявлений дисциплины структурированного программирования.

Аналогичным образом должна быть ограничена длина последовательностей команд, используемых вне схем выбора и цикла в качестве безусловно выполняемых линейных последовательностей. Например, в качестве последовательности команд, составляющей тело процедуры. Технически такое ограничение может выражаться в том, что, скажем, длина описания процедуры будет ограничена одной-двумя строками. Это предотвратит попытки изложения всех подробностей сразу — программисту придется на каждом этапе пользоваться достаточно крупными процедурами, чтобы последовательность их вызовов не превысила дозволенной длины. Это обеспечит обозримость описаний процедур и постепенность детализации программы.

Конечно, и при всех строгостях отыщутся лазейки для дурного программирования. Можно, например, вместо нисходящего разукрупнения принять обратный порядок, начинать с деталей и продвигаться снизу вверх. Будем, однако, надеяться, что предлагаемое укрепление дисциплины, по меньшей мере, поможет усовершенствоваться тем, кто к этому стремится.

§ 6. Усовершенствование структурированного управления

Существенным недостатком рассматриваемых схем управления является также то, что они не обеспечивают возможность эффективной реализации произвольного алгоритма. Известны схемы управления, не представимые композициями, построенными из данных схем.

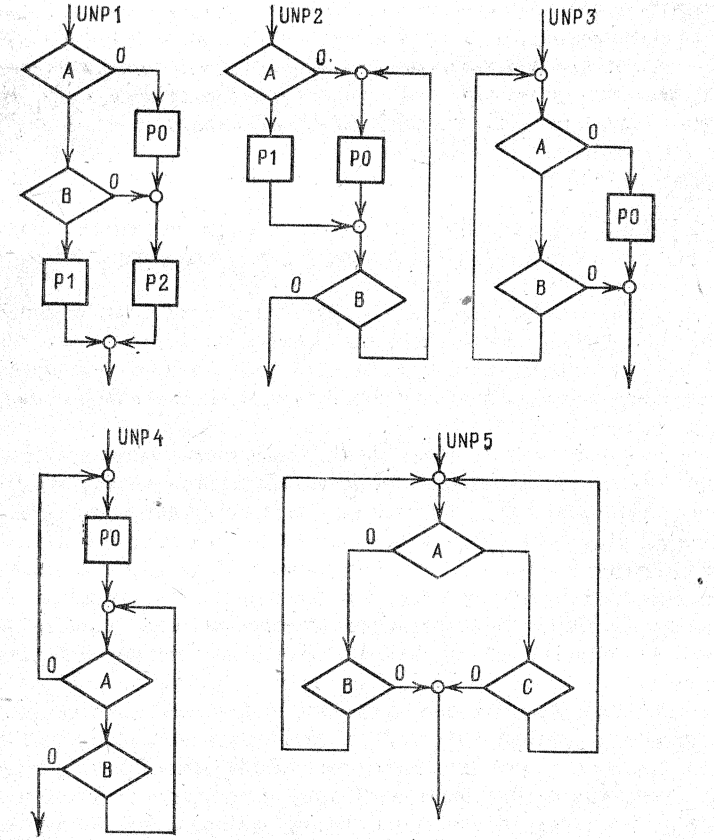


Рис. 1

Такие схемы называют неструктурируемыми. Чтобы запрограммировать их, не пользуясь переходами, прибегают к изощренным приемам (например, к введению булевских флажков для передачи информации о состояниях), вследствие чего утрачивается ясность, в достижении которой состоит цель структурирования.

Исследования показали, что источниками неструктурируемости служат пять следующих схем управления: так называемое аномальное ветвление, цикл с несколькими входами, цикл с несколькими выходами, перекрывающиеся циклы, параллельные циклы (рис. 1). Путем непосредственной проверки можно убедиться в том, что все эти схемы благополучно программируются, если в дополнение к рассмотренным конструкциям ветвления и цикла использовать команду выхода (возврата) из повторяемой в цикле процедуры. Замечательно, что при этом удается разукрупнить схему цикла **while-do** путем обособления совмещенных в ней предписаний «повторно выполнять» и «закончить выполнение, если не удовлетворяется заданное условие», осуществив последнее с помощью ветвления

if ⟨отношение⟩ **then** ⟨команда выхода⟩

Другими словами, схему управления, реализуемую командой **while-do**, предлагается осуществлять, используя пару более элементарных команд:

repeat ⟨имя процедуры⟩ — выполнять названную процедуру снова и снова неограниченное число раз,

EXIT — прекратить дальнейшее выполнение повторяемой в цикле процедуры и приступить к следующей в линейном порядке команде программы.

Возможности этой пары команд существенно шире возможностей интегральной команды **while-do**. Последняя позволяет выйти из цикла только в одной точке — перед очередным выполнением процедуры, составляющей тело цикла, а при использовании пары цикл можно покинуть в любой точке тела. Например, поместив выход по условию в конце тела, получим нередко применяемую схему цикла **do-until**. Именно данное расширение возможностей позволило эффективно структурировать ранее неструктурируемые схемы.

Можно, впрочем, возразить, что произошла подмена самого понятия структурируемости, поскольку изменен набор базовых конструкций, относительно которого обычно определяют структурированное программирование. К тому же поборники строгости ставят под сомнение приемлемость выхода из процедуры, усматривая в нем замаскированный переход. Но если иметь в виду, что целью является рациональная организация программ, а не отстаивание того или иного набора команд, то указанное возражение оказывается несущественным. А что касается неправомерности выхода из процедуры, то нетрудно видеть, что общего с переходом по метке у него не более, чем у перехода от команды к команде в линейной последовательности. И кроме того, выход, называемый обычно возвра-

том из процедуры, неявно содержится во всяком использовании процедур. Таким образом, выход из процедуры не введен, а лишь сделан явным, чтобы можно было производить его по условию. Существенно то, что выход представляет собой возврат из повторяемой процедуры, т. е. из тела цикла.

В качестве примера применения усовершенствованных команд структурированного программирования приведем программы, реализующие пять названных выше базовых неструктурируемых схем, блок-диаграммы которых в традиционной символике представлены на рис. 1. Определения процедур в этих программах записаны в следующей форме:

: ⟨имя процедуры⟩ ⟨тело процедуры⟩ ;

Двоеточие перед именем определяемой процедуры обозначает команду «Именовать данным именем следующую за ним последовательность команд (тело процедуры)». Точка с запятой символизирует конец определения. Команды, составляющие тело процедуры, отделены друг от друга пробелами, так же как и тело от предшествующего ему имени определяемой процедуры. Командой вызова процедуры является просто имя этой процедуры. Каждая программа представляет собой нисходящую последовательность определений, постепенно детализирующих заданную блок-диаграммой процедуру до уровня составляющих диаграмму элементов.

Процедура UNP1 — аномальное ветвление.

```
: UNP1   if A = 0 then PA else PB ;
: PA     P0 P2;
: PB     if B = 0 then P2 else P1 ;
```

Процедура UNP2 — цикл с несколькими (с двумя) входами.

```
: UNP2   if A = 0 then PA0 else PA1 ;
: PA0    P0 repeat PB ;
: PA1    P1 repeat PB ;
: PB     if B = 0 then EXIT else P0;
```

Процедура UNP3 — цикл с несколькими (с двумя) выходами.

```
: UNP3   repeat PAB;
: PAB    if A = 0 then PA else PB ;
: PA     P0 EXIT ;
: PB     if B = 0 then EXIT ;
```

Процедура UNP4 — пересекающиеся циклы.

```
: UNP4   P0 repeat PAB ;
: PAB    if A = 0 then P0 else PB ;
: PB     if B = 0 then EXIT ;
```


Процедура UNP5 — параллельные циклы.

```
: UNP5   repeat PA ;
: PA     if A = 0 then PB else PC ;
: PB     if B = 0 then EXIT ;
: PC     if C = 0 then EXIT ;
```

На рис. 2 представлены структурированные блок-диаграммы процедур UNP1 и UNP4. На этих диаграммах, чтобы сделать их отличными от традиционных диаграмм и легко рисуемыми, для изображения процедур вместо прямоугольников приняты кружки, а для

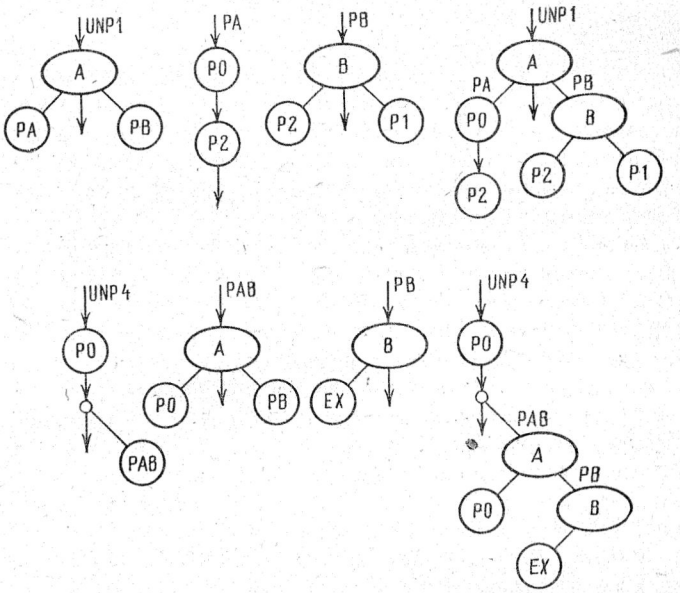


Рис. 2

изображения ветвлений вместо ромбов — овалы. Повторение изображается в виде малого кружка, к которому «подвешена» повторяемая процедура. Линии со стрелками, как и на традиционной диаграмме, обозначают сочленение в линейной последовательности. Линии без стрелок соответствуют вложенности, т. е. после выполнения процедуры или цепочки процедур, «подвешенной» к кружку или овалу посредством линии без стрелки, управление возвращается на выход той конструкции, к которой «подвешена» эта процедура или цепочка.

Процедуры UNP1 и UNP4 представлены каждая сначала в виде нисходящей последовательности диаграмм, соответствующих

определениям промежуточных процедур PA, PB, PAB, вводимых в процессе постепенного разукрупнения исходной процедуры, а затем — в виде единой, нерасчлененной диаграммы, в которой именами промежуточных процедур помечены входы соответствующих ветвей. Символ EX использован в качестве сокращенного обозначения процедуры EXIT — выход из цикла.

Блок-диаграммы структурированных программ, как расчлененные, так и нерасчлененные, по-видимому, мало что могут добавить к тексту этих программ в отношении обозримости и понятности, поскольку определения процедур в нем коротки, а иерархию вложенности можно отразить путем разделения текста на абзацы и подабзацы. Понятности текста скорее будет способствовать включение в него комментариев, содержательно раскрывающих функции определяемых и используемых в определениях процедур. Однако блок-диаграммы, как показывает опыт, удобны на этапах проектирования, отладки и тестирования программ. Это можно объяснить тем, что блок-диаграмма рисуется быстрее, чем пишется текст, и связи на ней прослеживаются лучше.

Подводя итог рассмотрению метода и средств структурированного описания архитектуры, еще раз подчеркнем их особую важность и всеобщность. Рассмотренное в связи с описанием архитектуры компьютера структурирование имеет решающее значение как перспективное направление в разработке самой архитектуры, так и в качестве высокоэффективного способа проектирования и реализации возводимой над архитектурой компьютера программной надстройки. Вполне очевидна применимость и полезность структурирования для любой составляющей компьютерной (впрочем, как и всякой другой) системы, однако наибольший эффект даст, конечно, структурирование всех составляющих на единой основе.

Так, структурированное программирование при всех его преимуществах в условиях неструктурированной архитектуры оказывается трудно осуществимым и сопряженным с потерей машинной эффективности программ. Структурированное описание неструктурированной архитектуры оправдано, пожалуй, главным образом как демонстрация того, что дает структурирование, как пример для осознания и усвоения его принципов. Наконец, структурированная архитектура должна быть, разумеется, основой соответствующего стереотипа мышления и системы создания структурированных программ. Только путем внедрения структурирования во все взаимосвязанные звенья аппаратно-программного комплекса, путем полной перестройки на его основе самого характера компьютерной архитектуры и возводимых над ней программных продолжений можно будет реализовать те важные преимущества, которые оно сулит.

ГЛАВА 4
АРХИТЕКТУРА 8-БИТНЫХ МИКРОКОМПЬЮТЕРОВ

Типичными примерами 8-битных микрокомпьютеров могут служить выпускаемые отечественной промышленностью машины СМ 1800, К1-10, К1-20, ВЭФМИКРО. Все они построены на основе микропроцессора К580ИК80, повторяющего архитектуру одного из первых 8-битных микропроцессоров Intel 8080.

Кратко эту архитектуру можно охарактеризовать как одноаккумуляторную с 6-ю 8-битными «блокнотными» регистрами и 16-битным указателем стека. Объем прямо побайтно адресуемой процессором памяти — до 65 536 байтов (64КВ). Адресация регистровая и по одной из пар регистров регистровая косвенная, прямая, непосредственная и стековая. Стековый механизм обращения к подпрограммам и обслуживания прерываний. Возможность непосредственно указывать до 256 однобайтных регистров ввода и до 256 — вывода.

§ 1. Внутренняя организация процессора

Собственную память процессора составляют 7 8-битных регистров, обозначенных буквами A, B, C, D, E, H, L, два 16-битных регистра — программный счетчик PC и указатель стека SP, а также 5 битов-флажков, автоматически фиксирующих характерные признаки результата операции, используемые затем при проверке условий и для других целей.

Регистр A(7:0) является аккумулятором, т. е. участвует в выполнении операций без явного указания на него в команде и обычно выступает в роли операнда, принимающего результат операции (операнда назначения). Любой другой из 8-битных регистров используется только в случае, когда он явно указан в выполняемой команде.

Регистры B, C, D, E, H, L доступны как по отдельности, так и в виде 16-битных соединений в пары BC, DE и HL. Регистр A входит в пару PSW (processor state word — слово состояния процессора), старший байт которой — регистр F — содержит биты-флажки, размещенные в следующем порядке:

7	6	5	4	3	2	1	0
Sb	Zb	0	C'b	0	Pb	1	Cb

Cb — carry bit (бит переноса) фиксирует факт переноса из старшего бита операндов;

Pb — parity bit (бит четности) в состоянии 1, если число единиц в коде результата четно;

C'b — auxiliary carry bit (вспомогательный бит переноса) фиксирует перенос из младшей четверки битов;

Zb — zero bit (бит нуля) в состоянии 1, если результат равен нулю;

Sb — sign bit (бит знака) повторяет значение старшего бита результата.

Пары BC, DE и HL, наряду с 16-битными регистрами PC и SP, рассчитаны главным образом на манипулирование адресами. При этом наибольшими возможностями обладает HL, относительно которой имеется косвенная адресация памяти, команды пересылки 16-битного слова из памяти в HL и из HL в память, из HL в PC и в SP, а также обмена между HL и DE, между HL и верхней парой ячеек стека. Кроме того, HL выступает в роли 16-битного аккумулятора с единственной операцией сложения, причем вторым операндом может быть BC, DE, HL, SP, для которых имеются также операции прибавления и вычитания единицы.

Для пары PSW предусмотрены только пересылки в стек и из стека, существующие для всех пар.

§ 2. Операции преобразования данных

Основным форматом данных в 8-битном микропроцессоре является, естественно, 8-битный байт. Операции преобразования и тестирования данных, выполняемые в аккумуляторе и регистрах, производятся над операндами, длиной в один байт, адресация данных в главной памяти осуществляется побайтно, пересылки данных между аккумулятором и регистрами, в том числе регистрами внешних устройств (ввод/вывод) — это пересылки в формате байта. Таким образом, 8-битный байт является единым базовым форматом для представления и обработки данных на уровне машинных команд. Все другие форматы данных и процедуры для манипулирования ими строятся (программируются) на основе байтов и операций, манипулирующих байтами.

Даже упомянутые выше операции с 16-битными парами регистров реализованы, хотя и на аппаратном уровне, путем последовательной обработки двух байтов, составляющих пару. Дело в том, что микропроцессор К580ИК80 обладает 16-битной адресной магистралью и 8-битной магистралью данных. Поэтому не только пары, но и «неделимые» 16-битные регистры PC и SP, имея возможность передачи в один прием двухбайтного адреса памяти, пересылку и

обработку данных, вынуждены производить в два приема по одному байту.

Унифицированный формат байта вводит норму количества данных, обрабатываемых в один прием, но нисколько не ограничивает многообразия возможных интерпретаций данных. Операции, непосредственно выполняемые микропроцессором над байтами, отражают элементы типичных интерпретаций двоичного слова как набора независимых битов (булевского вектора), двоичного целого числа без знака и со знаком в дополнительном коде, двоично-кодированного десятичного числа. Кроме того, процессор интерпретирует байты как команды или части команд, а устройства ввода/вывода — как коды литер используемого в этих устройствах алфавита.

Наиболее сложной и к тому же запутанной в микропроцессоре К580ИК80 является числовая интерпретация байта. Она основана на использовании двоичного сумматора, перерабатывающего коды двух байтов-слагаемых A(7:0) и B(7:0) в 9-битный код суммы CbS(7:0), где Cb — бит переноса из S(7). Кроме того, имеется вспомогательный бит переноса C'b, запоминающий цифру переноса, передающуюся из младшего полубайта в старший, т. е. из S(3) в S(4).

Как известно, двоичный код на входе и на выходе сумматора можно интерпретировать как естественное представление чисел без знака и как представление чисел со знаком в дополнительном коде. В первом случае 8-битный байт позволяет представить целые числа на интервале от 0 до 255, причем при сложении возникновение переноса из старшего бита, т. е. Cb: = 1, сигнализирует о переполнении — выходе значения суммы за пределы интервала представимых байтом чисел. Во втором случае интервал значений байта модифицирован так, что всем кодам, содержащим в 7-м бите 1, приписаны ближайшие к нулю отрицательные значения, сравнимые по модулю 2^8 с естественными положительными значениями этих кодов. Представимые числа расположены на интервале от —128 до +127, сумматор производит сложение с учетом знаков слагаемых, причем выход значения результата за пределы этого интервала обнаруживается по признаку Cb ≠ S(7).

Однако в микропроцессоре К580ИК80 последний признак действителен не всегда, потому что в случае вычитания биту переноса Cb присваивают не значение переноса из S(7), а его инверсию. При этом в случае чисел без знака ситуация Cb = 1, возникшая в результате сложения, является признаком переполнения, а при вычитании указывает на то, что имеет место заем из S(7), т. е. разность получилась отрицательной. Такая модификация функции Cb в известном смысле упрощает реализацию операций «длинного» сло-

жения и вычитания, т. е. с операндами, представленными не одним, а несколькими байтами. Но при этом утрачивается цельность и автоматизм базирующейся на дополнительном коде арифметики чисел со знаком. Так, значение, принимаемое Cb в результате операции вычитания, отлично от значения Cb, получающегося при нахождении разности тех же чисел путем сложения с дополнением вычитаемого. Программист обязан помнить об этом и при каждом использовании Cb учитывать, в результате какой операции получено его значение.

Вспомогательный бит переноса C'b в отличие от прочих битов-флажков не имеет отношения к условным переходам и вызовам процедур. Он предназначен для десятичной арифметики и используется единственной командой DAA (decimal adjust accumulator), приводящей частичный результат сложения или вычитания двоично-кодированных десятичных чисел в соответствие с требованием, что значение четверки битов, представляющей десятичную цифру, не должно превышать 9. Сумма двух десятичных цифр, представленных четверками битов, может не только превышать 9, но и вызвать перенос, превысив 15. Возникновение этого переноса из младшего полубайта фиксирует C'b, а из старшего полубайта — Cb. Для получения правильной десятичной цифры к значению полубайта, превысившему 9 или породившему перенос, прибавляют 6.

Ради более простой и эффективной реализации арифметических операций с длинными операндами и с десятичными цифрами в наборе команд микропроцессора, наряду с командами обычного сложения и вычитания ADD и SUB, предусмотрены команды ADC и SBB сложения и вычитания с учетом переноса (заема), возникшего при получении предыдущей части результата. При выполнении команды ADC сумма слагаемых A и B вычисляется в виде

X:=Cb; CbS:=A+B+X;

Команда SBB выполняется, соответственно, с учетом заема

X:=Cb; CbS:=A−(B+X); Cb:=¬Cb;

К операциям, связанным с числовой интерпретацией байта, относятся также приращение INR и убавление DCR значения операнда на 1, а также операция сравнения CMP, представляющая собой вычитание без изменения содержимого аккумулятора — обновляются только биты-флажки Cb, C'b, Zb, Sb, Pb.

Все пять флажков устанавливаются при выполнении арифметических команд, включающих, помимо уже названных, команды сложения ADI, ACI, вычитания SUI, SBI и сравнения CPI, в которых вторым операндом служит непосредственно заданная константа (литерал), а также команды POP PSW, восстанавливающей

сохраненное в стеке состояние процессора. При выполнении логических операций конъюнкции (команды ANA и ANI), дизъюнкции (ORA, ORI) и неэквивалентности (XRA, XRI) производится установка Cb, Zb, Sb и Pb, но C'b не затрагивается. Операции циклического сдвига («вращения») RLC, RRC, RAL и RAR, а также операция сложения двубайтных чисел DAD обновляют значение флажка Cb, не затрагивая четырех других. Состояние Cb изменяет, кроме того, команда STC, устанавливающая Cb = 1, и CMC, инвертирующая Cb. Все другие команды, в том числе команды приращения INX и убавления DCX значений двубайтного операнда, пересылок и ввода/вывода, выполняются, не влияя на состояние битов-флажков.

Из операций, трактующих байт как булевский вектор, ANA, ORA и XRA представляют собой побитные соответственно конъюнкцию, дизъюнкцию и неэквивалентность, осуществляемые в аккумуляторе с участием в качестве второго операнда указанного в команде регистра или ячейки памяти. В другой разновидности этих операций — ANI, ORI, XRI — вторым операндом служит непосредственно задаваемая в команде константа. Как было сказано, по результатам данных операций устанавливаются значения всех, за исключением C'b, флажков. Вместе с тем, относящаяся к этой же группе операция побитной инверсии CMA не затрагивает флажки.

Прочие операции над байтом-вектором составляют группу циклического сдвига. Две из них — RLC и RRC — осуществляют циклический сдвиг на один бит влево или вправо в аккумуляторе, а две других — RAL и RAR — такой же сдвиг в CbA, т. е. в аккумуляторе с присоединенным к нему битом Cb. В первом случае выполнению сдвига влево предшествует присваивание Cb:=A(7), а выполнению сдвига вправо — Cb:=A(0). Во втором случае значение Cb устанавливается путем сдвига соответственно из A(7) или из A(0). Состояние других битов-флажков команды циклического сдвига не затрагивают.

§ 3. Средства пересылки данных и управления ходом программы

Возможности пересылки данных между регистрами процессора, регистрами и ячейками памяти, аккумулятором и регистрами внешних устройств представлены значительным числом различных команд, мнемокоды которых вобрали в себя едва ли не все подходящие английские глаголы — move, load, store, push, pop, input, output. Всего имеется 16 мнемокодов, обозначающих пересылки, из них 8 для пересылки одиночных байтов и 8 для пересылки пар.

Команда MOV с указанием двух номеров регистров обеспечивает пересылку между 8-битными регистрами, а также между регистрами и ячейками главной памяти, адресуемыми посредством пары HL. Команда MVI засылает в указанные регистры или ячейки памяти непосредственно задаваемую в ней 8-битную константу. Команды LDA и STA пересылают соответственно из памяти в аккумулятор и из аккумулятора в память по содержащемуся в них 16-битному адресу (прямая адресация). Команды LDAX и STAX вызывают аналогичные пересылки, используя адрес, содержащийся в паре BC или в паре DE. Наконец, команды IN и OUT осуществляют пересылки между аккумулятором и внешним регистром, номер которого задается в них в виде второго байта.

Возможности пересылки парой байтов более ограничены. Команды LHLD и SHLD, содержащие 16-битный адрес памяти, осуществляют пересылку соответственно из двух соседних ячеек в HL и из HL в эти ячейки. Команда LXI обеспечивает засылку непосредственно задаваемой в ней 16-битной константы в любую из пар BC, DE, HL, SP. Команда SPHL пересылает из HL в SP. Команды XCHG и XTHL вызывают обмен значениями соответственно между HL и DE и между HL и парой верхних ячеек стека. Наконец, команда PUSH позволяет заслать в стек значение любой из пар BC, DE, HL, PSW, а команда POP — выполнить обратную процедуру пересылки из стека в указанную пару регистров.

Напомним, что пересылки не оказывают никакого влияния на состояние битов-флажков.

Средства управления процессом выполнения и структурирования программы представлены командами переходов, обращения к подпрограммам и возврата из подпрограмм. Имеется две команды безусловного перехода: PCHL, осуществляющая переход по адресу, содержащемуся в HL, и JMP ADDR — по адресу, содержащемуся в самой команде. Команд условного перехода восемь — по две на каждый из флажков Cb, Sb, Zb, Pb. Например, с флажком Cb связаны команды JC и JNC: первая производит переход при условии Cb = 1, а вторая — при условии Cb = 0. Аналогичными командами перехода по флажку Sb являются JP и JM (jump if positive, jump if minus), по флажку Zb — JZ и JNZ, по флажку Pb — JPE и JPO (jump if parity even, jump if parity odd).

Команды обращения к подпрограммам и возврата из подпрограмм имеются каждая в 9 вариантах: один без условия и восемь условных с теми же условиями, что и в командах перехода. Обращение к подпрограмме по условию и обусловленный возврат из подпрограммы относятся к средствам построения структурированных программ, но не составляют достаточного набора этих средств — нет аппарата циклов.

Команда безусловного обращения к подпрограмме CALL ADDR осуществляет засылку в стек адреса непосредственно следующей за ней команды («адреса возврата») и производит переход на начало подпрограммы по адресу ADDR, который в ней имеется. Команда безусловного возврата из подпрограммы RET пересылает адрес возврата из стека в программный счетчик, вследствие чего происходит переход на продолжение программы, которая обращалась к подпрограмме, с команды, расположенной непосредственно за командой CALL.

В командах обращения к подпрограмме по условию и возврата из подпрограммы по условию обращение и возврат поставлены в зависимость от указанного в каждой такой команде условия. Тестируется состояние соответствующего флажка и, если требующееся условие выполнено, производится обращение или возврат, а если условие не выполнено, то процессор продолжает программу, выполняя дальнейшие команды в линейной последовательности.

Имеется особая команда RST для обращения к подпрограммам, обслуживающим прерывания. Подобно команде CALL она сохраняет в стеке адрес возврата и производит переход на начало указанной в ней подпрограммы. Но адрес этого перехода задан не двумя байтами, как в CALL, а сокращенно, посредством 3-битного поля, значение которого для получения полного адреса умножается на 2^3. Таким образом обеспечивается возможность обращения к восьми подпрограммам, размещенным в первых 64 байтах главной памяти, из которых на каждую подпрограмму отведено 8 байтов.

Обычно команда RST поставляется процессору контроллером прерывающего устройства и выполняется по завершении выполнения текущей команды. При этом не производится обычное в начале других команд приращение программного счетчика, указывающего адрес команды, которую предстояло выполнять в отсутствие прерывания. Именно этот адрес засылается в стек в качестве адреса возврата, так что завершающая подпрограмму команда RET возобновит выполнение прерванной программы с той команды, перед которой произошло прерывание.

К группе управления относится также команда разрешения прерываний EI и команда запрещения прерывания DI, выполнение которой переводит процессор в состояние невосприимчивости к запросам устройств, добивающихся прерывания. Это состояние устанавливается также автоматически при каждом переходе в прерывание, и чтобы сделать возможными последующие прерывания, должна быть выполнена команда EI.

Еще две команды, которые можно отнести в разряд управляющих,— это не вызывающая никаких действий NOP (no operation) и останавливающая процессор HLT (halt).

§ 4. Форматы команд

Набор команд микропроцессора К580ИК80 включает 78 различных (различающихся мнемокодами операций) команд. Из них 44 однобайтных, 11 двубайтных и 23 трехбайтных. Второй байт в двубайтных командах используется либо для непосредственного задания значения одного из операндов, либо представляет собой номер внешнего устройства в команде ввода или вывода. В трехбайтных командах второй и третий байты составляют либо адрес операнда, либо непосредственно заданное значение операнда двойной длины. При этом второй байт содержит младшую часть адреса или значения, а третий байт — старшую часть.

Первый байт всякой команды указывает выполняемую операцию и обычно также операнд, а иногда и два операнда. Указателями операндов являются особо выделенные поля, длиной в 1—3 бита, содержимое которых интерпретируется как номер регистра или пары регистров или как код более сложной процедуры доступа к значению операнда.

Так, в командах, относящихся к данным одинарной длины (длиной в 1 байт), поле К(5:3), а при двух операндах и К(2:0), содержит трехбитный код Nd или Ns, значениям которого 0, 1, 2, 3, 4, 5 и 7 сопоставлены регистры B, C, D, E, H, L и A. Значению же 6 соответствует косвенная регистровая адресация по паре регистров HL, т. е. операндом является ячейка памяти m(HL). При записи команд на языке ассемблера номера регистров заменяются буквами, обозначающими эти регистры, а вместо номера 6 пишут букву M. Таким образом, буквы B, C, D, E, H, L, M, A, с одной стороны, служат мнемокодами значений, принимаемых номером Nd или Ns, а с другой стороны, являются (за исключением M) именами регистров RNd, RNs, соответствующих этим значениям.

Указателем операнда двойной длины служит двубитное К(5 : 4) или однобитное К (4) поле, код которого в зависимости от вида команды интерпретируется или как номер Np, относящийся к парам регистров BC, DE, HL и к 16-битному регистру SP, или как номер Nw, относящийся к парам BC, DE, HL и PSW, или как однобитный номер N, относящийся к парам BC и DE. Мнемокодами значений, принимаемых этими номерами, служат буквы B, D, H и буквосочетания SP, PSW.

Соответствие кодов и мнемокодов обозначенным ими объектам отражено в следующих таблицах.

Особым образом используется поле К(5:3) в команде RST. Его содержимое, рассматриваемое как целое число Na, $0 \leqslant Na \leqslant 7$, употребляется для вычисления адреса перехода на подпрограмму по формуле ADDR:=Na·2^3.

Nd, Ns	000	001	010	011	100	101	110	111
Мнемокод RNd, RNs	B B	C C	D D	E E	H H	L L	M m(HL)	A A

Np	00	01	10	11
Мнемокод RNp	B BC	D DE	H HL	SP SP

Nw	00	01	10	11
Мнемокод RNw	B BC	D DE	H HL	PSW FA

N	0	1
Мнемокод RN	B BC	D DE

За исключением случаев использования рассмотренных полей для указания операндов, все биты первого байта команды составляют код операции. Но большинство команд объединено в группы с фиксированными значениями 5—6 битов и различаются внутри группы по 3 битам поля К(5:3) или по 2 битам поля К(4:3). Эти биты составляют в пределах группы g код операции Fg, значениям которого сопоставлены мнемокоды операций в языке ассемблера согласно следующим таблицам.

К(5:3)	000	001	010	011	100	101	110	111
Fa	ADD	ADC	SUB	SBB	ANA	XRA	ORA	CMP
Fi	ADI	ACI	SUI	SBI	ANI	XRI	ORI	CPI
Fj	JNZ	JZ	JNC	JC	JPO	JPE	JP	JM
Fc	CNZ	CZ	CNC	CC	CPO	CPE	CP	CM
Fr	RNZ	RZ	RNC	RC	RPO	RPE	RP	RM

К(4:3)	00	01	10	11
Fs	RLC	RRC	RAL	RAR
Fm	DAA	CMA	STC	CMC
Fd	SHLD	LHLD	STA	LDA

С использованием обозначений содержимого полей К(5:3), К(5:4), К(4), К(4:3), К(2:0), значение которых применительно к конкретным командам или группам команд определено в приведенных таблицах, форматы команд микропроцессора К580ИК можно описать следующим образом.

Однобайтные команды

Двоичный код								Мнемокоды операций
7	6	5	4	3	2	1	0	— номера битов
0	0	0	0	0	0	0	0	NOP
0	0	Ns			1	0	0	INR
0	0	Ns			1	0	1	DCR
0	1	Nd			Ns			MOV
0	1	1	1	0	1	1	0	HLT
0	0	Np		0	0	1	1	INX
0	0	Np		1	0	1	1	DCX
0	0	Np		1	0	0	1	DAD
0	0	0	N	0	0	1	0	STAX
0	0	0	N	1	0	1	0	LDAX
0	0	0	Fs		1	1	1	RLC, RRC, RAL, RAR
0	0	1	Fm		1	1	1	DAA, CMA, STC, CMC
1	0	Fa			Ns			ADD, ADC, SUB, ..., CMP
1	1	Nw		0	1	0	1	PUSH
1	1	Nw		0	0	0	1	POP
1	1	0	0	1	0	0	1	RET
1	1	Fr			0	0	0	RNZ, RZ, RNC, ..., RM
1	1	Na			1	1	1	RST
1	1	1	0	1	0	0	1	PCHL
1	1	1	0	1	0	1	1	XCHG
1	1	1	0	0	0	1	1	XTHL
1	1	1	1	1	0	0	1	SPHL
1	1	1	1	0	0	1	1	DI
1	1	1	1	1	0	1	1	EI

Двубайтные команды

Первый байт								Второй байт	
0	0	ND		1	1	0		DATA	MVI
1	1	Fi		1	1	0		DATA	ADI, ACI, SUI, ..., CPI
1	1	0	1	1	0	1	1	DN	IN
1	1	0	1	0	0	1	1	DN	OUT

DN — номер внешнего устройства

Трехбайтные команды

Первый байт								Второй байт	Третий байт	
0	0	Np		0	0	0	1	DATA мл	DATA ст	LXI
0	0	0	Fd		0	1	0	ADDR мл	ADDR ст	SHLD, LHLD, STA, LDA
1	1	0	0	0	0	1	1	ADDR мл	ADDR ст	JMP
1	1	Fj			0	1	1	ADDR мл	ADDR ст	JNZ, JZ, ..., JM
1	1	0	0	1	1	0	1	ADDR мл	ADDR ст	CALL
1	1	Fc			1	0	0	ADDR мл	ADDR ст	CNZ, CZ, ..., CM

Мнемоническая запись команды на языке ассемблера состоит из мнемокода операции, которому может предшествовать метка, и следующих за ним, если это необходимо, одного или двух операндов. В команде, относящейся к регистрам или парам регистров, соответствующие операнды могут быть представлены номерами регистров или их мнемокодами. Операнды, помещаемые во второй или во второй и третий байты команды, задаются либо непосредственно в виде чисел, либо в виде выражений, в результате выполнения которых получаются эти числа. В выражения могут входить метки, числовые и литерные константы, а также текущее значение программного счетчика, обозначаемое литерой $. Более подробные сведения об этом приведены в описании языка ассемблера.

§ 5. Алгоритмы выполнения команд

Необходимое системному программисту и разработчику микрокомпьютерных систем четкое и однозначное описание архитектуры микропроцессора К580ИК80 включает представление регистров процессора и адресуемой памяти соответствующими структурами данных и запись на подходящем алгоритмическом языке алгоритмов выполнения команд.

Регистры процессора A, B, C, D, E, H, L будем рассматривать как массивы битов вида RN(7:0), а пары этих регистров — BC, DE, HL — соответственно, как конкатенации вида RI(7:0)RJ(7 : 0) и вместе с тем как 16-битные массивы битов вида RP(15 : 0). Программный счетчик PC и указатель стека SP имеют аналогичное представление: PC (15:0), SP(15:0). В качестве арифметических операндов массивы битов интерпретируются как двоичные числа.

Биты-флажки Cb, C'b, Sb, Zb, Pb и триггер разрешения прерываний INTE рассматриваются как одноэлементные массивы битов вида R(1:1) и, вместе с тем, как двузначные переменные, прини-

мающие значения 0 и 1. Пара PSW имеет структуру FA, где регистр F есть

Sb Zb 0 C'b 0 Pb 1 Cd

Главная память представлена массивом 8-битных байтов m(0:65535), совокупности портов ввода и вывода — также массивами 8-битных байтов Pin(0:255) и Pout(0:255). Второй байт двубайтной команды представляет либо операнд DATA(7 : 0), либо номер порта ввода или вывода DN, $0 \leqslant DN \leqslant 255$. Второй и третий байты трехбайтной команды составляют либо 16-битный адрес ADDR, либо операнд двойной длины DATAмлDATAст.

Используемые в описании команд номера регистров Ns, Nd, номера пар Np, Nw, N и номера портов ввода/вывода DN представляют собой целые числа с интервалами значений $0 \leqslant (Ns,\ Nd) \leqslant 7$, $0 \leqslant (Np,\ Nw) \leqslant 3$, $0 \leqslant N \leqslant 1$, $0 \leqslant DN \leqslant 255$.

Наряду с массивами, представляющими регистры процессора, при описании команд использованы вспомогательные массивы битов X (7:0) и Y (15:0). Ради компактности и обозримости представления алгоритмов введены вспомогательные процедуры ASSX, ADDX, SUBX, SZP, INPC, YADDR, CALLY, RETY.

Процедура ASSX присваивает X значение указанного в команде байта:

if Ns=6 **then** X:=m(HL) **else** X:=RNs;

Процедура ADDX прибавляет X к аккумулятору A:

C'bA(3:0):=A(3:0)+X(3:0);
CbA(7:4):=A(7:4)+X(7:4)+C'b;

Процедура SUBX вычитает X из аккумулятора A:

for i:=0 **to** 7 **do** X(i):=¬X(i); X:=X+1;
C'bA(3:0):=A(3:0)+X(3:0);
CbA(7 : 4):=A(7:4)+X(7:4)+C'b; Cb:=¬Cb;

Процедура SZP присваивает значения битам-флажкам Sb, Zb и Pb:

Sb:=A(7); **if** A=0 **then** Zb:=1 **else** Zb:=0;
Pb:=1; **for** i:=0 **to** 7 **do** Pb:= Pb ⊕ A(i);

Процедура INPC увеличивает значение программного счетчика PC на 1:

PC:=PC+1;

Процедура YADDR присваивает Y значение адреса, представленное 2-м и 3-м байтами команды:

Y(7:0):=m(PC); INPC; Y(15:8);=m(PC); INPC;

Процедура CALLY осуществляет обращение к подпрограмме по адресу Y и засылку в стек адреса возврата:

YADDR; SP:=SP—1; m(SP):=PC(15:8); SP:=SP—1;
m(SP):=PC(7:0); PC:=Y;

Процедура RETY осуществляет возврат из подпрограммы на извлеченный из стека адрес:

Y(7:0):=m(SP); SP:=SP+1; Y(15:8):=m(SP);
SP:=SP+1; PC:=Y;

Команды преобразования данных.

ADD Ns — прибавить указанный в команде операнд к аккумулятору:

INPC; ASSX; ADDX; SPZ;

ADC Ns — прибавить Cb и указанный в команде операнд к аккумулятору:

INPC; ASSX; C'bA(3:0):=A(3:0)+X(3:0)+Cb,
CbA(7:4):=A(7:4)+X(7:4)+C'b; SZP;

SUB Ns — вычесть указанный в команде операнд из аккумулятора:

INPC; ASSX; SUBX; SZP;

SBB Ns — вычесть указанный операнд и Cb из аккумулятора:

INPC; ASSX; X:=X+Cb; SUBX; SZP;

ANA Ns — образовать в аккумуляторе побитную конъюнкцию с указанным операндом:

INPC; ASSX; **for** i:=0 **to** 7 **do** A(i):=A(i)⋀X(i); Cb:=0; SZP;

XRA Ns — образовать в аккумуляторе побитную неэквивалентность с указанным операндом:

INPC; ASSX; **for** i:=0 **to** 7 **do** A(i):=A(i)⊕X(i); Cb:=0; SZP;

ORA Ns — образовать в аккумуляторе побитную дизъюнкцию с указанным операндом:

INPC; ASSX; **for** i := 0 **to** 7 **do** A(i) := A(i)⋁X(i); Cb := 0; SZP;

CMP Ns — сравнить аккумулятор с указанным операндом:

INPC; ASSX; Y(7 : 0) := A; SUBX; SZP; A := Y(7 : 0);

ADI DATA — прибавить байт-литерал DATA к аккумулятору:

INPC; X := m(PC); INPC; ADDX; SZP;

ACI DATA — прибавить байт-литерал DATA и Cb к аккумулятору:

INPC; X := Cb; ADDX; X := m(PC); INPC; ADDX; SZP;

SUI DATA — вычесть байт-литерал DATA из аккумулятора:

INPC; X := m(PC); INPC; SUBX; SZP;

SBI DATA — вычесть из аккумулятора байт-литерал DATA и Cb:

INPC; X := m(PC); INPC; X := X+Cb; SUBX; SZP;

ANI DATA — образовать в аккумуляторе побитную конъюнкцию с байтом-литералом DATA:

INPC; X := m(PC); INPC; **for** i:=0 **to** 7 **do** A(i):=A(i)∧X(i);

Cb:=0; SZP;

XRI DATA — образовать в аккумуляторе побитную неэквивалентность с байтом-литералом DATA:

INPC; X := m(PC); INPC; **for** i := 0 **to** 7 **do** A(i) := A(i) ⊕ X(i);

Cb := 0; SZP;

ORI DATA — образовать в аккумуляторе побитную дизъюнкцию с байтом-литералом DATA:

INPC; X:=m(PC); INPC; **for** i:=0 **to** 7 **do** A(i):=A(i)∨X(i);

Cb:=0; SZP;

CMI DATA — сравнить аккумулятор с байтом-литералом DATA:

INPC; X:=m(PC); INPC; Y(7:0):=A; SUBX; SZP; A := Y(7:0);

INR Ns — увеличить значение указанного операнда на 1:

INPC; Y(7:0):=A; A:=1; ASSX; C'bA(3:0):=A(3:0)+X(3:0); A(7:4):=A(7:4)+X(7:4)+C'b; SZP; **if** Ns=6 **then** m(HL):=A **else** RNs:=A; A:=Y(7:0);

DCR Ns — уменьшить значение указанного операнда на 1:

INPC; Y(7:0):=A; **if** Ns=6 **then** A:=m(HL) **else** A:=RNs; X:=1; **for** i:=0 **to** 7 **do** X(i):=¬X(i); X:=X+1; C'bA(3:0):=A(3:0)+X(3:0); A(7:4):=A(7:4)+X(7:4)+C'b; SZP; **if** Ns=6 **then** m(HL):=A **else** RNs:=A; A:=Y(7:0);

DAD Np — прибавить указанную пару к HL:

INPC; **if** Np=0 **then** Y:=BC **else if** Np = 1 **then** Y:=DE **else if** Np=2 **then** Y:=HL **else** Y:=SP; CbL:=L+Y(7:0); CbH:=H+Y(15:8) + Cb;

INX Np — увеличить значение указанной пары на 1:

INPC; **if** Np=0 **then** BC:=BC+1 **else if** Np=1 **then** DE:=DE+1 **else if** Np=2 **then** HL:=HL+1 **else** SP:=SP+1;

DCX Np — уменьшить значение указанной пары на 1:

INPC; **for** i:=0 **to** 15 **do** Y(i):=1; **if** Np=0 **then** BC:=BC+Y **else if** Np=1 **then** DE:=DE+Y **else if** Np=2 **then** HL:=HL+Y **else** SP:=SP+Y;

RLC — циклический сдвиг аккумулятора влево:

INPC; Cb:=A(7); A:=A(6:0)A(7);

RRC — циклический сдвиг аккумулятора вправо:

INPC; Cb:=A(0); A:=A(0)A(7:1);

RAL — циклический сдвиг аккумулятора с Cb влево:

INPC; CbA:=A(7:0)Cb;

RAR — циклический сдвиг аккумулятора с Cb вправо:

INPC; A(7:0)Cb:=CbA;

DAA — десятичная коррекция аккумулятора:

INPC; X:=6; **if** A(3:0) > 9 ∨ C'b=1 **then** ADDX; **if** A(7:4)>9∨Cb=1 **then begin** CbA(7:4):=A(7:4)+6 **end**; SZP;

CMA — побитно инвертировать аккумулятор:

INPC; **for** i:=0 **to** 7 **do** A(i):=¬A(i);

CMC — инвертировать флажок Cb:

INPC; Cb:=Cb;

STC — установить Cb в состояние 1:

INPC; Cb:=1;

Команды пересылки байта.

MOV Nd Ns — присвоить операнду назначения значение операнда отправления:

INPC; **if** Nd≠6∧Ns≠6 **then** RNd:=RNs **else if** Nd≠6∧Ns=6 **then** RNd:=m(HL) **else if** Nd=6∧Ns≠6 **then** m(HL):=RNs **else** HLT;

MVI Nd DATA — присвоить операнду назначения значение байта-литерала DATA:

INPC; **if** Nd=6 **then** m(HL):=m(PC) **else** RNd:=m(PC); INPC;

LDAX N — присвоить аккумулятору значение байта памяти, адрес которого задан парой BC или парой DE:

INPC; **if** N=0 **then** A:=m(BC) **else** A:=m(DE);

STAX N — присвоить значение аккумулятора байту памяти, адрес которого задан парой BC или парой DE:

INPC; **if** N=0 **then** m(BC):=A **else** m(DE):=A;

LDA ADDR — присвоить аккумулятору значение байта памяти, адрес которого ADDR задан в команде:

INPC; YADDR; A:=m(Y);

STA ADDR — присвоить значение аккумулятора байту памяти, адрес которого ADDR задан в команде:

INPC; YADDR; m(Y):=A;

IN DN — присвоить аккумулятору значение регистра ввода, номер которого DN задан в команде:

INPC; X:=m(PC); INPC; A:=Pin(X);

OUT DN — присвоить значение аккумулятора регистру вывода, номер которого DN задан в команде:

INPC; X:=m(PC); INPC; Pout(X):=A;

Команды пересылки пары байтов.

LHLD ADDR — присвоить паре HL значение двух байтов памяти, адрес которых ADDR задан в команде:

INPC; YADDR; L:=m(Y); H:=m(Y+1);

SHLD ADDR — присвоить значение пары HL двум байтам памяти, адрес которых ADDR задан в команде:

INPC; YADDR; m(Y):=L; m(Y+1):=H;

SPHL — присвоить указателю стека значение HL:

INPC; SP:=HL;

LXI DATA — присвоить указанной паре значение двубайтного литерала:

INPC; YADDR; **if** Np=0 **then** BC:=Y **else if** Np=1 **then** DE:=Y **else if** Np=2 **then** HL:=Y **else** SP:=Y;

XCHG — обменять значения DE и HL:

INPC; DE:=:HL;

XTHL — обменять значения HL и верхней пары позиций стека:

INPC; L:=:m (SP); H:=:m (SP + 1);

PUSH — заслать в стек значение указанной пары:

INPC; **if** Nw = 0 **then** Y:=BC **else if** Nw=1 **then** Y:=DE **else if** Nw=2 **then** Y:=HL **else** Y:= PSWA; SP:= SP—1;
m (SP):=Y (15 : 8); SP:= SP — 1; m (SP):= Y (7:0);

POP — значение двух верхних байтов стека заслать в указанную пару:

INPC; Y (7 : 0):= m (SP); SP:= SP + 1; Y (15 : 8):= m (SP); SP:= SP + 1;
if Nw = 0 **then** BC:= Y **else if** Nw = 1 **then** DE:= Y **else if** Nw = 2 **then** HL:= Y **else** PSWA:= Y;

Команды управления.

PCHL — перейти по адресу, который содержится в HL:

PC:= HL;

JMP ADDR — перейти по адресу, заданному во 2-м и 3-м байтах команды:

INPC; YADDR; PC:=Y;

JC ADDR — перейти по ADDR, если бит переноса в состоянии 1:

INPC; YADDR; **if** Cb=1 **then** PC:=Y;

JNC ADDR — перейти по ADDR, если бит переноса в состоянии 0:

INPC; YADDR; **if** Cb:=0 **then** PC:=Y;

JM ADDR — перейти по ADDR, если знак результата минус:

INPC; YADDR; **if** Sb=1 **then** PC:=Y;

JP ADDR — перейти по ADDR, если результат не отрицателен:

INPC; YADDR; **if** Sb=0 **then** PC:=Y;

JZ ADDR — перейти по ADDR, если результат равен нулю:

INPC; YADDR; **if** Zb=1 **then** PC:=Y;

JNZ ADDR — перейти по ADDR, если результат не равен нулю:

INPC; YADDR; **if** Zb=0 **then** PC:=Y;

JPE ADDR — перейти по ADDR, если число единиц в байте четно:

INPC; YADDR; **if** Pb=1 **then** PC:=Y;

JPO ADDR — перейти по ADDR, если число единиц в байте нечетно:

INPC; YADDR; if Pb=0 then PC:=Y;

CALL ADDR — обратиться к подпрограмме по адресу ADDR, содержащемуся во 2-м и 3-м байтах команды:

INPC; YADDR; CALLY;

CC ADDR — обратиться к подпрограмме ADDR, если бит переноса в состоянии 1:

INPC; YADDR; if Cb=1 then CALLY;

CNC ADDR — обратиться к подпрограмме ADDR, если бит переноса в состоянии 0:

INPC; YADDR; if Cb=0 then CALLY;

CM ADDR — обратиться к подпрограмме ADDR, если знак результата минус:

INPC; YADDR; if Sb=1 then CALLY;

CP ADDR — обратиться к подпрограмме ADDR, если результат не отрицателен:

INPC; YADDR; if Sb=0 then CALLY;

CZ ADDR — обратиться к подпрограмме ADDR, если результат равен нулю:

INPC; YADDR; if Zb = 1 then CALLY;

CNZ ADDR — обратиться к подпрограмме ADDR, если результат не равен нулю:

INPC; YADDR; if Zb = 0 then CALLY;

JPE ADDR — обратиться к подпрограмме ADDR, если число единиц в байте четно:

INPC; YADDR; if Pb = 1 then CALLY;

JPO ADDR — обратиться к подпрограмме ADDR, если число единиц в байте нечетно:

INPC; YADDR; if Pb = 0 then CALLY;

RET — возвратиться из подпрограммы:

INPC; RETY;

RC — возвратиться из подпрограммы, если бит переноса в состоянии 1:

INPC; if Cb = 1 then RETY;

RNC — возвратиться из подпрограммы, если бит переноса в состоянии 0:

INPC; if Cb = 0 then RETY;

RM — возвратиться из подпрограммы, если знак результата минус:

INPC; if Sb = 1 then RETY;

RP — возвратиться из подпрограммы, если результат не отрицателен:

INPC; if Sb = 0 then RETY;

RZ — возвратиться из подпрограммы, если результат равен нулю:

INPC; if Zb = 1 then RETY;

RNZ — возвратиться из подпрограммы, если результат не равен нулю:

INPC; if Zb = 0 then RETY;

RPE — возвратиться из подпрограммы, если число единиц в байте четно:

INPC; if Pb = 1 then RETY:

RPO — возвратиться из подпрограммы, если число единиц в байте нечетно:

INPC; if Pb = 0 then RETY;

RST — обратиться к подпрограмме, вычислив ее адрес по заданному N:

Y:= N * 8; CALLY;

EI — разрешить прерывания:

INPC; INTE:= 1;

DI — запретить прерывания:

INPC; INTE:= 0;

NOP — не производя операции, перейти к следующей команде:

INPC;

HLT — остановить процессор:

INPC; Stop;

ГЛАВА 5

АРХИТЕКТУРА 16-БИТНЫХ МИКРОКОМПЬЮТЕРОВ

Выпускаемые отечественной промышленностью в больших количествах 16-битные микрокомпьютеры серии «Электроника 60», «Электроника НЦ-80» и др. являются микрокомпьютерами так называемой унифицированной архитектуры. Это архитектура миникомпьютеров СМ-3 и СМ-4, а также 16-битных компьютеров высокой производительности «Электроника 79». Первоисточником ее является семейство PDP-11 американской фирмы Digital Equipment Corporation (DEC), первый член которого — миникомпьютер PDP-11/20 — появился в 1970 г. Семейство DEC PDP-11 было задумано и осуществлено как ряд снизу вверх совместимых процессоров с широким ассортиментом периферийного оборудования и набором операционных систем различных возможностей и назначения. Нижняя часть этого ряда представлена микрокомпьютерами, которые в отношении архитектуры отличаются от старших моделей главным образом меньшим объемом адресного пространства (как правило, 64К) и минимальным набором команд.

§ 1. Общая характеристика унифицированной архитектуры

Унифицированную архитектуру в самом первом приближении можно охарактеризовать как 16-битную, с многорегистровым процессором, обеспечивающим возможность как пословного, так и побайтного (пополусловного) доступа к памяти и манипулирования данными, с прямой и косвенной регистровой, автоинкрементной, автодекрементной и индексной адресацией в едином для главной памяти и периферийных регистров адресном пространстве, с векторной схемой прерывания и автоматическим запоминанием слова состояния процессора и адреса возврата из подпрограммы в стеке.

То, что архитектура является 16-битной, означает, что переработка данных процессором осуществляется в виде операций с 16-битными операндами, а точнее — длина операндов, непосредственно обрабатываемых процессором, не превышает 16 битов. Процессор рассчитан на обработку 16-битных слов: регистры его 16-битные, устройство преобразования и тестирования данных выполняет операции с 16-битными операндами в один прием, ширина одноразовой выборки из главной памяти также равна 16 битам. Работа с данными больших длин на таком процессоре реализуется при помощи операций с составными операндами удвоенной, утроенной и другой многократной длины. Вместе с тем, 16-битная архитектура не исключает возможности обработки 8-битных байтов, рассматриваемых как полуслова. Почти все операции преобразования и тестирования данных на микрокомпьютерах унифицированной архитектуры имеются в двух вариантах — с операндами-словами и с операндами-байтами.

Многорегистровый процессор, в отличие от аккумуляторного и стекового, построен на основе набора пронумерованных регистров, называемых регистрами общего назначения или кратко — общими регистрами. Каждый из этих регистров, обозначаемый в команде его номером, может использоваться как для хранения значений операндов, так и в качестве указателя, содержащего адрес местонахождения операнда в главной памяти, причем имеется возможность индексирования и автоиндексирования указателя, а также прямого и косвенного обращения. Таким образом, регистры, с одной стороны, представляют собой быстродействующую и экономно адресуемую память процессора как преобразователя данных, а с другой стороны, обеспечивают эффективную адресацию главной памяти.

Микрокомпьютеры унифицированной архитектуры имеют по восемь общих регистров, в совокупности образующих массив битов R(0:7, 15:0). В коде команды регистр Rn представлен своим трехбитным номером n = 0, 1, . . ., 7. Ассемблер воспринимает в качестве номера регистра n сочетание %n, поэтому применяя в тексте программы обозначения R0, R1, ..., необходимо объявить эквивалентность: R0 = %0, R1 = %1 и т. д. Регистр R7, помимо обычных функций общего регистра, выполняет функции программного счетчика, поэтому для него принято также обозначение PC (Program Counter). Соответственно, регистр R6, выступающий в роли указателя стека, обозначают SP (Stack Pointer).

Адресное пространство микрокомпьютера унифицированной архитектуры (без расширителя памяти) определяется совокупностью значений, принимаемых 16-битным адресом, т. е. (0:65535) или 64К. Память рассматривается как массив m(0:65535, 7:0) с побайтной адресацией и вместе с тем как массив 16-битных слов, адресуемых четными значениями адреса. Адрес слова W является также адресом младшего байта этого слова W(7:0), адрес же старшего байта W(15:8) на 1 больше. Попытка обратиться к слову по нечетному адресу расценивается процессором как ошибка. Для представления команд и адресов используются только полные слова, поэтому значения счетчика команд и указателя стека всегда четны, а последовательное приращение и убавление их производится прибавлением и вычитанием 2.

Доступ процессора к периферийным регистрам обеспечивается путем включения этих регистров в адресное пространство наравне

с ячейками главной памяти. Для адресации регистров выделены старшие 8К адресов, позволяющие идентифицировать 4К слов. Вследствие такого «отображения» периферийных объектов в адресное пространство, на них распространяются все операции, выполняемые процессором над ячейками главной памяти, т. е. открывается возможность эффективно манипулировать периферией, не предусматривая для этого никаких специальных команд. Технически объединенное адресное пространство осуществлено на основе единой магистрали, посредством которой взаимодействуют друг с другом все подключенные к ней компоненты микрокомпьютера: процессор, главная память, контроллеры внешних устройств. Взаимодействие осуществляется путем адресуемой передачи данных. Все доступные для обмена данными ячейки памяти и регистры подключенных к магистрали устройств обладают адресами. Для доступа к требуемой ячейке (регистру) на магистраль выдается ее адрес и команда, задающая вид обмена: считать/записать слово/байт. В случае считывания на магистрали появляется копия содержимого опрашиваемой ячейки, а в случае записи ячейка принимает установленное на магистрали значение слова или байта.

Магистраль реализована в виде 38 параллельных линий, из которых 16 служат для передачи поочередно адресов и данных, а остальные используются для сигнализации и управления. В каждый момент по магистрали может производиться не более одного взаимодействия, происходящего между двумя устройствами. Одно из этих устройств выступает в активной роли запросчика, задающего адрес и вид взаимодействия, другое устройство является ответчиком, послушно выполняющим требование запросчика. Запросчиком обычно является процессор, но им может быть также, например, контроллер периферийного устройства с правом прямого доступа к памяти или прерывания процессора. Как прямой доступ, так и прерывание связаны с захватом магистрали инициирующим их устройством-запросчиком. В случае прямого доступа магистраль предоставляется запросчику по завершении взаимодействия, происходящего в момент поступления запроса. В случае прерывания предоставление магистрали запрашивающему устройству возможно только перед выборкой очередной команды (т. е. по завершении выполняемой команды) при условии, что прерывание не запрещено ни по состоянию процессора, ни по состоянию самого запрашивающего устройства, и что данное устройство наделено наивысшим приоритетом из всех добивающихся прерывания устройств.

Захватив магистраль, прерывающее устройство передает процессору адрес своего вектора прерывания — пары ячеек главной памяти, содержащих начальный адрес программы обработки прерывания и установленное для этой программы значение слова состояния процессора. Процессор засылает текущие значения программного счетчика и слова состояния в стек, заменяя их значениями соответствующих компонент вектора прерывания, вследствие чего начинается выполнение программы обработки прерывания. Эта программа заканчивается командой обратной пересылки из стека в программный счетчик и регистр состояния процессора тех значений, которыми они обладали в момент возникновения прерывания. Таким образом процессор переключается на продолжение прерванной программы.

Если во время обработки прерывания значение состояния процессора таково, что прерывания не запрещены, то возможно возникновение нового прерывания («прерывания в прерывании»). Оно осуществляется точно так же, как и предшествующее: процессор упрятывает в стек текущие значения программного счетчика и слова состояния, используя сообщенный прерывающим устройством адрес вектора прерывания, засылает в программный счетчик и регистр состояния начальный адрес и слово состояния для программы, обслуживающей новое прерывание. Программа, обрабатывавшая первое прерывание откладывается, уступая процессор более срочной программе обработки второго прерывания. Когда же последняя будет выполнена, завершающая ее команда возврата из прерывания вызовет пересылку из стека в программный счетчик и регистр состояния процессора упрятанных при втором прерывании их значений для продолжения программы обработки первого прерывания. Когда же и эта программа будет выполнена, процессор возвратит из стека в программный счетчик и регистр состояния их значения, упрятанные при первом прерывании, и продолжит выполнение первоначально прерванной программы.

Векторный механизм прерывания с запоминанием текущего состояния процессора в стеке, как видно, действует автоматически, обеспечивая возможность многоуровнего гнездования прерываний. Все, что требуется от программиста, использующего данный механизм,— это сопоставить прерывающим устройствам значения векторов, отсылающие на начала программ обработки прерываний и определяющие слово состояния процессора, а в конце каждой из этих программ поставить команду возврата из прерывания. Заметим, что тот же стек аналогичным образом используется при вызовах и гнездовании подпрограмм, причем обращения к подпрограммам могут чередоваться с прерываниями в любых сочетаниях — стек, действующий по правилу «последним вошел — первым вышел», пунктуально отслеживает последовательность упрятываний и возвратов, а кроме того, может употребляться для передачи параметров подпрограммам и для динамического распределения памяти под локальные переменные.

§ 2. Операции, выполняемые процессором

Набор операций, выполняемых процессором унифицированной архитектуры, типичен для миникомпьютеров — в него входят элементарные операции преобразования данных, производимые над 16-битными словами и над байтами, операции пересылки, сравнения и тестирования слов и байтов, большая группа условных переходов с разными условиями, обращение к подпрограмме и ряд операций управления работой процессора. Собственно арифметических операций в основном наборе только две — сложение и вычитание 16-битных слов, интерпретируемых как целые числа. Операции умножения и деления, а также операции над числами с плавающей запятой реализуются программно или добавлением к базовому процессору устройств расширенной и «плавающей» арифметики.

Операции пересылки, преобразования и тестирования данных сочетаются с богатым выбором способов адресации операндов, обеспечивающем выполнение их в режимах регистр—регистр, регистр—память, память—память, а также с непосредственным заданием значений операндов, с косвенной, индексной и автоиндексной адресацией.

При выполнении операций устанавливаются значения битов N, Z, V, C слова состояния процессора, характеризующие результат операции: N = 1 — результат отрицателен (старший бит слова или байта равен 1), Z = 1 — результат равен нулю, V = 1 — арифметическое в дополнительном коде переполнение, C = 1 — перенос 1 в находящийся вне пределов слова/байта n-й бит. Значения данных битов используются при выполнении условных переходов. Имеются операции присваивания каждому из этих битов значений 0 и 1.

Различаются одноопе рандные и двухоперандные операции и соответственно команды. К одноопе рандным относятся: CLR (clear — очистить, присвоить 0), COM (complement — побитная инверсия), INC (increment — прирастить, прибавить 1), DEC (decrement — убавить, вычесть 1), NEG (negate — изменить знак числа на противоположный), TST (test — тестировать, выработать значения битов N, Z, V, C), ADC (add carry — прибавить C), SBC (subtract carry — вычесть C), SXT (sign extend — произвести «расширение» знака), MTPS (move byte to PSW — переслать байт в слово состояния процессора), MFPS (move byte from PSW — переслать байт из слова состояния процессора), ROR (rotate right — вращать вправо, циклический сдвиг вправо вместе с битом C), ROL (rotate left — вращать влево, циклический сдвиг влево вместе с битом C), ASR (arithmetic shift right — арифметический сдвиг вправо), ASL (arithmetic shift left — арифметический сдвиг влево), SWAB (swap bytes — переставить байты слова).

Двухоперандными операциями базового процессора являются: MOV (move — переслать), CMP (compare — сравнить, выработать значения битов N, Z, V, C по разности), BIT (bit test — выработать значения битов N, Z, V, C по конъюнкции), BIC (bit clear — очистить биты, конъюнкция с побитно инвертированным), BIS (bit set — установить биты, дизъюнкция), ADD (add — сложить), SUB (subtract — вычесть). Перечисленные операции, за исключением SXT, SWAB, ADD и SUB, имеются каждая в двух вариантах: над словами и над байтами. К мнемокоду операции, относящейся к байтам, добавляется B, например: CLRB, MOVB, BICB. Операндами для SXT, SWAB, ADD и SUB могут быть только слова.

Применительно к 16-битному регистру операции над байтами выполняются над 8-ю младшими битами R(7:0) регистра, причем значение старшего байта R(15:8) сохраняется неизменным. Исключением является операция MOVB: при пересылке байта в регистр значение этого байта присваивается младшей половине R(7:0) регистра, а биты старшей половины R(15:8) регистра принимают значение бита знака R(7), т. е. производится так называемое расширение знака — значение слова в целом оказывается равным значению пересылаемого байта с учетом его знака.

Операции переходов называются branches — ветвления, ответвления, например: BR (branch unconditional — ответвиться безусловно, т. е. безусловный переход), BPL (branch if plus — ответвиться, если плюс, т. е. если N = 0), BCS (branch if carry is set — ответвиться, если C = 1) и т. д. Для операций branch характерно задание адреса перехода в виде смещения (со знаком) относительно текущего значения программного счетчика. Вместе с тем, имеется операция безусловного перехода JMP (jump — перейти на, перепрыгнуть на), при задании которой для указания адреса перехода предусмотрены все возможности адресации, используемые в связи с операциями преобразования, тестирования и пересылки данных, за исключением прямой регистровой.

Операция обращения к подпрограмме называется JSR (jump to subroutine — перейти на подпрограмму). Для задания начального адреса подпрограммы употребимы те же способы адресации, что и в случае JMP. Кроме того, указывается один из общих регистров в качестве хранителя адреса возврата, т. е. адреса следующей по ходу программы ячейки памяти. При выполнении JSR текущее значение этого регистра упрятывается в стек, после чего регистр запоминает адрес возврата. Подпрограмма должна оканчиваться командой RTS Rn (return from subroutine with Rn — возвратиться из подпрограммы по Rn), в которой Rn — тот же регистр, что и указанный в JSR для сохранения адреса возврата. При выполне-

нии RTS производится пересылка адреса возврата из Rn в программный счетчик, а в Rn принимается из стека прежнее значение.

Упрятывание адреса возврата из подпрограммы в общий регистр, а не непосредственно в стек обусловлено стремлением использовать этот адрес для связи подпрограммы с вызвавшей ее программой — он служит базой, относительно которой адресуются параметры, с которыми работает подпрограмма. Однако параметры можно передавать через стек, и в этом случае сохранять адрес возврата в регистре нет необходимости, его можно упрятывать прямо в стек. Такой режим устанавливается заданием в командах JSR и RTS в качестве регистра Rn программного счетчика PC, т. е. R7.

Наряду с JSR имеется другой аппарат обращения к подпрограммам, правда, с ограниченными возможностями адресации и передачи параметров, в форме так называемого командного прерывания, осуществляемого командами TRAP (trap — ловушка) и EMT (emulator trap — ловушка эмулятора), которые эмулируют прерывание, упрятывая текущие значения программного счетчика и слова состояния процессора в стек и устанавливая вместо них значения компонент сопоставленного каждой из них вектора прерывания. Таким образом происходит переход на соответствующую обслуживающую программу, которая должна оканчиваться командой RTI (return from interrupt — возвратиться из прерывания). Обслуживающая программа использует младший байт команды TRAP или EMT в качестве указателя, позволяющего идентифицировать до 256 подпрограмм. Разумеется, должна быть построена таблица перевода значений указателя в начальные адреса подпрограмм. Команда EMT используется в фирменных системных программах, команда TRAP предназначена для нужд пользователей.

К командам управления относятся также BPT (breakpoint trap — ловушка точки останова) и IOT (input/output trap — ловушка ввода/вывода), вызывающие прерывания с обслуживанием по сопоставленным им вектором, RTT (return from interrupt — возврат из прерывания), являющаяся вариантом RTI, HALT (halt — остановиться), вызывающая останов процессора, WAIT (wait for interrupt — ждать прерывания), приостанавливающая процессор в ожидании внешнего прерывания, RESET (reset external bus — установить внешнюю магистраль в исходное состояние), NOP (no operation — нет операции).

Операции, реализуемые в виде добавлений к базовому процессору, включают: MUL (multiply — перемножить), DIV (divide — поделить), ASH (arithmetic shift — арифметический сдвиг), производящая сдвиг в общем регистре на указанное в команде число позиций n, при $n < 0$ вправо, при $n > 0$ влево, ASHC (arithmetic shift combined — комбинированный арифметический сдвиг) производит сдвиг содержащегося в двух смежных регистрах 32-битного кода на указанное в команде число позиций n, при $n < 0$ вправо, при $n > 0$ влево, XOR (exclusive OR — исключающее ИЛИ) образует побитную неэквивалентность операнда назначения и указанного в команде регистра, SOB (subtract 1 and branch if not equal to 0 — вычесть 1 и ответвиться, если не равно 0), а также операции арифметики с плавающей запятой: FADD, FSUB, FMUL, FDIV с 32-битными операндами.

§ 3. Форматы и коды команд

Команды процессора унифицированной архитектуры представляются 16-битными словами, причем обычно команда бывает представлена одним словом, но в случае непосредственной, абсолютной и индексной адресации требуется дополнительно по одному слову на каждый операнд, т. е. одноопернадная команда оказывается двухсловной, а двухоперандная — трехсловной, если для обоих операндов использованы указанные виды адресации. Основное (первое) слово команды содержит код операции и, если необходимо, данные, обеспечивающие доступ к операнду или к операндам. Дополнительные слова в случае непосредственной адресации содержат значения операндов, а в прочих случаях — адреса или числовые константы для вычисления адресов. Имеются способы адресации, позволяющие модифицировать значения дополнительных слов.

Основными форматами команд являются: одно- и двухоперандный с 12-ю способами адресации операндов, формат команд ветвления с заданием адреса перехода в виде смещения относительно текущего значения программного счетчика, безоперационный — все командное слово является кодом операции, например, в командах HALT, NOP, RTI. Кроме того, имеются формат, в котором для одного операнда предусмотрена возможность адресации всеми 12-ю способами, а для другого — только путем прямой ссылки на общий регистр (команды JSR, XOR, MUL и DIV), а также подобный ему формат с заданием одного операнда в регистре, а другого — в виде числового значения, непосредственно заданного в командном слове (команды SOB, ASH, ASHC). В команде RTS единственный операнд задается указанием номера общего регистра. Команды EMT и TRAP в качестве операнда используют непосредственно заданное числовое значение младшего байта командного слова, команда MARK — значение, заданное 6-ю младшими битами командного слова. В командах установки и очистки битов состояния N, Z, V, C вид действия «установить/очистить» (1/0) и подвергаемые ему биты

указываются кодом младших 5 битов командного слова. В командах арифметики с плавающей запятой FADD, FSUB, FMUL, FDIV задается номер общего регистра, являющегося указателем стека, содержащего 32-битные значения операндов, а по выполнении операции — ее результат.

Коды команд приведены ниже в виде таблицы. Значения 16-битного командного слова К(15:0) представлены шестерками восьмеричных цифр, причем позиции, составляющие поля операндов,

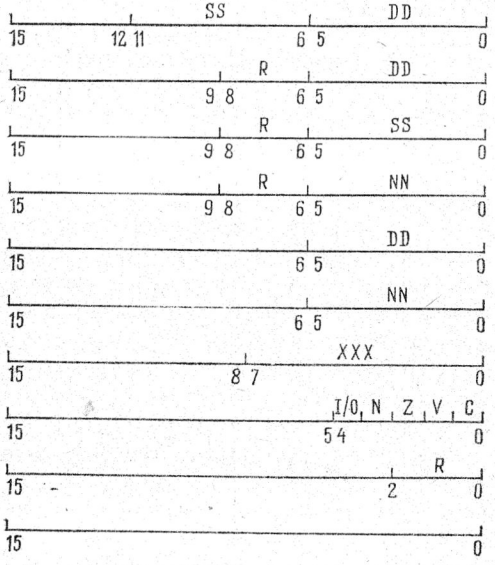


Рис. 3

заполнены буквами D, S, R, N, X, используемыми в следующем смысле:

DD — поле операнда назначения (6 битов),

SS — поле операнда отправления (6 битов),

R — указатель (номер) регистра (3 бита),

NN — числовой литерал (6 битов),

XXX — относительный адрес-смещение (8 битов).

Команды с двумя полями SS и DD могут иметь одно или два дополнительных слова. Команды с одним полем DD или SS могут иметь одно дополнительное слово. Форматы основного слова команды приведены на рис. 3.

Основным группам команд соответствуют следующие интервалы значений кода операции:

Таблица

Коды команд унифицированной архитектуры

Восьмеричный код	Мнемоника	Восьмеричный код	Мнемоника
00 00 00	HALT	10 00 XXX	BPL
00 00 01	WAIT		
00 00 02	RTI		
00 00 03	BPT		
00 00 04	IOT		
00 00 05	RESET		
00 00 06	RTT		
00 00 07 … 00 00 77	свободны		
00 01 DD	JMP		
00 02 0R	RTS		
00 02 10 … 00 02 37	свободны		
00 02 40	NOP		
00 02 41 … 00 02 77	установка/ очистка битов		
00 03 DD	SWAB		
00 04 XXX	BR	10 04 XXX	BMI
00 10 XXX	BNE	10 10 XXX	BHI
00 14 XXX	BEQ	10 14 XXX	BLOS
00 20 XXX	BGE	10 20 XXX	BVC
00 24 XXX	BLT	10 24 XXX	BVS
00 30 XXX	BGT	10 30 XXX	BCC, BHIS
00 34 XXX	BLE	10 34 XXX	BCS, BLO
00 4R DD	JSR	10 40 00 … 10 43 77	EMT
		10 44 00 … 10 47 77	TRAP
00 50 DD	CLR	10 50 DD	CLRB
00 51 DD	COM	10 51 DD	COMB
00 52 DD	INC	10 52 DD	INCB
00 53 DD	DEC	10 53 DD	DECB
00 54 DD	NEG	10 54 DD	NEGB
00 55 DD	ADC	10 55 DD	ADCB
00 56 DD	SBC	10 56 DD	SBCB
00 57 DD	TST	10 57 DD	TSTB
00 60 DD	ROR	10 60 DD	RORB
00 61 DD	ROL	10 61 DD	ROLB
00 62 DD	ASR	10 62 DD	ASRB
00 63 DD	ASL	10 63 DD	ASLB

Продолжение

Восьмеричный код	Мнемоника	Восьмеричный код	Мнемоника
00 64 NN	MARK	10 64 SS	MTPS
00 65 00	свободны	10 65 00	свободны
.. .. ..		.. .. ..	
00 66 77		10 66 77	
00 67 DD	STX	10 67 DD	MFPS
00 70 00	свободны	10 70 00	свободны
.. .. ..		.. .. ..	
00 77 77		10 77 77	
01 SS DD	MOV	11 SS DD	MOVB
02 SS DD	CMP	12 SS DD	CMPB
03 SS DD	BIT	13 SS DD	BITB
04 SS DD	BIC	14 SS DD	BICB
05 SS DD	BIS	15 SS DD	BISB
06 SS DD	ADD	16 SS DD	SUB
07 0R SS	MUL	17 00 00	свободны
07 1R SS	DIV		
07 2R NN	ASH		
07 3R NN	ASHC		
07 4R DD	XOR		
07 50 0R	FADD		
07 50 1R	FSUB		
07 50 2R	FMUL		
07 50 3R	FDIV		
07 50 40	свободны		
.. .. ..			
07 67 77			
07 7R NN	SOB	17 77 77	

— двухоперандные с полным набором способов адресации для обоих операндов — K(17:12) = 001:006, 101:106;

— одноопе рандные с полным набором способов адресации операнда — K(15:6) = 0001, 0050:0063, 0067, 1050:1064, 1067;

— ветвления с адресацией смещением относительно текущего значения программного счетчика — K(15:6) = 0004:0034, 1000: 1034;

Коды, отмеченные в таблице как свободные, не используются на существующих микрокомпьютерах унифицированной архитектуры — при попытке выполнить такой код возникает прерывание по «резервной» команде. Некоторые из указанных в таблице команд могут быть реализованы не на всех моделях унифицированной архитектуры. Например, на микрокомпьютерах серии «Электроника НЦ-80» отсутствует команда деления DIV и команды арифметики с плавающей запятой: FADD, FSUB, FMUL, FDIV.

§ 4. Способы адресации

Основная система адресации в унифицированной архитектуре связана с указанием операндов 6-битными полями SS и DD в первом слове команды. Структура этих полей одна и та же и их содержимое интерпретируется одним и тем же механизмом формирования исполнительного адреса, названия же SS и DD отражают разделение операндов на исходные (source), или операнды отправления, и назначения (destination), или принимающие значение результата операции. Разделение весьма условное и вполне оправданное, пожалуй, только в случае операции пересылки данных: в поле SS содержатся данные для вычисления адреса отправления, а в поле DD — адреса назначения пересылаемого слова или байта. В других двухоперандных, а также в одноопе рандных командах адресат, указываемый в поле DD, служит и отправлением и назначением, т. е. является как операндом-источником, так и операндом, принимающим результат. В одноопе рандных командах и в двухоперандных с полями SS и DD поле DD составляют шесть младших битов K(5:0), при этом полем SS являются следующие шесть битов K(11:6). В двухоперандных командах с регистром в качестве операнда назначения полем SS служат младшие биты K(5:0).

Шестибитное адресное поле обеспечивает указание операнда в общем регистре либо в ячейке главной памяти (в периферийном регистре, входящем в единое адресное пространство микрокомпьютера) посредством общего регистра и, может быть, дополнительного слова команды. Младшая тройка битов адресного поля служит указателем общего регистра, т. е. содержит номер n регистра. Старшая тройка битов составляет код mc способа адресации. Два старших бита этого кода обозначают вид адресации: 00 — регистровая, 01 — автоинкрементная, 10 — автодекрементная, 11 — индексная. Младший бит кода mc является указателем косвенности: 0 — прямая адресация, 1 — косвенная. В совокупности код mc представляют обычно, как и номер n, восьмеричной цифрой, четным значениям которой соответствует прямая, а нечетным — косвенная адресация, например: 0_8 — регистровая прямая, 1_8 — регистровая косвенная, 2_8 — автоинкрементная прямая, 3_8 — автоинкрементная косвенная и т. д. — всего восемь различных способов адресации.

В символических обозначениях языка ассемблера номер n общего регистра задается в виде %n, а при вводимых обычно эквивалентных обозначениях — Rn. Задание способа адресации обеспечивается при помощи знака косвенности @, а также путем заключения Rn в скобки и употребления знака «минус» в качестве префикса и знака «плюс» в качестве постфикса.

Рассмотрим символику и смысл каждого вида адресации, задаваемой в команде при помощи шестибитного адресного поля, в отдельности.

Регистровая прямая адресация (mc = 0) по общему регистру с номером n выражается на языке ассемблера употреблением в качестве операнда имени этого регистра — Rn. Как операнд отправления Rn доставляет копию хранимого в данном регистре значения, а как операнд назначения, кроме того, представляет данный регистр в качестве переменной, которой присваивается значение результата операции. Например, однооперандная команда изменения знака

```
NEG R2
```

вызывает изменение знака числа, содержащегося в регистре R2, т. е. эквивалентна присваиванию R2 := —R2 .

Двухоперандная команда пересылки

```
MOV R0, R3
```

с операндом отправления R0 и операндом назначения R3 (в записи на языке ассемблера операнды разделяются запятой) задает присваивание

```
R3 := R0;
```

т. е. пересылку в регистр R3 копии значения, содержащегося в регистре R0. Разумеется, R0 при этом сохраняет свое значение неизменным.

Команда вычитания с прямой регистровой адресацией обоих операндов

```
SUB R2, R3
```

вызывает присваивание

```
R3 := R3 — R2;
```

т. е. из операнда назначения вычитается операнд отправления, значение которого при этом сохраняется неизменным.

Еще пример — команда сравнения

```
CMP R2, R3
```

вычисляет разность R2 — R3 (отправление минус назначение) и, как всякая команда, производит установку битов N, Z, V, C по полученному результату. Значения операндов R2 и R3 сохраняются неизменными.

Операции, выполняемые с операндами-битами, в случае прямой регистровой адресации относятся к младшим 8 битам R(7:0) регистров, причем значение старшего байта R(15:8) сохраняется

неизменным. Например, команда циклического сдвига вправо над байтом

```
RORB R3
```

реализуется как присваивание

```
R(7:0)Cb := CbR(7:0);
```

где Cb — бит переноса в слове состояния процессора, обозначаемый также просто C. Команда сравнения байтов

```
CMPB R2 R3
```

установит биты слова состояния по разности R2(7:0) — R3(7:0).

Исключением является операция MOVB: пересылка байта в регистр сопровождается расширением знака числа, принимаемого в младший байт, на всю длину регистра, т. е. все биты R(15 : 8) принимают значение бита знака R(7) пересылаемого байта. Например, команда

```
MOVB R4, R4
```

произведет расширение знака младшего байта в регистре R4, установив биты старшего байта R4 (15 : 8) равными R4 (7), так что значение регистра в целом будет равно значению его младшего байта, интерпретируемого как число со знаком.

Заметим, что операции ADD и SUB, а также MUL и DIV выполняются только над операндами-словами, разновидности этих операций над байтами не имеется. Заметим также, что прямая регистровая адресация неприемлема в командах JMP и JSR. Принято, что переход производится по исполнительному адресу, а не по значению операнда, интерпретируемому как адрес. Поэтому, например, запись JMP R3 не имеет смысла и процессор, получив ее в качестве команды, выдает сообщение об ошибке.

Регистровая косвенная адресация (mc = 1) заключается в том, что регистр, номер которого представлен в адресном поле команды, рассматривается как указатель ячейки главной памяти, содержащей значение операнда или принимающей это значение в качестве результата операции. Другими словами, содержимое регистра является адресом ячейки, над которой производится операция, т. е. исполнительный (эффективный) адрес получает свое значение из регистра:

```
EA := Rn;
```

По этому адресу осуществляется доступ к ячейке-операнду m(EA) или переход в командах JMP и JSR.

В обозначениях языка ассемблера косвенная адресация по регистру Rn записывается в виде

@Rn или (Rn)

Например, команду изменения знака числа, содержащегося в ячейке памяти, адрес которой находится в регистре R3, можно записать двумя способами:

NEG @R3 и NEG (R3)

Обе записи означают одно и то же:

m(R3) := —m(R3);

т. е. процессор получит копию числа, хранящегося в ячейке m(R3), изменит знак этого числа на противоположный и значение результата поместит в ту же ячейку m(R3). Поскольку операция относится к операнду-слову (NEG, а не NEGB), значение адреса R3 должно быть четным — нечетный адрес при адресации слова вызывает сообщение об ошибке. Если же указана операция над байтом, например, NEGB(R3), то адрес R3 может быть как четным, так и нечетным — будет изменен знак соответствующего байта в главной памяти.

Двухоперандная команда с косвенной регистровой адресацией обоих операндов задает операцию в режиме память — память: значения операндов считываются из указываемых регистрами ячеек памяти, значение результата записывается по адресу назначения. Например, команда сложения

ADD (R2), (R4)

выполняется так:

m(R4) := m(R4) + m(R2);

Команда пересылки с косвенной регистровой адресацией операндов вызывает копирование в ячейку назначения содержимого ячейки отправления. Например, пересылка байта

MOVB (R3), (R2)

выполняется так:

m(R2) := m(R3);

В случае пересылки слова или иной операции над словами значения регистров-указателей должны быть четными.

Адресация операндов двухоперандной команды не обязана быть одинаковой. Например, один операнд может адресоваться косвенно, а другой прямо. Так, команда

SUB (R4), R2

означает вычитание из регистра R2 содержимого ячейки, на которую указывает регистр R4:

R2 := R2 — m(R4);

Команда с обратной последовательностью операндов

SUB R2, (R4);

вычитает значение регистра R2 из содержимого ячейки, указываемой регистром R4:

m(R4) := m(R4) — R2;

Команды JMP и JSR с косвенной регистровой адресацией вызывают переход и переход с запоминанием возврата, используя содержимое регистра-указателя в качестве адреса перехода. Например, команда

JMP (R5)

означает PC := R5. Значение регистра должно быть четным, поскольку адреса команд не могут быть нечетными.

Автоинкрементная прямая адресация (mc = 2), подобно регистровой косвенной, интерпретирует значение названного в адресном поле регистра как исполнительный адрес, но кроме того, включает автоматическое приращение этого адреса после использования его для доступа к операнду (поставтоинкремент). Приращение осуществляется прибавлением 1 в случае операнда-байта и прибавлением 2 в случае операнда-слова, чтобы происходил перебор соответственно последовательно расположенных в памяти байтов и слов.

На языке ассемблера автоинкрементная адресация операнда по регистру Rn записывается в виде

(Rn)+

где знак +, стоящий после заключенного в скобки имени регистра, означает поставтоинкремент значения регистра. Реализация данной записи заключается в следующем:

EA := Rn; IncrRn;

Процедура IncrRn прибавляет к Rn единицу, если адресуется байт, а если адресуется слово, то прибавляет двойку. С учетом того, что команды обработки слов, за исключением вычитания (код операции K(15:12) = 16_8), содержат в старшем бите нуль, т. е. K(15) = = 0, а также того, что по указателю стека R6 и программному счетчику, т. е. при n $\geqslant$ 6, возможна адресация только слов, процедура

IncrRn может быть выражена так:

if K (15) = 1 ∧ K(15 : 12) ≠ 16_8 ∧ n < 6 **then** Rn := Rn + 1 **else** Rn := Rn + 2;

Пример. Команда сложения с автоинкрементной адресацией операнда отправления

ADD (R2)+, R3

выполняется процессором так:

R3 := R3 + m(R2); R2 := R2 + 2;

т. е. путем прибавления к регистру назначения R3 значения ячейки, указываемой регистром R2, и последующего приращения адреса в регистре R2 добавлением двойки. Чтобы получить сумму ряда чисел, расположенных в k последовательных ячейках главной памяти, достаточно заслать в R2 адрес младшей из этих ячеек, очистить R3 и выполнить данную команду k раз.

Копирование содержимого одной области памяти в другую реализуется многократным выполнением, например, команды

MOV (R4)+, (R2)+

после того, как в регистры R4 и R2 засланы начальные адреса соответственно области отправления и области назначения. Автоинкрементная адресация вместе с автодекрементной является средством организации стеков.

В случае двухоперандной команды с автоинкрементной (и автодекрементной) адресацией результат операции может зависеть от того, в какой последовательности осуществляется доступ к операндам. Например, при адресации обоих операндов по одному и тому же регистру, как в команде

ADD R2, (R2)+

Нормальной является последовательность, при которой сначала определяется значение операнда отправления, а затем обрабатывается операнд назначения. Впрочем, имеются процессоры, не удовлетворяющие этому правилу.

Автоинкрементная косвенная адресация (mc=3) на языке ассемблера записывается в виде

@(Rn)+

т. е. к обозначению автоинкрементной прямой адресации добавлена спереди литера @, символизирующая косвенность. Это означает, что содержимое ячейки памяти, являющееся в случае прямой автоинкрементной адресации значением операнда, теперь должно рассматриваться как исполнительный адрес EA, т. е. адрес ячейки, в которой хранится значение операнда. Короче говоря, запись @(Rn)+ реализуется так:

EA := m(Rn); Rn := Rn + 2;

Приращение регистра указателя Rn равно 2, потому что указывает он ячейки, содержащие адреса, т. е. слова, а не байты.

Команда приращения байта с косвенной автоинкрементной адресацией по регистру R3

INCB @(R3)+

выполняется следующим образом:

m(m(R3)) := m(m(R3)) + 1; R3 := R3 + 2;

Аналогичная команда приращения слова

INC @(R3)+

выполняется точно так же, но прямой адрес m(R3) операнда должен быть четным, потому что адресами слов являются четные числа.

Автодекрементная прямая адресация (mc = 4) в противоположность автоинкрементной является преавтодекрементной: убавление (декремент) регистра-указателя производится прежде, чем содержащийся в нем адрес будет использован для доступа к операнду. На языке ассемблера это обозначается знаком «минус», стоящим перед заключенным в скобки именем регистра:

−(Rn)

Соответствующая процедура получения исполнительного адреса EA имеет вид

DecrRn; EA := Rn

где DecrRn определяется аналогично IncrRn, а именно:

if K (15) = 1 ∧ K (15:12) ≠ 16_8 ∧ n < 6 **then** Rn := Rn − 1 **else** Rn := Rn − 2;

Как указывалось выше, использование совместно автоинкрементной и автодекрементной адресации позволяет организовывать стеки. Преавтодекрементную и поставтоинкрементную адресации естественно употребить для получения стека, растущего в направлении убывания адресов с указателем, показывающим на позицию, занятую последней (вершину стека). Засылка в такой стек осуществляется при помощи преавтодекрементной, а выталкивание из стека — при помощи поставтоинкрементной адресации. Например,

используя в качестве указателя стека регистр R4, засылку в стек значения регистра R3 реализуем командой

MOV R3, —(R4)

а выталкивание из стека в регистр R3 соответственно командой

MOV (R4)+ , R3

Эти же виды адресации позволяют осуществить стек, растущий в направлении возрастания адресов, но с указателем, показывающим первую незанятую позицию (надвершину). Занесение и выталкивание элемента из такого стека реализуется командами

MOV R3, (R4)+
MOV —(R4), R3

Автодекрементная косвенная адресация (mc = 5) как и автоинкрементная косвенная заключается в интерпретации значения регистра в качестве косвенного адреса, но с предварительным убавлением на 2:

Rn:= Rn — 2; EA:= m (Rn);

На языке ассемблера записывается в виде

@—Rn

Например, команда приращения операнда-байта, адресуемого по регистру R0:

INCB @—(R0)

выполняется так:

R0:= R0 — 2; m(m(R0)):= m(m(R0)) + 1;

Индексная прямая адресация (mc = 6) представляет собой вычисление исполнительного адреса EA как суммы содержимого регистра Rn, номер которого указан в поле адресации, и числа, значение которого задано в виде дополнительного слова команды. Если команда двухоперандная и для обоих операндов употреблена индексная адресация, то в команде имеется два дополнительных слова: первое — для операнда отправления, второе — для операнда назначения.

На языке ассемблера прямая индексная адресация операнда обозначается в виде записи в скобках имени регистра, перед которой стоит, вообще говоря, выражение X, в результате вычисления которого ассемблер получает значение числа, включаемое в команду в виде дополнительного слова:

X(Rn)

В простейшем и наиболее частом случае выражение X является числовым литералом, возможно со знаком, или именем известной ассемблеру величины.

При выполнении команды исполнительный адрес операнда с индексной прямой адресацией по регистру Rn вычисляется так:

EA := Rn + m(PC); PC := PC + 2;

где PC представляет текущее значение программного счетчика. Поскольку каждая выборка из памяти слова программы сопровождается приращением PC, то после выборки основного слова команды PC указывает на первое дополнительное слово (если оно есть), а после выборки первого дополнительного слова — на второе дополнительное (если оно есть).

Пример. Команда сложения с прямой индексной адресацией обоих операндов

ADD —20 (R0), 8 (R6)

модифицирует значение находящегося на глубине, равной 8/2 = 4, элемента стека (R6 — указатель стека SP) прибавлением содержимого ячейки m (R0 — 20). Выполнение этой команды, введя переменную TMP для временного запоминания операнда отправления, можно описать так:

TMP := m(R0 + m(PC)); PC := PC + 2;
m(R6 + m(PC)) := m(R6 + m(PC)) + TMP; PC := PC + 2;

Значение m(PC), прибавляемое к R0, в данном примере равно —20, а прибавляемое затем к R6, оно равно 8.

Заметим, что в случае, когда выражение X равно 0, индексная прямая адресация равносильна регистровой косвенной, но в отличие от последней требует дополнительного слова команды.

Индексная косвенная адресация (mc = 7) отличается от прямой индексной тем, что вычисленное рассмотренным только что способом значение EA интерпретируется не как прямой, а как косвенный адрес операнда и используется для доступа к прямому адресу, т. е. исполнительный адрес вычисляется так:

EA := m(Rn + m(PC)); PC := PC + 2;

На языке ассемблера это записывается с использованием обозначения прямой индексной адресации, к которому спереди добавляется символ косвенности

@X(Rn)

Например, команда пересылки слова в регистр R0 из ячейки, адресуемой с помощью косвенной индексной адресации по

регистру R6, имеющая вид

MOV @20(R6), R0

реализуется следующим образом:

R0 := m(m(R6 + m(PC)); PC := PC + 2;

Адресация с использованием программного счетчика в качестве общего регистра в автоинкрементном и индексном режимах заслуживает особого рассмотрения, поскольку оказывается равносильной таким важным видам адресации как непосредственная, абсолютная, относительная и относительная косвенная.

Непосредственная адресация эмулируется как автоинкрементная прямая по регистру PC (т. е. R7). Запись операнда на языке ассемблера в виде

(PC)+

реализуется процессором так:

EA := PC; PC := PC + 2;

Поскольку PC после выборки основного слова команды получил приращение на 2 и, таким образом, указывает следующее слово программы, то автодекрементная прямая адресация обеспечивает доступ именно к этому слову с последующим приращением PC, благодаря чему PC будет указывать на очередное слово программы. Другими словами, при использовании рассматриваемого способа адресации резервируется дополнительное слово команды в качестве операнда, над которым осуществляется указанная в этой команде операция.

Практически полезным и важным применением данной возможности является использование введенного слова в качестве литерала, т. е. непосредственно заданного в команде значения. Поэтому в языке ассемблера автодекрементной прямой адресации по регистру PC сопоставлено особое обозначение, представляющее именно непосредственную адресацию:

#X

Знак непосредственной адресации #, стоящий перед выражением X, означает, что операндом является литерал, равный вычисленному ассемблером и помещенному в дополнительное слово команды значению выражения X. Например, команда вычитания с непосредственной адресацией операнда отправления

SUB #32, R0

вызовет вычитание заданного в ней значения из операнда назначения:

R0 := R0 — 32;

Технически это осуществляется следующим образом:

R0 := R0 — m(PC); PC := PC + 2;

Здесь m(PC) — дополнительное слово команды, содержащее занесенное в него ассемблером значение операнда, в данном случае равно 32.

Ассемблер переводит непосредственное задание операнда в обладающую значительно большими возможностями автоинкрементную прямую адресацию по регистру PC. Поэтому оказывается, например, что команда

INC #25

при каждом выполнении ее увеличивает значение дополнительного слова (первоначально равное 25) на 1. Это несовместимо с общепринятым смыслом непосредственной адресации, связываемым с символом #, и не должно допускаться в программах — не следует употреблять непосредственную адресацию для операндов назначения.

Абсолютная адресация мыслится как «непосредственная косвенная» — ее запись на языке ассемблера состоит из записи непосредственной адресации, перед которой помещен знак косвенности:

@#X

Это надо понимать так, что литерал #X, хранящийся в дополнительном слове команды, является не значением операнда, а адресом ячейки, в которой находится значение.

В действительности ассемблер отображает предписание @#X на автоинкрементную косвенную адресацию с регистром PC: значение X запоминается в виде дополнительного слова команды, а в адресное поле операнда записывается код mc = 3, n = 7, которому нормально соответствует в языке ассемблера обозначение

@(R7)+ или @(PC)+

При выполнении команды исполнительный адрес операнда получается так:

EA := m(PC); PC := PC + 2;

Например, команда перехода по абсолютному адресу

JMP @#2000

выполняется фактически как команда с автоинкрементной косвенной адресацией следующим образом:

EA := m(PC); PC:= PC + 2; PC := EA;

причем m(PC) = 2000.

Относительная адресация реализуется как прямая индексная по регистру PC. Исполнительный адрес получается сложением числа, возможно со знаком, хранящегося в дополнительном слове команды, доступном как m(PC) при текущем значении программного счетчика PC:

TMP := m(PC); PC := PC + 2; PC := PC + TMP;

Здесь, как и ранее, TMP — вспомогательная переменная.

На языке ассемблера данная адресация задается в виде выражения X, представляющего символический адрес (метку) операнда. В машинном коде этот адрес реализуется посредством составляющего значение дополнительного слова команды приращения, равного X —PC —2, к значению программного счетчика, принимаемому им непосредственно после выборки дополнительного слова, т. е. превышающему адрес этого слова на 2.

Таким образом, рассмотренная выше абсолютная адресация представлена на языке ассемблера как косвенная непосредственная @#X, а относительная задается в форме, которая скорее соответствует заданию абсолютной: X. Оправдание этому следует искать в стремлении облегчить разработку позиционно-независимых (свободно перемещаемых в памяти) программ — программист манипулирует метками, а вычисление соответствующих относительных адресов (приращений) возложено на ассемблер.

Пример. Первое (основное) слово команды

INCB L + 5

находится в ячейке, адрес которой больше адреса, соответствующего метке L, на 20. Текущее значение PC при обращении к дополнительному слову команды будет равно PC = L + 20 + 2 = L + 22 и, соответственно, величина смещения (значение дополнительного слова) составит:

X — PC — 2 = L + 5 — (L + 22) — 2 = —19

Относительная косвенная адресация — это индексная косвенная по регистру PC. Запись на языке ассемблера получается из принятой для относительной прямой адресации добавлением спереди символа косвенности

@X

Здесь X — выражение, определяющее теперь косвенный адрес (адрес адреса) операнда.

Значение дополнительного слова команды, адрес которого PC, равно X — PC — 2. Исполнительный адрес вычисляется так:

TMP := m(PC); PC := PC + 2; EA := m(PC + TMP);

Относительная адресация в командах ветвления (branch) осуществляется без дополнительного слова — для представления смещения используется младший байт K(7:0) основного слова команды. Поскольку переходы должны производиться только по четным адресам, то значение байта смещения, рассматриваемое как число со знаком, для получения исполнительного адреса прибавляется к текущему значению программного счетчика удвоенным:

EA:= PC + 2 * sgnvl(K(7:0));

Функция sgnvl( ) доставляет числовое со знаком значение кода. Будучи удвоенным, это значение лежит в пределах от —256 до +254. С учетом того, что используемое для вычисления EA значение PC на 2 больше адреса команды ветвления, имеем интервал приращений относительно этой команды от —254 до +256.

На языке ассемблера адреса перехода в командах ветвления, как и в случае рассмотренной выше относительной адресации в команде с дополнительным словом, задаются в виде выражения X, доставляющего символический адрес ячейки памяти. Ассемблер вычисляет смещение относительно текущего в момент выполнения команды значения программного счетчика и, поделив его на 2, записывает в младший байт формируемой команды ветвления. Если ячейка, на которую должен производиться переход, отстоит от формируемой команды более, чем на +256 или на —254, величина смещения оказывается не представимой байтом — ассемблер выдает сообщение об этом.

Пример относительной адресации в команде ветвления. Команда безусловного ответвления, помеченная меткой L1 и задающая переход на метку L2, записывается в виде

L1: BR L2

Смещение L2 — L1 должно удовлетворять условию

$-254 \leqslant L2 - L1 \leqslant 256$

При этом младший байт команды ветвления K(7:0) полагается равным (L2 — L1 + 2)/2.

§ 5. Алгоритмы выполнения команд

Команды, выполняемые процессором унифицированной архитектуры, по их назначению разделяются прежде всего на две группы: 1) обеспечивающие пересылку, преобразование и тестирование данных, 2) управляющие последовательностью выполнения частей

программы и функционированием процессора. Используемая командами управления информация о результате пересылки или преобразования данных передается посредством слова состояния процессора: при выполнении команды пересылки, преобразования или тестирования биты N, Z, V, C принимают значения, характеризующие полученный результат, а команды управления выполняются тем или иным образом в зависимости от значений этих битов.

Поведение процессора зависит также от двух других битов слова состояния: T-бита (trap bit) и бита PS(7), характеризующего приоритет процессора в отношении к захвату магистрали. Если PS(7) = 1, процессор обладает наивысшим приоритетом — устройства, запрашивающие прерывания, лишены возможности захватить магистраль, внешние прерывания запрещены. При PS(7) = 0 прерывания разрешены. Если T = 1, то по завершении выполняемой команды происходит внутреннее прерывание, переключающее процессор на программу связи с оператором (режим отладочных действий).

Перечисленные биты составляют младший байт слова состояния процессора (старший байт на микрокомпьютерах не используется) и размещены в нем следующим образом:

7	6	5	4	3	2	1	0
PS(7)			T	N	Z	V	C

Биты 5 и 6 не используются.

Младший байт слова состояния PS(7:0) можно копировать в байт главной памяти командой MFPS, имеющей код 1067DD. Обратная засылка байта из памяти в PSW осуществляется командой MTPS (код 1064SS), однако этой командой нельзя установить T = 1.

Имеется также группа команд очистки и установки в 1 битов N, Z, V, C. Команды CLN (clear N), CLZ, CLV, CLC очищают указанный в каждой из них бит, а команды SEN (set N), SEZ, SEV, SEC устанавливают указанный бит в 1. Имеются также команда CCC (clear condition code), очищающая все четыре бита, и команда SCC (set condition code), устанавливающая все четыре бита в 1. Код команд данной группы можно представить в виде суммы $000240_8 + K(4:0)_8$, имея в виду, что K(4) = 0 означает очистку, а K(4) = 1 — установку в 1 тех битов слова состояния, которым в командном слове соответствуют биты, содержащие 1. Например, код CLC: 000240 + 01 = 000241, код SEC: 000240 + 21 = 000261, код SCC: 000240 + 37 = 000277, код очистки V и C: 000240 + + 03 = 000243 и т. д.

Рассмотренные средства принудительного задания значений битов слова состояния являются вспомогательными — в процессе

выполнения программы значения этих битов вырабатываются автоматически по результатам производимых операций, а значениями слова состояния в целом, включая бит приоритета и T-бит, служат вторые компоненты векторов прерывания, хранящиеся в отведенных для них ячейках главной памяти или пересылаемые командами RTI и RTT из стека возвратов по завершении обработки прерываний.

Команды пересылки, преобразования и тестирования данных задают операции, выполняемые над операндами, которые указываются при помощи рассмотренных в предыдущем параграфе способов адресации. В случае прямой регистровой адресации в качестве операнда выступает регистр, в случае непосредственной адресации — значение, непосредственно заданное в команде, в остальных случаях — ячейка m(EA) главной памяти, адрес которой EA вычисляется на основании имеющихся в адресном поле операнда данных. Описывая алгоритмы выполнения команд, мы отвлекаемся от адресации операндов и считаем их уже данными в виде слова s(15:0) или байта s(7:0) отправления и слова d(15:0) или байта d(7:0) назначения. Таким образом, буквы s и d в описаниях команд обозначают регистры или ячейки памяти, или непосредственно заданные в командах значения в зависимости от указанной в соответствующих адресных полях SS и DD адресации. Если операция выполняется над байтами, то в случае прямой регистровой адресации обозначениям s и d соответствует младший байт регистра.

Знаки $\neg$, $\wedge$, $\vee$, $\oplus$ в последующих описаниях команд обозначают побитные операции инверсии, конъюнкции, дизъюнкции и неэквивалентности, выполняемые над всеми битами или всеми парами соответствующих друг другу битов в словах-операндах.

Ради сокращения записи введена вспомогательная процедура ASSNZ, присваивающая значения битам N и Z слова состояния процессора:

if d $<$ 0 **then** N:=1 **else** N:=0; **if** d=0 **then** Z:=1 **else** Z:=0;

Кроме того, в описаниях используется для временного сохранения значений однобитная переменная b и переменная r, выступающая в операциях над 16-битными словами в качестве 17-битного слова r(16:0), а в операциях над байтами как 9-битное слово r(8:0). Операнд d в ряде случаев употребляется в конкатенации с битом C, например: dC значит d(15:0)C(1:1) или d(7:0)C(1:1), Cd значит C(1:1)d(15:0) или C(1:1)d(7:0).

Описания команд, задающих одну и ту же операцию над словами и над байтами в тех случаях, когда они полностью совпадают, совмещены. При этом к мнемокоду команды, относящейся к операндам-словам, добавлена буква B в скобках, например: CLR(B) означает CLR и CLRB.

Двухоперандные команды.

MOV s, d — переслать значение слова s в d:

d:=s; ASSNZ; V:=0;

MOVB s, d — переслать значение байта s в d:

d(7:0):=s; **for** i:=8 **to** 15 **do** d(i):=d(7); ASSNZ; V:=0;

Предложение **for** выполняется при условии, что d является регистром Rn.

CMP s, d — сравнить значения слов s и d:

r:=s+(¬d+1); N:=r(15); **if** r(15:0)=0 **then** Z:=1 **else** Z:=0;
V:=(s(15)⊕d(15))∧(s(15)⊕r(15)); C:=r(16);

CMPB s, d — сравнить значения байтов s и d:

r:=s+(¬d+1); N:=r(7); **if** r(7:0)=0 **then** Z:=1 **else** Z:=0;
V:=(s(7)⊕d(7))∧(s(7)⊕r(7)); C:=r(8);

BIT s, d — тестировать конъюнкцию слов s и d:

r:=s∧d; N:=r(15); **if** r(15:0)=0 **then** Z:=1 **else** Z:=0;
V:=0;

BITB s, d — тестировать конъюнкцию байтов s и d:

r:=s∧d; N:=r(7); **if** r(7:0)=0 **then** Z:=1 **else** Z:=0;
V:=0;

BIC(B) s, d — очистить в слове (байте) d биты, которым в слове (байте) s соответствуют установленные в 1:

d:=d∧¬s; ASSNZ; V:=0;

BIS(B) s, d — образовать в d дизъюнкцию d и s:

d:=d∨s; ASSNZ; V:=0;

XOR Rn, d — образовать в d неэквивалентность d и Rn:

d:=d⊕Rn; ASSNZ; V:=0;

ADD s, d — прибавить s к d:

b:=d(15)⊕¬s(15); Cd:=d+s; ASSNZ;
V:=b∧(d(15)⊕s(15));

SUB s, d — вычесть s из d:

b:=d(15)⊕s(15); Cd:=d+(¬s+1); ASSNZ;
V:=b∧(d(15)⊕¬s(15)));

Однооперандные команды.

CLR(B) d — очистить, положив равным нулю, слово (байт) d:

d:=0; Z:=1; N:=V:=C:=0;

COM(B) d — побитно инвертировать слово (байт) d:

d:=¬d; ASSNZ; V:=C:=0;

INC d — прибавить 1 к значению слова d:

if d=77777_8 **then** V:=1 **else** V:=0; d:=d+1; ASSNZ;

INCB d — прибавить 1 к значению байта d:

if d=177_8 **then** V:=1 **else** V:=0; d:=d+1; ASSNZ;

DEC d — вычесть 1 из значения слова d:

if d=100000_8 **then** V:=1 **else** V:=0; d:=d−1; ASSNZ;

DECB d — вычесть 1 из значения байта d:

if d=200_8 **then** V:=1 **else** V:=0; d:=d−1; ASSNZ;

NEG d — изменить знак значения слова d на противоположный:

Cd:=¬d+1; ASSNZ; **if** d=100000_8 **then** V:=1 **else** V:=0;

NEGB d — изменить знак значения байта d на противоположный:

Cd:=¬d+1; ASSNZ; **if** d=200_8 **then** V:=1 **else** V:=0;

ADC d — прибавить значение бита переноса C к значению слова d:

if d=77777_8 ∧C=1 **then** V:=1 **else** V:=0; Cd:=d+C; ASSNZ;

ADCB d — прибавить значение бита переноса C к значению байта d:

if d=177_8 ∧C=1 **then** V:=1 **else** V:=0; Cd:=d+C; ASSNZ;

SBC d — вычесть значение бита переноса C из значения слова d:

Cd:=d−C; ASSNZ; **if** d=100000_8 **then** V:=1 **else** V:=0;

SBCB d — вычесть значение бита переноса C из значения байта d:

Cd:=d−C; ASSNZ; **if** d=200_8 **then** V:=1 **else** V:=0;

TST(B) d — тестировать слово (байт) d:

ASSNZ; V:=C:=0;

ROR(B) d — циклический сдвиг (вращение) вправо в слове (байте) d с битом переноса C:

dC:=Cd; ASSNZ; V:=N⊕C;

ROL(B) d — циклический сдвиг влево в слове (байте) d с битом переноса C:

Cd:=dC; ASSNZ; V:=N⊕C;

ASR d — арифметический сдвиг вправо в слове d с битом переноса C:

d(14:0)C:=d; ASSNZ; V:=N⊕C;

ASRB d — арифметический сдвиг вправо в байте d с битом переноса C:

d(6:0)C:=d; ASSNZ; V:=N⊕C;

Операции ASR и ASRB обычно квалифицируют как целочисленное деление на 2. Однако следует иметь в виду, что в случаях применения их к нечетным отрицательным числам результат не совпадает с получаемым для тех же значений выполнением команды DIV. Дело в том, что операция ASR(B) выдает в результате целое, не превосходящее точного значения частного (целое в смысле алгола), а операция DIV — целое, абсолютная величина которого не превосходит абсолютной величины точного значения (целое в смысле фортрана). Например, применение ASR к значению —1 дает в результате —1, а частное от деления —1 на 2 операцией DIV равно 0.

ASL(B) d — арифметический сдвиг влево в слове (байте) d с битом переноса C:

C:=0; Cd:=dC; ASSNZ; V:=N⊕C;

SWAB d — перестановка байтов в слове d:

d:=d(7:0)d(15:8); ASSNZ; V:=C:=0;

SXT d — расширение знака числа в слове d по N:

if N=0 **then** d:=0 **else** d:=—1; Z:=N; V:=0;

Команды перехода с относительной адресацией.

Адрес в командах перехода (ветвления) с относительной адресацией представлен младшим байтом команды K(7:0) в виде числа со знаком, задающего смещение в словах программы относительно текущего значения программного счетчика. Для получения исполнительного адреса это смещение надо выразить числом байтов, умножив на 2, произвести расширение знака и сложить с текущим

значением программного счетчика, которое превышает адрес самой команды перехода на 2:

D:=2*K(7:0); **for** i:=8 **to** 15 **do** D(i):=D(8); EA:=PC+D;

При выполнении команды условного перехода вычисленное значение EA присваивается программному счетчику, если указанное в команде условие выполнено, а в противном случае сохраняется текущее значение PC, являющееся адресом следующей в линейном порядке командой программы. При выполнении команды безусловного перехода EA присваивается PC непременно. Всего имеется 15 команд перехода с адресацией байтом смещения.

BR (branch) — перейти безусловно:

PC:=EA;

BNE (branch if not equal (to 0)) — перейти, если операнды предшествующей операции сравнения не равны или если результат иной предшествующей операции не равен нулю:

if Z=0 **then** PC:=EA;

BQE (branch if equal (to 0)) — перейти, если сравниваемые операнды равны или если результат равен нулю:

if Z=1 **then** PC:=EA;

BPL (branch if plus) — перейти, если результат неотрицателен:

if N=0 **then** PC:=EA;

BMI (branch if minus) — перейти, если результат отрицателен

if N=1 **then** PC:=EA;

BCC (branch if carry is clear) — перейти, если перенос равен нулю:

if C=0 **then** PC:=EA;

BCS (branch if carry is set) — перейти, если перенос равен 1:

if C=1 **then** PC:=EA;

BVC (branch if overflow is clear) — перейти, если переполнения нет:

if V=0 **then** PC:=EA;

BVS (branch if overflow is set) — перейти, если имеет место переполнение:

if V=1 **then** PC:=EA;

Переходы по сравнению или тестированию чисел без знака:

BHI (branch if higher) — перейти, если выше, т. е. если в предшествующей команде сравнения значение операнда отправления превышает значение операнда назначения, $s > d$:

if $C=0 \wedge Z=0$ **then** PC:=EA;

BLOS (branch if lower or same) — перейти, если ниже или то же, т. е. если сравниваемые значения удовлетворяют условию $s \leqslant d$:

if $C=1 \vee Z=1$ **then** PC:=EA;

BHIS (branch if higher or same) — перейти, если выше или то же, т. е. если $s \geqslant d$ (это другая интерпретация команды BCC):

if C=0 **then** PC:=EA;

BLO (branch if lower) — перейти, если ниже, т. е. если $s < d$ (это другая интерпретация команды BCS):

if C=1 **then** PC:=EA;

Переходы по сравнению или тестированию чисел со знаком:

BGE (branch if greater or equal (to 0)) — перейти, если больше или равно (нулю), т. е. если в предшествующем сравнении чисел со знаком значение операнда отправления больше или равно значению операнда назначения ($s \geqslant d$) или если тестируемое число со знаком больше нуля или равно нулю:

if N=V **then** PC:=EA;

BLT (branch if less then (0)) — перейти, если меньше чем (нуль), т. е. если в сравнении чисел со знаком $s < d$, или если значение тестируемого числа меньше нуля:

if $N \neq V$ **then** PC:=EA;

BGT (branch if greater then (0)) — перейти, если больше чем (нуль), т. е. если $s > d$, или если значение тестируемого числа больше нуля:

if $Z=0 \wedge N=V$ **then** PC:=EA;

BLE (branch if less or equal (to 0)) — перейти, если меньше или равно (нулю), т. е. если $s \leqslant d$, или если тестируемое значение не больше нуля:

if $Z=1 \vee N \neq V$ **then** PC:=EA;

Заметим, что при сравнении чисел со знаком условие N=V равносильно $s \geqslant d$, а его отрицание $N \neq V$ равносильно $s < d$.

Действительно, если d не превосходит s, то результат выполняемого при сравнении вычитания s—d будет либо неотрицательным без переполнения (N=0 и V=0), либо произойдет переполнение с переходом фиксируемого словом значения в отрицательную область (N=1 и V=1). Таким образом, значения N и V в обоих возможных случаях совпадают: N=V. Аналогично, если s меньше d, то разность s—d будет либо отрицательной без переполнения (N=1, V=0), либо произойдет переполнение с изменением знака результата (N = 0, V = 1) — в обоих случаях значения N и V различны, т. е. $N \neq V$.

Следует иметь в виду, что в случае переполнения условия перехода по отношению результата к нулю не совпадают с условиями перехода по знаку результата. Например, BLT — перейти, если меньше чем нуль, не то же, что BMI — перейти, если результат отрицателен. Команда BMI вызывает переход при N = 1, а команда BLT — при $N \neq V$.

Специальные команды перехода для работы с числами без знака необходимы главным образом в связи с тем, что адреса ячеек памяти по традиции интерпретируются как числа без знака. Вместе с тем использование в качестве адреса числа со знаком привлекательно не только с точки зрения унификации, но и возможностью расширять адресное пространство в обоих направлениях от нуля. При этом, например, адреса, сопоставленные периферийным регистрам, будут неизменными и представленными начальными отрицательными значениями: —1, —2, —3 и т. д.

Прочие команды управления.

К ним относятся: безусловный переход JMP с адресацией, принятой в командах пересылки и преобразования данных, переход в обратном направлении SOB по ненулевому значению автодекрементного регистра, команды JSR, RTS и MARK для работы с подпрограммами, команды, связанные с прерываниями, а также сброса магистрали и остановки процессора.

JMP (jump) — переход по адресу EA, вычисленному на основании содержимого адресного поля K(5:0):

PC:=EA;

SOB (subtract 1 and branch if not equal to 0) — вычесть 1 и перейти, если результат не равен нулю:

n:=K(8:6); D:=2*K(5:0); Rn:=Rn—1; **if** $Rn \neq 0$ **then** PC:=PC—D;

На языке ассемблера эта команда записывается в виде

SOB Rn, X

где X — выражение, задающее адрес перехода (обычно метка). Поскольку смещение D является числом без знака, данная команда не позволяет осуществить переход в прямом направлении.

JSR (jump to subroutine) — переход на подпрограмму, начальный адрес которой EA вычисляется на основании содержимого адресного поля K(5:0), с сохранением адреса возврата в регистре Rn:

n:=K(8:6); SP:=SP—2; m(SP):=Rn; Rn:=PC; PC:=EA;

RTS (return from subroutine) — возврат из подпрограммы:

n:=K(2:0); PC:=Rn; Rn:=m(SP); SP:=SP+2;

Команда возврата из подпрограммы должна содержать тот же номер регистра, что и команда перехода на эту подпрограмму.

MARK — команда, обеспечивающая передачу параметров процедуре через стек. До выполнения команды JSR PC SUBR перехода на подпрограмму SUBR в стек засылаются последовательно: значение R5, N параметров и копия команды MARK, а значение указателя стека SP, являющееся адресом этой копии, присваивается регистру R5. Выполняемая затем команда JSR заносит в стек текущее значение PC в качестве адреса возврата и производит переход на начало подпрограммы. Подпрограмма заканчивается командой RTS R5 (в этом случае JSR и RTS ссылаются на различные регистры), которая пересылает в PC из R5 адрес команды MARK, находящейся в стеке, а в R5 засылает извлекаемый из стека адрес возврата:

PC:=R5; R5:=m(SP); SP:=SP+2;

Команда MARK, содержащая в поле K(5:0) число помещенных в стек параметров N, выполняется так:

SP:=PC+2*N; PC:=R5; R5:=m(SP); SP:=SP+2;

Указатель стека смещается на N позиций (на 2N байтов) от следующей за командой MARK, удалив из стека параметры и указывая теперь на поступившее в стек перед параметрами значение регистра R5. Программный счетчик принимает из R5 адрес возврата, а в R5 засылается извлеченное из стека прежнее значение. Во время выполнения подпрограммы адресация находящихся в стеке параметров возможна относительно регистра R5, который указывает позицию, занимаемую командой MARK, за которой непосредственно располагаются позиции, занятые параметрами.

Команды EMT и TRAP предназначены для прерывания выполняемой программы по сопоставленному каждой из них вектору прерывания, компоненты которого для команды EMT находятся по адресам 30 и 32, а для команды TRAP — 34 и 36 главной памяти.

Команда EMT вызывает следующую последовательность действий:

SP:=SP—2; m(SP):=PS; SP:=SP—2; m(SP):=PC; PC:=m(30); PS:=m(32);

Команда TRAP выполняется аналогично:

SP:=P—2; m(SP):=PS; SP:=SP—2; m(SP):=PC; PC:=m(34); PS:=m(36);

Таким образом, в результате выполнения каждой из данных команд процессор переключается на программу, обслуживающую соответствующее прерывание. Эта программа, используя сохраненный в стеке адрес возврата, обращается к младшему байту инициировавшей ее команды EMT или TRAP и получает из него номер подпрограммы, для выполнения которой возбуждено прерывание.

С командой EMT связаны системные подпрограммы, команда TRAP предусмотрена для подпрограмм пользователя и позволяет адресовать до 256 подпрограмм. Такие подпрограммы, как и все программы обработки прерываний, должны заканчиваться командой возврата из прерывания RTI или RTT. Обе эти команды выполняют одно и то же, а именно:

PC:=m(SP); SP:=SP+2; PS:=m(SP); SP:=SP+2;

Различаются они тем, что в случае, когда слово, засылаемое из стека в регистр состояния процессора PS, имеет T = 1, т. е. устанавливает режим пошагового (покомандного) выполнения программы, прерывание по T-биту после команды RTI происходит немедленно, а после RTT — с отсрочкой на одну команду: выполняется следующая за RTT команда и после нее происходит прерывание. Вектор прерывания по T-биту находится в ячейках с адресами 14 и 16 главной памяти. Засылка компонент этого вектора в PC и PS переключает процессор в пультовый режим (на программу связи с оператором), обеспечивая возможность выполнять отладочные действия.

Переключение в этот режим производится также командой прерывания для отладки:

BPT (breakpoint trap) — ловушка точки останова:

SP:=SP—2; m(SP):=PS; SP:=SP—2; m(SP):=PC; PC:=m(14); PS:=m(16);

Для обращения к подпрограмме управления вводом/выводом служит прерывание по команде

IOT (input/output trap) — ловушка ввода/вывода:

SP:=SP—2; m(SP):=PS; SP:=SP—2; m(SP):=PC; PC:=m(20); PS:=m(22);

Команда WAIT — ждать прерывания — запрещает процессору выборку команд из памяти в ожидании запроса прерывания внешним устройством, благодаря чему сокращается время реакции на запрос прерывания. По окончании обработки прерывания возобновляется выполнение прерванной программы с команды, непосредственно следующей за командой WAIT.

Команда RESET — сброс — устанавливает подключенные к магистрали внешние устройства в начальное состояние, т. е. в то состояние, в котором они находились после включения питания.

Команда HALT останавливает процессор в состоянии ожидания сигнала с пульта управления.

ГЛАВА 6

ДИАЛОГОВАЯ СИСТЕМА СТРУКТУРИРОВАННОГО ПРОГРАММИРОВАНИЯ ДССП

Описанные в гл. 4 и 5 8- и 16-битные архитектуры типичны для микрокомпьютеров и дают достаточно полное представление о характере микрокомпьютерной архитектуры вообще. Эту архитектуру можно квалифицировать как ориентированную на обеспечение возможно большего быстродействия и компактности кода путем изощренного кодирования, усложнения процессора и языка команд, т. е. в конечном счете путем увеличения трудности освоения и использования компьютера человеком. Такая ориентация унаследована от миникомпьютеров, для которых она являлась источником удешевления аппаратуры, сопровождавшегося все большим расширением сферы применений. Однако в реализуемых методами интегральной технологии микрокомпьютерных системах стоимость аппаратуры сократилась в сотни раз и составляет теперь, как правило, менее одной десятой стоимости системы, в то время как более девяти десятых приходится на программное оснащение. Понятно, что в этих условиях повышение машинной эффективности путем усложнения архитектуры и удорожания программирования не имеет смысла. И тем не менее, за крайне редкими исключениями, развитие архитектуры микрокомпьютеров продолжается, так сказать, по инерции, в направлении все большего усложнения с целью повысить машинную эффективность, обеспечить максимум «сырой мощности».

С точки зрения рационального подхода к делу требуется, очевидно, переориентировать развитие архитектуры на повышение эффективности программирования, нацеливаясь не на достижение максимума сырой мощности, а на оптимизацию эффективности с учетом ее как машинной, так и человеческой составляющей. Архитектура компьютера должна быть благоприятной для программиста, для программирования.

Ранее было показано, что данному требованию удовлетворяет архитектура, ориентированная на структурированное программирование, предоставляющая средства для естественного и эффективного построения строго структурированных программ, понуждающая программиста действовать рационально, дисциплинированно. Теперь в качестве примера такой архитектуры мы рассмотрим диа-

логовую систему структурированного программирования ДССП, созданную проблемной лабораторией электронных вычислительных машин Московского государственного университета *).

§ 1. Общая характеристика ДССП

ДССП осуществлена в виде эмулируемого на микрокомпьютере традиционной архитектуры двухстекового процессора (ДССП-процессора), отличающегося возможностью легко определять и использовать процедуры и наличием команд выполнения процедур по условию, циклического выполнения и выхода из цикла. Эти средства структурирования программы сочетаются в ДССП с диалоговым функционированием — все ресурсы системы и любые фрагменты создаваемой или используемой программы немедленно доступны программисту непосредственно с терминала во внешнем представлении, т. е. на языке высокого уровня. Соединением структурированного программирования и диалога достигнуто многократное увеличение человеческой составляющей эффективности системы. При этом машинная эффективность программ, выполняемых на эмулированном ДССП-процессоре, только в 1,5—2 раза меньше по сравнению с программами, созданными на языке ассемблера используемой машины. Реально ожидать, что при аппаратной реализации архитектура ДССП в отношении машинной эффективности будет сравнимой с типичными архитектурами современных микрокомпьютеров.

Главной целью создания ДССП было уменьшение трудоемкости и соответственно стоимости программирования микрокомпьютеров при примерно тех же, что и в системах программирования на языке ассемблера, универсальности предоставляемых средств и машинной эффективности производимых программ. Основной выигрыш мог быть обеспечен, как показал опыт работы с экспериментальной машиной «Сетунь 70», бескомпромиссным переходом на структурированное программирование путем введения процессора соответствующей архитектуры. Дополнительный выигрыш достигался за счет диалогового взаимодействия программиста с процессором. Вместе с тем, повышению эффективности человеческого труда в системе способствовали также другие воплощенные в ней принципы и решения, отмечаемые ниже.

*) Первая версия ДССП (ДССП/НЦ) разработана и реализована в 1980—1982 гг., вторая версия (ДССП-80) — в 1982—1983 гг. Работу по созданию этих версий выполнили Г. В. Златкус, И. А. Руднев, В. Б. Захаров, С. А. Сидоров. Основой ДССП явилась двухстековая архитектура структурированного программирования, реализованная ранее в экспериментальной машине «Сетунь 70», и словарная организация диалога, заимствованная из системы FORTH.

Программа в ДССП строится в виде нисходящей иерархии вложенных друг в друга процедур, причем определения процедур выражаются в бесскобочной постфиксной форме. Каждое определение записывается в виде последовательности разделенных пробелами слов, обозначающих процедуры, операции и операнды. Программа представляет собой нисходящую последовательность определений, т. е. в конечном счете последовательность слов, разделенных, как в естественных языках, пробелами. Этот примитивный синтаксис един во всех звеньях системы: управление процессором, периферийными устройствами, резервирование памяти, манипулирование файлами, редактирование текста и отладка программ — все осуществляется командами в виде слов или последовательностей слов.

Как правило, при выполнении слова, обозначающего процедуру или операцию (процедуру, входящую в набор команд процессора), необходимые операнды изымаются из стека операндов, а результаты засылаются в стек. При выполнении слов, именующих данные, в стек заносятся соответствующие значения или параметры, обеспечивающие доступ к структурам и значениям данных. Таким образом, овладение языком ДССП сводится к усвоению принципа действия стека и содержания выполняемых процессором операций, но не связано с необходимостью запоминать множество формальных правил и исключений, форматов команд или предложений и других условностей, характерных для большинства языков программирования. Язык ДССП — это не правила и форматы, а концепции и процедуры.

Соответственно надежность программ обеспечивается в ДССП не обилием спецификаций и придирчивым контролем типов, а обретенной в результате структурирования практической возможностью непосредственно убедиться в правильности функционирования всех звеньев во всех возможных режимах, т. е. экспериментально осуществить исчерпывающую проверку («доказательство корректности») программы. Это одно из проявлений принципа «меньше слов — больше дела», понимаемого, в частности, так, что нет смысла в умозрительных построениях и обоснованиях в случаях, когда легко осуществима прямая проверка практикой. Обеспечению надежности способствует также немудреность и открытость доступных программисту механизмов ДССП-процессора, естественность и очевидность производимого по умолчанию, нетрудность испытания и адаптации заимствуемых программ, неформальность базирующейся на понятии знака числа логики управления ходом программы.

Большим преимуществом ДССП является возможность легко и просто расширять язык процессора включением в него операций и структур данных, определяемых с помощью уже имеющихся в нем

средств. Любая определенная пользователем процедура может быть включена в набор команд процессора и использоваться наравне с базовыми операциями, в частности, в определениях новых операций. Таким образом, наряду с нисходящей декомпозицией в качестве основного способа программирования, ДССП позволяет осуществить восходящие иерархии процедур и структур данных, которые явятся языковыми средствами более высоких уровней и послужат богатой основой для создания проблемно-ориентированных систем самого различного назначения. Исходный уровень операций ДССП соответствует наиболее элементарным микрокомпьютерным командам, обеспечивающим возможность манипулирования битами. Это при наличии эффективного аппарата определения новых операций сообщает языку ДССП подлинную универсальность и гибкость.

Расширяемость языка ДССП позволила просто и единообразно решить проблемы программной реализации системы, описания ее и освоения пользователями. Все это осуществляется путем постепенного развития и наращивания минимальной совокупности крайне элементарных средств самими этими средствами. Так, только небольшая часть базовых операций ДССП реализована непосредственно в коде команд используемого микрокомпьютера, а все другие операции определены на постепенно расширяемом постфиксном языке. Соответственно, описание системы базируется на минимуме понятий и операций, в терминах которых выражается все дальнейшее. Наконец, освоение системы пользователем начинается с тех же простейших понятий и операций, первоначально выполняемых в режиме стекового микрокалькулятора, а затем постепенно дополняемых все более сложными средствами.

Понятность ДССП-программ обеспечивается, наряду с нисходящим структурированным изложением, комментариями, поясняющими сущность и назначение употребляемых процедур и данных. Комментарии, заключаемые для отделения от основного текста программы в круглые скобки, могут находиться в любом месте: после, перед и между словами. Рекомендуется использовать комментарии для объяснения смысла идентификаторов, а не пытаться передать этот смысл при помощи хитроумной мнемоники. Для часто употребляемых операций выбраны короткие обозначения (одна-две литеры), хотя и не всегда мнемоничные: мнемоника для постоянно применяемых обозначений не нужна, а краткость существенна. Для менее употребительных операций сохранена общепринятая мнемоника, например: AND, NOT, NEG, EXEC и т. п.

Определение процедуры включает в качестве обязательных следующие комментарии:

— прежде всего содержательную характеристику определяемой процедуры.

— перед телом определения показ исходного состояния затрагиваемой процедурой части стека операндов;

— по окончании тела определения показ той же части стека в том состоянии, которое она приобретет в результате применения определяемой процедуры;

— содержательную характеристику используемых в теле определения, но еще не определенных процедур.

Кроме перечисленных, могут быть включены комментарии, показывающие состояние стека операндов в процессе выполнения определяемой процедуры, содержащие пояснения к выбору процедур по условию, указывающие назначение отдельных фрагментов тела определения и т. п. Надлежащим образом комментированная программа должна быть понятна без дополнительных пояснений, описаний и блок-схем.

ДССП-процессор представляет собой управляемую входным потоком команд единую систему, осуществляющую все необходимые для создания, хранения и выполнения программ функции. Создаваемая в виде исходного файла программа, наряду с определениями процедур и данных, может включать команды управления процессором, в частности, его подсистемами, такими как редактор текстов, манипулирование файлами, работа с периферийными устройствами и т. п.

Реализуя программу, процессор заносит в свой словарь имена определенных в ней процедур и данных, компилирует (переводит во внутреннее представление) и запоминает соответствующие этим именам тела определений, находит в словаре и выполняет встретившиеся в тексте программы слова и словосочетания (процедуры, команды, операции). При этом базовые операции процессора, определенные в машинном коде, выполняются непосредственно, а скомпилированные определения интерпретируются.

В качестве внутреннего представления программы в ДССП используется *процедурный* («сшитый») код, заключающийся в том, что внешним именам процедур и данных сопоставлены начальные адреса соответствующих тел, причем сами тела представляют собой последовательности адресов, указывающих начала процедур, использованных в этих телах. Таким образом, внутреннее представление, во-первых, однозначно соответствует исходному (внешнему) представлению программы, а во-вторых, оно весьма компактно. Последнее означает как компактность самой ДССП, что позволило сделать ее резидентной в главной памяти, так и компактность создаваемых в ней программ.

§ 2. ДССП в режиме микрокалькулятора

Ознакомление с ДССП естественно начать с минимума возможностей системы, соответствующего программируемому микрокалькулятору. Для этого надо ограничиться употреблением только тех команд, которые входят в типичный набор команд микрокалькулятора, и пользоваться ими в режиме непосредственного выполнения с клавиатуры. ДССП-процессор при этом будет действовать как постфиксный программируемый микрокалькулятор, оперирующий с целыми числами. Слово «постфиксный» указывает на то, что команды должны задаваться в постфиксной форме, т. е. символ операции должен вводиться *после* операндов, например: команду 2 + 3 следует вводить как 2 3 +. Оперирование с целыми числами означает, что и исходные данные, и результаты операций являются целыми числами.

Будучи загруженной в память компьютера, ДССП выдает на дисплей в первой позиции очередной строки символ *, означающий готовность принимать предписания, которые можно вводить с клавиатуры. Таким же образом система сигнализирует о выполнении полученных предписаний и о готовности принимать дальнейшие. Короче говоря, звездочка в начале очередной строки разрешает ввод с клавиатуры.

Ввод предписания осуществляется в виде последовательности *слов*. Слово может состоять из одной или нескольких литер (букв, цифр, специальных знаков) и отделяется от соседних слов пробелами. Слово, состоящее только из цифр и, может быть, имеющее в начальной позиции знак «минус», воспринимается процессором как число, подлежащее засылке в стек операндов. Слово, не являющееся числом, либо означает одну из операций, которые может выполнять процессор, либо оказывается неизвестным процессору. В последнем случае процессор, не найдя заданного слова в своем словаре, выдаст на дисплей сообщение: «Не знаю...», в котором вместо многоточия будет напечатано это слово.

Для опробывания процессора в режиме калькулятора достаточно слов +, —, *, /, обозначающих арифметические операции, слова . (точка), вызывающего копирование на дисплей числа, которое находится в вершине стека операндов, слова D, вызывающего удаление верхнего элемента стека (delete — удалять), а также клавиш ⟨пробел⟩ и ⏎. Клавиша ⟨пробел⟩ оставляет незаполненный печатным знаком промежуток, используемый для разделения слов. Клавиша ⏎ («возврат каретки») воспринимается процессором как символ конца вводимого предписания и вместе с тем как команда выполнять это предписание. Подобно пробелу, ⏎ является также ограничителем слова, поэтому между ⏎ и непосредственно

предшествующим ему словом вводить пробел нет необходимости.

Чтобы процессор вычислил и выдал на дисплей сумму двух чисел, например, 234 и 56, надо ввести эти числа, отделив их друг от друга пробелом, затем, разграничивая пробелами, ввести слова-литеры + и . (сложить и выдать копию результата) и закончить предписание нажатием клавиши ⏎. На дисплее все эти действия отобразятся в следующем виде (символ ⏎ означает нажатие клавиши «возврат каретки», на дисплее не отображается):

```
* 234 56 + . ⏎ 290
*
```

Вводимое с клавиатуры предписание отображается, за исключением ⏎, на дисплее и запоминается в буфере на входе процессора в виде входной строки. При нажатии клавиши ⏎, произведенном после ввода точки, процессор приступает к обработке входной строки. Числа 234 и 56 засылаются друг за другом в стек операндов. Затем выполняется операция +, состоящая в том, что из стека изымаются два числа, производится сложение этих чисел и полученная сумма засылается в стек. Далее выполняется операция «точка», которая выдает копию содержимого верхнего элемента стека, т. е. хранящегося в нем значения суммы, на дисплей. Это значение, равное 290, печатается в той же строке вслед за обработанным предписанием, после чего производится переход на новую строку и печатается звездочка, символизирующая готовность процессора получить новое предписание.

Общее правило, регламентирующее работу стекового (постфиксного) процессора, состоит в том, что каждый встретившийся при обработке входной строки операнд заносится в стек, а каждая операция выполняется процессором, потребляя из стека требующиеся ей операнды и засылая в стек значение или значения результата. Исключением является операция «точка», относящаяся к средствам отладки. Она не изымает из стека операнда, а лишь позволяет заглянуть в вершину стека, ничего в нем не нарушая. Следующий пример иллюстрирует это.

```
* 2 4 6 . D . D . ⏎ 6 4 2
*
```

Клавиша ⏎ нажата после ввода третьей точки. Процессор заслал в стек последовательно 2 4 6. Первая точка вызвала на дисплей копию содержимого вершины стека — числа 6. Затем операция D удалила верхний элемент стека и вершиной стал элемент, содержащий число 4. Вторая точка скопировала 4, а следующая за ней операция D удалила верхний элемент, после чего вершиной стал элемент, содержащий 2. Третья точка вызвала копию 2 на дисплей. В данном

примере наглядно проявилась характерная особенность стека — изъятие элементов из него происходит в последовательности, обратной той, в которой они были занесены.

Операция «точка» позволяет скопировать на дисплей значение вершины. В нашем примере для просмотра других элементов стека пришлось удалять находящиеся над ними элементы. Однако в ДССП имеется отладочная операция .. (точка-точка) позволяющая просмотреть содержимое стека на любую глубину, ничего в нем не нарушая. Выполняя эту операцию, процессор сначала выдает на дисплей число содержащихся в стеке элементов (глубину стека), а затем последовательность их значений, начиная с вершины. Если число элементов в стеке превышает 6, то после выдачи очередной шестерки значений для продолжения просмотра надо нажать клавишу ‹пробел›, а для прекращения — клавишу ВК.

При выполнении некоммутативных операций первым операндом считается поступивший в стек ранее другого и находящийся в подвершине, а вторым — поступивший после первого и находящийся в вершине. Так, вычитание 95—352 выполняется в виде

```
* 95 352 — . ВК —257
*
```

Операция деления / потребляет содержимое вершины стека в качестве делителя, содержимое подвершины — в качестве делимого и засылает в стек сначала частное, затем остаток. Например, деление 67/4 с отображением остатка и частного на дисплей будет выполнено так:

```
* 67 4 / . D . ВК 3 16
*
```

Нажатие клавиши ВК произведено после ввода второй точки. Операция / изымает из стека делитель 4, делимое 67 и засылает в стек частное 16 и остаток 3. Первая точка выдает на дисплей содержащийся в вершине стека остаток. Операция D удаляет из стека верхний элемент, вследствие чего вершиной становится элемент, содержащий частное. Вторая точка копирует на дисплей значение частного. При необходимости удалить из стека частное следует ввести операцию D после второй точки.

Благодаря наличию стека, ДССП-процессор способен автоматически вычислять арифметические целочисленные выражения произвольного вида, вводимые в постфиксной форме. Например, выражение (20 + 24/9) * (67—92) вычисляется так:

```
* 20 24 9 / D + 67 92 — * . D ВК —550
*
```


После засылки первых трех чисел, выполнения деления и удаления остатка в подвершине стека находится число 20, в вершине — частное 2. Операция + изымает эти числа и засылает в стек их сумму 22. Затем засылаются числа 67 и 92, а операция вычитания изымает их и засылает разность —25. Теперь стек содержит в подвершине 22, а в вершине —25. Операция * потребляет эти числа в качестве сомножителей и засылает в стек их произведение —550. Операция «точка» выдает на дисплей копию этого результата, а операция D удаляет его из стека.

Помимо рассмотренных операций, в режиме микрокалькулятора могут использоваться имеющиеся в ДССП операции копирования (дублирования) и обмена элементов стека. Добавление этих операций значительно расширяет возможности манипулирования данными в стеке.

Операция копирования (дублирования) вершины стека C (copy — копировать) производит засылку в стек копии текущего значения его вершины. Это равносильно дублированию верхнего элемента стека: прежняя вершина становится подвершиной, а ее копия — новой вершиной. Покажем применение этой операции на примере вычисления многочлена $2x^2 + 3x - 5$ при условии, что значение x содержится в вершине стека.

```
* C 2 * 3 + * 5 — ВК
*
```

Операция C дублирует обладающий значением x верхний элемент стека. Дубликат (новая вершина) потребляется первой операцией умножения, а оригинал — второй. В результате вычисления содержавшееся в вершине исходное значение x будет заменено вычисленным значением многочлена. На дисплей копия результата не выдается, поскольку предписание не содержит операции «точка».

Наряду с операцией C копирования вершины стека в ДССП имеются также операции C2, C3, C4, копирующие элементы, находящиеся соответственно на глубине 2 (подвершина), 3, 4. Имеется также операция CT копирования элемента, находящегося на глубине, которая указана в вершине (top) стека. Выполняя CT, процессор изымает из стека верхний элемент, использует его значение как указатель глубины «залегания» копируемого элемента и засылает копию последнего в стек. Так, копирование элемента, находящегося на глубине 5, задается словосочетанием 5 CT, реализуя которое, процессор зашлет в стек число 5, а затем выполнит операцию CT.

Операции обмена E2, E3, E4 (exchange — обменять) производят перестановку 1-го (верхнего) элемента стека соответственно со 2-м, 3-м, 4-м, т. е. находящимся на глубине 2, 3, 4, элементом. Для обмена на большую глубину служит операция ET, употребляющая,

так же как CT, значение вершины в качестве указателя глубины залегания элемента, который обменивается с 1-м элементом. Например, чтобы элемент, находящийся на глубине 7, занял положение вершины, а элемент, являющийся вершиной, оказался на глубине 7, выполняют 7 ET.

§ 3. Процедуры, структурирование программы

В качестве основного приема программирования ДССП предоставляет пользователю возможность определять поименованные последовательности операций, называемые далее процедурами. Пусть требуется, например, вычислять значения квадратного трехчлена $3x^2 - 4x + 9$ для задаваемых значений x. В таком случае следует определить процедуру, реализующую формулу трехчлена, выдачу на терминал копии результата и удаление верхнего элемента стека, а затем применять эту процедуру к конкретным значениям x. Искомая процедура, назовем ее PX, определяется следующим образом:

```
: PX [X] C 3 * 4 — * 9 + [3*X*X—4*X+9] . D [ ] ;
```

Двоеточие означает операцию «определить процедуру», причем имя процедуры следует за двоеточием после разделительного пробела. Определяющая последовательность операций (тело процедуры) располагается вслед за именем процедуры и заканчивается точкой с запятой. Короче, процедура определяется в форме:

: ⟨имя процедуры⟩ ⟨тело процедуры⟩;

В квадратные скобки заключены комментарии, в частности, отражающие текущее состояние стека: вначале в стеке находится заданное значение X, после выполнения вычислений — полученное значение трехчлена, а после копирования на терминал и удаления стек пуст.

Рассматриваемую процедуру можно прокомментировать более подробно, отслеживая состояние стека при каждой операции

```
: PX [X] C [X, X] 3 [X, X, 3] * [X, X*3]
    4 [X, X*3, 4] — [X, X*3 — 4] * [X*X*3—X*4]
    9 + [X*X*3—X*4 + 9] . D [ ] ;
```

Комментарии помогают человеку понять и использовать процедуру, процессор же просто игнорирует все, что заключено в скобки. Поэтому при вводе определения процедуры непосредственно в ДССП-процессор комментарии следует опускать, иначе текст определения не уложится во входной строке, длина которой — 80 литер, включая пробелы.

После того как определение процедуры введено и нажатием клавиши ВК процессору сообщено о конце ввода, на экране терминала появляется звездочка, сигнализирующая о выполнении команды «определить процедуру» и готовности процессора продолжить диалог. Теперь можно применить процедуру PX к задаваемым с клавиатуры значениям X, например, к 2, 3, 4:

```
* 2 PX ВК 13
* 3 PX ВК 24
* 4 PX ВК 41
*
```

Определим более общую процедуру вычисления трехчлена вида $a_2x^2 + a_1x + a_0$, позволяющую задавать значения как x, так и a_0, a_1, a_2. Назовем ее PXA:

```
: PXA [A0,A1,A2,X] C E4 E3 [A0,X,A1,X,A2] * +
      [A0, X, A2 * X+A1] * +[A2*X*X+A1 * X+A0] ;
```

При использовании PXA в стеке должны находиться в требуемой последовательности значения a_0, a_1, a_2, x. Например: $a_0 = 1$, $a_1 = 2$, $a_2 = -3$, $x = 4$:

```
* 1 2 —3 4 PXA . D ВК —39
*
```

В теле процедуры, наряду с базовыми операциями процессора, могут находиться процедуры, определяемые пользователем. Например, можно определить процедуру P, которая дополнительно к вычислениям, выполняемым PXA, будет выдавать копию результата на терминал и удалять результат из стека.

```
: P [A0, A1, A2, X] PXA [A2 * X * X+A1 * X+A0] . D [ ] ;
```

В частности, тело процедуры может включать имя самой определяемой процедуры, т. е. процедура может быть рекурсивной. Пример:

```
: TIME [T] 1 + [T+1] TIME ;
```

Эта процедура увеличивает на 1 значение вершины стека и снова обращается к себе же, т. е. работает как счетчик времени.

Счетчик TIME в принципе не может остановиться: процедура прибавления единицы будет выполняться снова и снова, пока не будет исчерпана отведенная для гнездования процедур память. Но в ДССП имеются средства, позволяющие управлять ходом процесса в зависимости от получаемых результатов — операции управления ходом программы.

Чтобы сделать программу легко обозримой (читабельной) и понятной, ее разбивают на обладающие определенным смыслом

поименованные части — *процедуры*, определяемые каждая своей последовательностью процедур, которые в свою очередь определены последовательностями более мелких процедур и т. д. до процедур, определяемых непосредственно последовательностями базовых операций ДССП. Такая программа, записываемая в виде иерархии определений процедур, называется *структурированной*. Метод построения структурированной программы, заключающийся в постепенном разложении решаемой задачи на все более мелкие подзадачи, как известно, называется структурированным программированием. Создание методом структурированного программирования любых программ возможно при наличии операций выполнения процедуры по условию, повторения процедуры и выходе из повторяемой процедуры. Имеющийся в ДССП набор операций этого рода обеспечивает возможность структурированного построения произвольной программы.

Условия выполнения или невыполнения процедуры формулируются относительно знака числа, точнее, относительно знака значения, которым обладает в текущий момент вершина стека. Основная операция условного выполнения процедуры — BRS (branch on sign — ветвиться по знаку) предписывает выполнить одну из трех названных вслед за BRS процедур в зависимости от знака текущего значения вершины стека. Выполняя BRS, процессор изымает из стека верхний элемент, тестирует его значение и, если оно отрицательно, то выполняет первую из названных процедур, если равно нулю, то — вторую, а если положительно, то — третью. Так, словосочетание

```
BRS N Z P
```

вызовет удаление из стека одного элемента и выполнение процедуры N, если значение этого элемента отрицательно, выполнение процедуры P, если положительно, и выполнение процедуры Z, если равно нулю.

Примером использования операции BRS служит следующее определение процедуры SGN (sign — знак):

```
: SGN [X] BRS -1 0 1 [SGN(X)] ;
```

Эта процедура заменяет содержащуюся в вершине стека величину X числом -1, если $X < 0$, числом 0, если $X = 0$, и числом 1, если $X > 0$. Процедура SGN имеется в ДССП в качестве базовой операции процессора.

Операция BRS, наряду с выбором одной процедуры из трех данных, обеспечивает возможность реализации двузначных операторов вида IF-THEN и IF-THEN-ELSE. Например, предложение

if $x \geqslant 0$ **then** P1 **else** P0

реализуется словосочетанием BRS P0 P1 P1, а предложение **if** $x \neq 0$ **then** P — словосочетанием BRS P NOP P, в котором NOP — имя пустой операции. Но в ДССП имеется более эффективная реализация двузначных условий — операций IF—, IF0, IF+, BR—, BR0, BR+.

Операции группы IF соответствуют оператору IF-THEN. Например, словосочетание IF— P предписывает изъять из стека верхний элемент и тестировать его знак, причем, если этот элемент имеет знак минус, то выполнить процедуру P. Словосочетания IF0 P и IF+ P предписывают выполнить процедуру P соответственно в случае, когда изъятый элемент равен нулю, и в случае, когда его значение положительно.

Операции BR—, BR0 и BR+ соответствуют оператору IF-THEN-ELSE, предписывая выбирать одну из двух называемых в них процедур. Если знак изъятого из стека элемента совпадает с имеющимся в обозначении операции, то выполняется процедура, названная первой, а если не совпадает, то выполняется вторая процедура. Например, команда BR0 P0 P1 предписывает выполнить процедуру P0 в случае, когда изъятый из стека элемент равен нулю, а если это условие не удовлетворено, то выполнить процедуру P1.

Рассмотренные операции позволяют экономно запрограммировать выполнение процедуры в зависимости от данных условий. Наиболее часто встречающиеся условия вида $x < 0$, $x = 0$, $x > 0$ прямо реализуются операциями группы IF. Условия $x \geqslant 0$, $x \neq 0$, $x \leqslant 0$ программируются в виде операций BR—, BR0, BR+ с использованием в качестве первой процедуры пустой операции NOP. Например, предложению **if** $x \leqslant 0$ **then** P соответствует команда BR+ NOP P.

В группу BR входит также команда BR, записываемая в виде:

```
BR A1 P1 A2 P2 ... AK PK ... AN PN ELSE P0
```

Реализуя эту команду, процессор сначала выполняет процедуру-указатель A1 и сравнивает занесенное ею в стек значение с находящимся под ним значением прежней вершины стека. Если значения совпали, то из стека удаляются два верхних элемента и выполняется сопоставленная указателю A1 процедура P1, после чего остальная часть команды BR пропускается и выполняется следующая за ней команда. Если же сравниваемые значения не совпали, то из стека удаляется один верхний элемент и те же действия производятся с парой A2 P2, затем, если совпадения не получилось, то с парой A3 P3 и т. д. по AN PN включительно. В случае, когда ни одна из N попыток не дала совпадения, выполняется названная после слова ELSE процедура P0. В роли процедур-указателей могут выступать

просто числовые константы, например:

```
BR 5 P1  —3 P2  0 P3  25 P4 ELSE P0
```

В качестве иллюстрации того, как используются операции условного выполнения процедур, осуществим модификацию приведенной в предыдущем разделе процедуры TIME с тем, чтобы счетчик останавливался по заданному условию:

```
: TIME [T] 1 + [T+1] C IF— TIME [0] ;
```

Теперь процедура TIME вызывает себя только при отрицательном значении вершины стека. Счетчик сработает ровно N раз, если к началу первого выполнения TIME вершина содержит отрицательное число —N. Например, чтобы получить 7 срабатываний, надо задать

```
—7 TIME ⏎
```

Поскольку IF— в определении TIME, как и всякая условная операция, изымает из стека тестируемый элемент, а элемент этот необходим для последующих операций, то его приходится дублировать, помещая перед IF— операцию C.

Многократное выполнение процедуры программируется также при помощи операций RP (repeat — повторять) и DO (do — делать, выполнять).

Команда RP F предписывает выполнять процедуру F снова и снова неограниченное число раз. Чтобы повторения могли прекратиться, тело процедуры F должно содержать операцию EX (exit — выйти), выполняемую при заданном условии. Операция EX осуществляет переход к выполнению процедуры, которая следует по тексту программы за повторяемой процедурой, содержащей эту операцию EX. Так, счетчик, реализованный выше в виде рекурсивной процедуры TIME, можно запрограммировать как повторение процедуры F, которая определена так:

```
: F [T] 1 + [T+1] C IF0 EX [T+1] ;
```

Чтобы счетчик сработал 25 раз, надо задать

```
—25 RP F ⏎
```

Наряду с операцией EX, которая употребляется в командах выполнения по условию, имеются операции условного выхода EX—, EX0, EX+, производящие тот же эффект, что и команды IF— EX, IF0 EX, IF+ EX, т. е. потребляющие из стека верхний элемент, тестирующие его знак и выполняющие выход, если знак совпадает с указанным в обозначении операции.

Операция DO вызывает повторение названной вслед за ней процедуры N раз, где N — число, содержащееся в вершине стека

к началу выполнения DO. Например, чтобы процедура P выполнилась 8 раз, надо задать

```
8 DO P ⏎
```

Если в теле процедуры P имеется хотя бы одна операция выхода и условие ее выполнения окажется удовлетворенным до того, как произойдет заданное число повторений, то повторения будут прекращены путем выхода из процедуры подобно тому, как это делается в случае операции RP. Например, при повторении посредством DO описанной выше процедуры F, в определении которой содержится IF0 EX, запись

```
30 DO F ⏎
```

вызовет 30 выполнений F, если текущее значение вершины $T < -30$ или $T \geqslant 0$. Если же $-30 \leqslant T < 0$, то процедура F выполнится $-T$ раз.

Повторяемые процедуры могут содержать в своих телах операции RP и DO, приводящие к возникновению вложенных циклов, причем глубина вложенности допустима любая. При этом имеется операция EXT выхода из вложенного цикла с указанием глубины вложенности в вершине стека. Например, выход с глубины 2 по отрицательному значению вершины можно задать так:

```
2 E2 BR— EXT D ⏎
```

Здесь E2 осуществляет обмен 1-го и 2-го элементов стека с тем, чтобы используемое EXT число 2 оказалось подвершиной, которую использует BR— .

Операция EXT с заданной глубиной 1 равносильна EX.

§ 4. Операции преобразования данных

Элементы стека операндов ДССП — это двоичные 16-битные слова, интерпретируемые тем или иным образом в зависимости от выполняемых над ними операций. Наиболее общей является интерпретация элемента стека как булевского вектора A (15 : 0). Убывающая нумерация компонент вектора повторяет принятую для машин унифицированной архитектуры нумерацию битов машинного слова. Компоненты булевского вектора (биты) принимают значения 0 и 1.

К операциям, непосредственно выполняемым над булевскими векторами, относятся:

— побитная инверсия INV, изменяющая значение каждого бита вектора, т. е. заменяющая 0 на 1, а 1 на 0;

— побитная конъюнкция & устанавливающая в i-м бите результата, $i = 15, 14, \ldots, 0$, значение 1, если i-е биты обоих

операндов равны 1, а в прочих случаях полагающая i-й бит равным 0;

— побитная дизъюнкция &0, устанавливающая в i-м бите результата, i = 15, 14, . . ., 0, значение 0, если i-е биты обоих операндов равны 0, а в прочих случаях полагающая i-й бит равным 1;

— побитное сложение '+' (побитная «разделительная дизъюнкция» или «неэквивалентность») устанавливает в i-м бите результата значение 0, если i-е биты обоих операндов имеют одинаковые значения, и устанавливает значение 1, если i-й бит одного операнда не равен i-му биту другого операнда;

— сдвиг влево SHL — каждый бит вектора, начиная с 15-го, принимает значение следующего за ним в порядке убывания номеров, а последний, нулевой, бит принимает значение 0;

— сдвиг вправо SHR — каждый бит вектора, начиная с нулевого, принимает значение следующего за ним в порядке возрастания номеров, а 15-й бит принимает значение 0;

— сдвиг по вершине SHT — верхний элемент изымается из стека и рассматривается как целое число N, указывающее сколько сдвигов и в каком направлении надо произвести в вершине стека: при $N > 0$ производится сдвиг влево, при $N < 0$ — вправо.

Ввод и вывод значений булевских векторов, равно как и числовых значений, можно производить в двоичном, восьмеричном и шестнадцатеричном коде, устанавливая желаемый режим соответственно командами B2, B8, B16 (base 2, 8, 16 — основание 2, 8, 16). По умолчанию нормально имеет место десятичный режим, т. е. вводимые последовательности цифр (возможно, со знаком) воспринимаются процессором как целые десятичные числа, а при выводе элементы стека интерпретируются как числа, представленные в дополнительном коде, и переводятся в прямой десятичный код. Для перехода из недесятичного режима в десятичный выполняется команда B10.

Числовая интерпретация элементов стека основана на дополнительном двоичном коде. Слово A(15 : 0) при A(15) = 0 истолковывается как неотрицательное число, равное

$$A(14)\cdot 2^{14} + A(13)\cdot 2^{13} + \ldots + A(1)\cdot 2 + A(0).$$

При A (15) = 1 то же слово представляет собой отрицательное число, равное

$$-2^{15} + A(14)\cdot 2^{14} + A(13)\cdot 2^{13} + \ldots + A(1)\cdot 2 + A(0).$$

Таким образом, значения слова понимаются как целые числа на интервале от -2^{15} до $+2^{15} - 1$ (от —32768 до +32767). Если в результате выполнения операции получается число, находящееся вне данного интервала, то оно автоматически заменяется сравнимым по модулю 2^{16} числом из этого интервала. Другими словами, в представлении результата все биты старше 15-го (т. е. отсутствующие в 16-битном слове) считаются равными нулю. Например, сумма двух максимальных на рассматриваемом интервале чисел $(2^{15} - 1) + (2^{15} - 1) = (2^{16} - 2)$ получается равной —2. Во избежание подобных искажений, надо следить, чтобы результаты операций не выходили за пределы указанного интервала представимых целых чисел.

В ДССП имеются арифметические операции сложения +, вычитания —, умножения *, деления /, выбора большего MAX и меньшего MIN из двух верхних элементов стека, изменения знака числа NEG, приращения и убавления на единицу 1+ и 1—, приращения и убавления на два 2+ и 2—, а также операции T0 и T1, заменяющие значение вершины стека соответственно нулем и единицей.

Операции сравнения $<$, $=$, $>$ определены над числами и в качестве результата выдают числовые значения 0 и 1. Так, операция $<$ потребляет из стека два элемента и засылает в стек число 1, если значение верхнего элемента оказалось больше значения нижнего, а в противном случае засылает 0. Например, в результате выполнения последовательности 5 —20 $<$ в стек будет заслан 0. Операция $=$ засылает 1 в случае равенства потребленных ею элементов. Операция $>$ засылает 1, когда нижний элемент больше верхнего.

К операциям сравнения примыкают одноместные операции отрицания NOT и тестирования знака числа SGN. Операция NOT заменяет значение вершины стека, не равное нулю, нулем, а равное нулю — единицей. Операция SGN заменяет отрицательное значение вершины стека на —1, а положительное — на 1.

Использование операций сравнения в совокупности с операциями выполнения процедур по условию существенно расширяет возможности программирования условий выполнения процедуры или выбора одной из данных процедур. Например, предложение **if** $x \leqslant a$ **then** P с помощью этих средств программируется в виде:

X A > IF0 P

Операция отношения X A > дает в результате 0, если $X \leqslant A$, и в этом случае выполняется P. Другой пример: **if** $x < a$ **then** P1 **else** P2 можно запрограммировать как

X A < BR+ P1 P2

Дальнейшее расширение возможностей программирования условий обеспечивается применением, наряду с операциями отношений, логических операций конъюнкции &, дизъюнкции &0 и отрицания NOT. Например, предложение **if** $a < x$ & $x < b$ **then** P

программируется с использованием конъюнкции так:

```
A X < X B < & IF+ P
```

Предложение if a ⩽ x & a < b then P можно запрограммировать одним из следующих двух равноценных в отношении эффективности способов:

```
1) A X > NOT X B < & IF+ P
2) A X > X B < NOT &0 IF0 P
```

Во втором случае операция &0 дает в результате 0 только тогда, когда оба ее операнда равны 0, фиксируя таким образом «совпадение нулей».

При использовании операции тройственного выбора BRS естественно ввести третье логическое значение —1, получив его путем изменения знака значения 1, доставляемого операциями отношений. Например, предложение if x < a then P1 else if x = a then P2 else P3 при помощи BRS можно запрограммировать в виде:

```
X A < NEG X A > + BRS P1 P2 P3
```

Впрочем, данное предложение может быть запрограммировано короче и эффективней, но без операций отношения, т. е. без явного задания условий:

```
X A — BRS P1 P2 P3
```

§ 5. Именуемые данные

Стек операндов является основным, но не единственным механизмом манипулирования данными в ДССП. Имеется также возможность, наряду с определениями процедур, объявлять элементы и стандартно организованные совокупности элементов (так называемые структуры) данных, доступные затем для использования по их именам. Реализуя объявления данных, процессор резервирует требующуюся для их хранения память и обеспечивает необходимые механизмы доступа к этой памяти.

Базовый язык ДССП включает ряд рассмотренных ниже словдиректив для объявления переменных и массивов. В порядке расширения языка системы в него могут быть введены другие слова этого рода и, соответственно, другие элементы и структуры данных.

Слово VAR объявляет 16-битную числовую переменную. Например, запись

```
VAR X
```

объявляет переменную X, т. е. сообщает процессору, что имя X есть имя переменной. Процессор связывает с этим именем 16-битную ячейку памяти, в которой будет храниться значение данной переменной. Команда присваивания переменной X значения, которое содержится в вершине стека операндов, имеет вид

```
! X
```

Выполняя эту команду, процессор изымает из стека верхний элемент и записывает его значение в отведенную для переменной ячейку.

Команда, состоящая только из имени переменной, перед которым нет литеры !, вызывает засылку в стек значения этой переменной, причем засылка производится путем копирования содержимого соответствующей ячейки памяти, т. е. значение переменной остается неизменным. Таким образом, всякое вхождение в программу имени переменной X, если ему непосредственно не предшествует слово, предписывающее иное действие, будет засылать в стек текущее значение этой переменной, подобно тому, как засылаются непосредственно заданные числа (числовые литералы).

Слово VCTR объявляет одномерный массив (вектор) 16-битных ячеек, причем номер старшего элемента этого массива задается значением вершины. Например, в результате выполнения записи

```
9 VCTR ROW
```

процессор резервирует 10 последовательно адресуемых ячеек памяти, образуя вектор ROW (0 : 9). Сначала производится засылка числа 9 в стек, а затем выполняется процедура VCTR, употребляющая верхний элемент стека для определения длины создаваемого вектора ROW.

Засылка в стек значения j-го элемента вектора ROW, $0 \leqslant j \leqslant 9$, задается командой

```
J ROW
```

Переменная J засылает в стек номер элемента, а имя вектора ROW вызывает замену в вершине стека этого номера значением соответствующего элемента. Если же непосредственно перед именем вектора стоит литера !, то указываемому вершиной элементу данного вектора присваивается значение подвершины, и глубина стека уменьшается на 2.

Аналогично работает одномерный массив байтов, объявляемый словом BVCTR — различие только в том, что элементами массива являются не 16-битные ячейки, а 8-битные байты.

Имеется также возможность создания соответствующих векторов-констант, т. е. вектора 16-битных или 8-битных чисел, значения которых определены при его объявлении и в дальнейшем не изменяются. Так, вектор 16-битных констант ABC длины L + 1

объявляется с помощью слова CNST в виде

```
CNST ABC k0 k1 . . . kL;
```

где k0, k1, . . ., kL — числовые литералы.

Объявление вектора констант-байтов аналогично, но вместо CNST содержит BCNST, а вместо 16-битных чисел — байты.

Многомерный массив 16-битных ячеек объявляется с помощью слова ARR, перед которым указываются максимальные значения индекса по каждому измерению и число измерений. Например, трехмерный массив TIR (0 : 8, 0 : 2, 0 : 24) объявляется так:

```
8 2 24 3 ARR TIR
```

Число 3, находящееся непосредственно перед ARR, означает размерность объявляемого массива.

Засылка элемента массива в стек достигается заданием индекса этого элемента в сопровождении имени массива. Например, команда засылки в стек элемента TIR(0,2,2) выражается в виде

```
0 2 2 TIR
```

Соответственно, присваивание этому элементу текущего значения вершины стека задается командой

```
0 2 2 ! TIR
```

Рассмотренные средства обеспечивают возможность именования данных и манипулирования данными независимо от адресной системы компьютера. Но в базовый язык входят, кроме того, средства, позволяющие манипулировать адресами элементов памяти. Адрес переменной или элемента массива X засылается в стек командой

```
' X
```

В случае с элементом массива этой команде предпосылается значение индекса.

Операция @ заменяет находящийся в вершине стека адрес ячейки памяти значением, которое содержит эта ячейка. Операция B @ заменяет адрес значением соответствующего байта, полагая старший байт вершины стека равным нулю. Операция !T записывает в ячейку по адресу, изъятому из вершины стека, значение, изымаемое из подвершины. Операция !TB осуществляет аналогичную запись младшего байта подвершины в байт, адресуемый вершиной.

Операция ' ' засылает в стек начальный адрес тела процедуры, поименованной следующим за ' ' словом программы. Выполнение процедуры, начальный адрес которой содержится в вершине стека, вызывает команда EXEC, потребляющая верхний элемент стека и инициирующая указываемую им процедуру.

Текстовый литерал, т. е. заключенный в кавычки текст, записывается в память компилятором в виде последовательности байтов, причем доступ к этой последовательности реализуется посредством ее дескриптора, ссылка на который замещает во внутреннем представлении программы вхождение литерала. В процессе выполнения ссылки в стек операндов засылается начальный адрес и длина текста без кавычек в байтах. Например, словосочетание «ТЕКСТОВЫЙ ЛИТЕРАЛ» будет записано в память в виде 17 байтов, а соответствующая ссылка (адрес дескриптора) вызовет засылку в стек адреса начального байта и числа байтов — 17.

Все эти и некоторые другие имеющиеся в базовом языке возможности манипулирования адресами предназначены не для основных пользователей системы — программистов-прикладников, а для системных программистов, которые будут производить надстройку системы и расширение ее языка.

§ 6. Операции ввода/вывода

Основным средством взаимодействия пользователя с ДССП является терминал, которым может служить электронно-лучевой дисплей с клавиатурой, электрическая пишущая машинка или телетайп. С терминала осуществляется первоначальный ввод, редактирование и отладка программ, подготовка данных и все управление системой. Программы и данные, а также сама ДССП, сохраняются в виде файлов на гибких дисках и на перфоленте и могут распечатываться на принтере. Для управления вводом/выводом в наборе базовых процедур ДССП имеются описываемые далее средства.

Программирование работы терминала обеспечивается операциями ввода и вывода чисел, литер и последовательностей литер (строк), а также выдачи пробела SP, перехода на начало новой строки CR, звукового сигнала BELL.

Операция TIB (Terminal Input Byte — ввод байта с терминала) инициирует цикл ожидания нажатия клавиши на клавиатуре терминала. При нажатии клавиши 7-битный код соответствующей литеры засылается в стек в виде младшего байта 16-битного слова, причем старшие 8 битов этого слова содержат нули. Копия введенной таким образом литеры отображается на дисплей. Имеется также операция TRB (Terminal Read Byte), отличающаяся от TIB тем, что засылка в стек кода введенной литеры не сопровождается отображением этой литеры на дисплей.

Операция TIN (Terminal Input Number — ввод числа с терминала) инициирует цикл ввода в стек и отображения на дисплей набираемого с клавиатуры числа. Вводимое число должно быть последовательностью цифр, которая может начинаться знаком «минус»,

а заканчиваться пробелом или ВК. В зависимости от установленного режима ввода/вывода цифры воспринимаются процессором как шестнадцатеричные, десятичные, восьмеричные или двоичные. Если шестнадцатеричное число начинается цифрой, обозначаемой буквой, то перед ней добавляется цифра нуль. Введенное число переводится в двоичный дополнительный код, который засылается в стек в качестве целочисленного значения 16-битного слова, т. е. с отсечением битов, расположенных левее имеющего вес 2^{15} старшего бита.

Каждая операция TIN вводит одно число. При наборе строки чисел, разделенных пробелами, количество этих чисел должно быть равным количеству операций TIN, выполняемых процессором для их ввода. Если чисел окажется меньше чем операций, процессор будет ждать ввода недостающих. Если же чисел больше чем операций, процессор продолжит выполнение программы, недоиспользовав набранную строку, причем оставшиеся числа могут быть введены встретившимися в процессе продолжения операциями TIN.

Последовательность, содержащая N набираемых с клавиатуры литер, вводится в память компьютера в виде N байтов, располагаемых по последовательно возрастающим адресам, начиная с адреса A, операцией TIS (Terminal Input String), перед которой указывается адрес A и число литер N. Например, при наличии объявленного вектора байтов X достаточной длины можно осуществить ввод, скажем, 9 литер, присвоив их значения элементам этого вектора, начиная с нулевого элемента, командой

```
0 ' X 9 TIS
```

Словосочетание 0 ' X засылает в стек адрес нулевого элемента массива X.

Аналогично операцией TOS задается вывод последовательности из N байтов-литер с начальным адресом A:

```
A N TOS
```

Вывод на терминал непосредственно включаемых в программу элементов текста обеспечивается конструкцией

```
." <текст>"
```

Например, чтобы при выполнении данного фрагмента программы на дисплее появился текст ВВЕДИТЕ НОМЕР ВАРИАНТА, программа должна содержать запись ."ВВЕДИТЕ НОМЕР ВАРИАНТА".

Операция TON (Terminal Output Number) выводит на дисплей число, изымаемое из подвершины стека, причем длина поля вывода должна быть задана в вершине. Выводимое число выравнивается по правому краю поля, свободные позиции слева заполняются пробелами, а если длина числа превышает заданную длину поля, то происходит отсечение слева. В режиме десятичного ввода/вывода отрицательные числа начинаются знаком минус.

Операция TOB (Terminal Output Byte) печатает литеру, код которой задан младшим байтом вершины стека. Глубина стека уменьшается на 1.

Имеются также операции прямого управления курсором дисплея или механизмом телетайпа:

CR — переход на начало новой строки,

SP — пробел, т. е. перемещение на одну позицию вправо.

Операция BELL вызывает короткий звуковой сигнал («звонок»).

Операции вывода на принтер подобны операциям вывода на терминал и базируются на аналогичной мнемонике, в которой литеры LP (Line Printer) либо заменены TO, либо добавлены в качестве ведущих. Например:

LPSP — пробел,

LPCR — переход на начало новой строки,

LPFF — переход на начало новой страницы,

LPN — вывод числа из подвершины стека в поле, длина которого задана вершиной,

LPB — вывод литеры, код которой содержится в младшем байте вершины,

A N LPS — вывод N байтов-литер из главной памяти, начиная с адреса A,

N LPT — перевод печатающей головки в позицию N текущей строки.

Длина текста, который может быть непосредственно воспринят ДССП-процессором с клавиатуры терминала, ограничена размером входной строки — 80 литер. Для ввода текстов большей длины используется специальная программа — *редактор текста*. Эта программа работает с отведенной ей областью главной памяти — *буфером редактора*, обладающим емкостью около 5 тыс. литер. Вводимый текст помещается в буфер и отображается на дисплее терминала. Команды редактора, отдаваемые с клавиатуры нажатием соответствующих клавиш редактирования, а также других клавиш, нажимаемых вместе с клавишей СУ (CTRL) позволяют эффективно редактировать и формировать находящийся в буфере текст, производя изменения, замены, удаления, вставки, перемещения и другие операции над произвольно задаваемыми его фрагментами. Вызов редактора текстов осуществляется командой E (edit — редактировать). Выход из редактора производится нажатием клавиши E при нажатой клавише СУ.

Подготовленный в буфере редактора текст можно использовать в различных целях следующим образом.

Командой PF (perform — выполнить) выдать содержимое буфера на вход ДССП-процессора в качестве входного потока команд. При этом процессор последовательно выполняет содержащиеся в тексте предписания, создавая требуемые структуры данных, компилируя определения процедур, производя другие заданные операции. Таким путем можно осуществлять отладку и корректировку формируемой в буфере программы или ее частей. Обнаруженные при попытке выполнения дефекты устраняются повторным редактированием находящегося в буфере исходного текста программы, исправленный текст снова выдается на выполнение и т. д. Разумеется, отладка должна быть организована рационально: попытке выполнить программу в целом должна предшествовать отладка ее частей, которые в свою очередь отлаживаются по частям — в этом одна из целей структурирования.

Команда LPE выдает содержимое буфера для распечатки на принтере, а команда PNE осуществляет выдачу на перфоленту. Соответственно имеется команда ввода с перфоленты в буфер редактора — RDE. Содержимое буфера можно также сохранить в файле N на гибком диске командой DOE N. Команда DIE N выполняет обратную функцию — ввод файла N с диска в буфер редактора. В версиях ДССП, работающих с магнитной лентой, имеются аналогичные команды вывода из буфера на ленту и ввода с ленты в буфер.

Возможность вводить тексты в буфер редактора с внешних носителей и выводить подготовленный или отредактированный текст из буфера на внешние носители, в частности, путем распечатки на принтере придает редактору характер достаточно универсального и мощного инструмента для формирования и обработки не только программ, но и других текстов различного назначения. Например, редактор можно с пользой употребить для создания всевозможной текстовой документации, составления отчетов, написания писем и т. п.

Для устройств перфолентного ввода/вывода, гибких дисков и магнитной ленты, помимо уже названных команд обмена с буфером редактора, предусмотрены команды ввода данных в программно адресуемую память и вывода из этой памяти, а также сохранения системы на внешнем носителе и загрузки программ с внешнего носителя для выполнения их процессором. Так, ввод N байтов в область главной памяти с начальным адресом A с перфоленты задается командой A N RDS, а с диска — соответственно командой A N DIS. Для вывода N байтов, начиная с адреса A, на перфоленту служит команда A N PNS, а на диск — команда A N DOS. Сохранение системы на диске осуществляется командой SAVE, на перфоленте — TSAVE. Загрузка файла N с диска производится командой LOAD N, загрузка с перфоленты — командой TLOAD. Ввод байта в вершину стека с перфоленты — RDB, с диска — DIB. Вывод байта из вершины стека на перфоленту — PNB, на диск — DOB. Кроме того, имеются специфические для каждого типа устройств вспомогательные команды, например: PNT — вывести на перфоленту T нулевых байтов где T — значение вершины стека, DNP — начальная установка диска и т. п.

§ 7. Управление словарем

Воспринимаемый базовым процессором входной поток команд, может, в частности, содержать команды определения процедур и описания именуемых данных, вызывающие компиляцию во внутреннее представление и запоминание тела процедуры или выделение памяти под указанные данные, а также занесение в *словарь* ДССП имени скомпилированной процедуры или структуры данных.

Словарь устанавливает соответствие между внешними (употребляемыми в исходном тексте программы) именами и адресами соответствующих этим именам объектов (тел процедур и описателей данных) во внутреннем представлении. При обработке определения процедуры или описания поименованного данного процессор наращивает словарь, образуя в нем новый словарный вход, содержащий имя (точнее, 7 первых литер имени) и адрес сопоставляемого этому имени тела процедуры или описателя данных.

В процессе компиляции словарь используется для перевода внешних имен в адреса обозначенных этими именами внутренних объектов. При этом последовательность имен, составляющая в исходном представлении, например, тело процедуры, превращается в последовательность адресов, которые служат ссылками на соответствующие тела процедур или описатели данных во внутреннем представлении. В случае непосредственного обращения к процедуре по ее внешнему имени с терминала или в потоке команд, поступающем на вход процессора из буфера редактора текстов или с внешнего носителя, осуществляется поиск данного имени в словаре и выполнение тела, начальный адрес которого содержится в найденном словарном входе.

В ходе наращивания словаря имеется возможность образования подсловарей — поименованных совокупностей словарных входов, доступ к которым для их наращивания или использования можно открывать и закрывать специальными командами, к числу которых относятся следующие.

GROW ¤V — наращивать подсловарь с именем ¤V (имена, присваиваемые подсловарям, начинаются литерой ¤), т. е. пока не будет предписано иное, вносить имена всех компилируемых процедур и данных в подсловарь ¤V;

USE ⌑V — открыть для использования (для поиска в нем имен) подсловарь ⌑V;

SHUT ⌑V — закрыть возможность использования подсловаря ⌑V;

ONLY ⌑V — сделать доступным для использования только подсловарь ⌑V;

CANCEL — отменить последнее ONLY.

Базовые процедуры ДССП составляют подсловарь с именем ⌑PRIME, открытый для использования и наращивания «по умолчанию», т. е. если не было команды, исключающей его использование или предписывающей наращивание иного подсловаря, а также если режим, установленный такой командой, отменен. В частности, до тех пор, пока не применены команды управления словарем, имена всех компилируемых процедур и данных заносятся в подсловарь ⌑PRIME и в нем же производится поиск по имени.

Имеются команды удаления из словаря той или иной совокупности словарных входов и, может быть, связанных с ними внутренних объектов. Так, команда FORGET ⌑V удаляет все имена, занесенные в словарь (не только в подсловарь ⌑V) после последнего выполнения команды GROW ⌑V вместе с обозначенными этими именами объектами, и отменяет установленное им наращивание подсловаря ⌑V. Команда ERASE удаляет все, что занесено с момента загрузки системы в память компьютера, т. е. приводит систему к виду, который она имела непосредственно после загрузки.

Сохранение модифицированной путем наращивания версии системы на внешнем носителе производится командой SAVE. Будучи затем загруженной в главную память, эта версия вместе с внесенными в нее добавлениями рассматривается как единое целое: команда ERASE загруженных добавлений не удаляет.

Ввиду того, что в готовой программе для подавляющего большинства процедур обращения по внешнему имени не требуется, предусмотрена возможность удаления из словаря словарных входов с сохранением сопоставленных им внутренних объектов. Команда CLEAR ⌑V удаляет из подсловаря ⌑V все словарные входы, за исключением порожденных компиляцией определений или описаний, перед которыми имеется команда :: (два двоеточия) — «Зафиксировать определяемое имя в словаре».

Особым случаем является обращение к процедуре обработки прерывания. Такая процедура определяется в виде:

INT : ‹имя процедуры› ‹тело процедуры›;

Затем ее можно связать с адресом вектора прерывания командой

‹адрес вектора› LINK ‹имя процедуры›.

На этом мы закончим описание базовых команд ДССП — более полный и систематически описанный набор их приведен в приложении. Теперь же, чтобы проиллюстрировать характер программирования в ДССП, рассмотрим пример программы, реализующей бесхитростный алгоритм сортировки — упорядочения одномерного массива байтов по возрастанию значений элементов. Просмотр элементов производится в порядке увеличения индекса, причем, если очередная пара смежных элементов не удовлетворяет условию упорядоченности: «Каждый элемент должен быть не меньше предшествующего ему», то члены пары меняются местами, и производится просмотр элементов в обратном направлении с попарными перестановками, пока не встретится упорядоченная пара, после чего возобновляется продвижение вперед с той позиции, из которой начался обратный просмотр.

Сначала мы построим процедуру BSORT, упорядочивающую массив байтов X(1:N), который создан программой, обращающейся к процедуре BSORT. Эта программа должна объявить переменную N, указывающую число элементов массива, и присвоить ей значение, объявить массив байтов N BVCTR X и присвоить значения его элементам. Определение процедуры BSORT программируется в следующем виде.

```
[BSORT — упорядочить вектор байтов X(1 : N) по возрастанию его
компонент]
:: : BSORT  [ ] START N DO FORWD [ ] ;
VAR J
[START — начальная установка, FORWD — шаг вперед, J — указатель компонент]
: START [ ] 0 0! X 0! J [ ] ;
[X(0) := 0 , наименьшее возможное значение; J := 0]
: FORWD [ ] PUSH [X(J), X(J + 1)] > IF+ ADJUST !1+ J [ ] ;
[PUSH — взять очередную пару компонент, если не упорядочена,
то ADJUST — упорядочивать ]
: PUSH [ ] J X [X(J)] J 1+ X [X(J), X(J + 1)] ;
: ADJUST ( ) J RP BACK [J] ! J [ ] ;
[BACK — шаг назад, упорядочивая ]
: BACK [ ] PUSH [X(J), X (J + 1)] > BR+ SWAP  EX !1— J [ ] ;
[если не упорядочена, то SWAP, иначе EX — выход, SWAP —
обменять значения компонент пары]
:SWAP [ ]PUSH [X(J), X(J + 1)] POP [ ];
[POP — вернуть пару в обратном порядке]
: POP [X(J), X(J + 1)] J ! X J 1+ ! X [ ] ;
```

Введем текст процедуры в буфер редактора и произведем отладку. Прежде всего следует воссоздать окружение, в котором будет

работать процедура — создать массив байтов X и переменную N, указывающую его длину. Поэтому тексту процедуры мы предпосылаем следующее:

```
VAR N 40 ! N
N BVCTR X
1 ' X N RDS [ввод массива X(1 : N) с перфоленты]
```

С помощью редактора помещаем эту вставку в буфер перед описанием процедуры BSORT.

Теперь можно покинуть редактор и командой PF выполнить содержащиеся в буфере предписания: создание переменной N со значением 40, создание вектора X и ввод с перфоленты значений его компонент, а также компиляцию определения процедуры BSORT и всей иерархии вложенных в нее процедур. Однако сделаем еще одно добавление, чтобы иметь возможность легко производить повторное выполнение содержимого буфера после исправления обнаруживаемых в процессе отладки ошибок. Введем для наших процедур подсловарь ⌧DEB и каждое новое выполнение содержимого буфера будем начинать «забыванием» того, что в нем наращено, и открытием его для наращивания заново. Это осуществляется при помощи еще одной вставки в самое начало буфера:

```
FORGET ⌧DEB
GROW ⌧DEB USE ⌧DEB
```

Перед первым выполнением содержимого буфера, т. е. перед первой командой PF, необходимо один раз ввести GROW ⌧DEB, чтобы могло выполниться находящееся в начале буфера FORGET ⌧DEB, а затем забывание/наращивание будет срабатывать автоматически от каждого PF.

Выполнение содержимого буфера теперь создает открытый для наращивания и использования подсловарь ⌧DEB, обеспечивающий доступ к переменным N, J, вектору X и иерархии процедур, возглавляемой BSORT. Ни одна из этих процедур в процессе выполнения команды PF не работает, поскольку в буфере имеются только их определения, но нет обращений к ним. В процессе отладки обращение к проверяемой процедуре производится вручную заданием ее имени. Например, чтобы проверить правильность функционирования процедуры START, следует выполнить ее при $X(0) \neq 0$, $J \neq 0$ и убедиться, что обе эти величины в результате выполнения процедуры приняли нулевое значение.

Отладку процедур естественно производить в такой последовательности, при которой процедура, содержащая обращения к другим процедурам, отлаживается после того, как отлажены все вложенные в нее процедуры. В нашем случае целесообразна следующая последовательность отладки: START, PUSH, POP, SWAP, BACK, ADJUST, FORWD, BSORT. При проверке каждой процедуры необходимо четко определить возложенные на нее функции и практически убедиться в том, что все эти функции реализованы верно. Только таким путем можно обеспечить надежность программы.

Завершив отладку процедуры и убедившись в ее правильности, можно воспользоваться редактором для удаления произведенных в целях отладки вставок и полученный таким образом чистый текст распечатать на принтере и/или сохранить в коде на диске или на перфоленте.

Существенным недостатком рассмотренной процедуры BSORT является то, что она определена применительно к конкретному имени вектора — X, иначе говоря, что имя вектора нельзя передать этой процедуре в качестве параметра, нельзя применить ее к вектору с другим именем. В ДССП этот недостаток преодолевается путем определений процедур, в которых вместо имен используются указатели (адреса) передаваемых объектов. В нашем случае это будет адрес нулевого элемента вектора. Процедуры, работающие с указателями, не используют имеющиеся в ДССП средства доступа к элементам массивов и поэтому более трудоемки и менее наглядны, но, как правило, являются более быстрыми.

Ниже приведен аналог процедуры BSORT, в котором доступ к вектору и его элементам осуществляется при помощи указателей. В отличие от предыдущей версии употреблены русские имена процедур и опущена часть комментариев. Указатель вектора X (адрес элемента X(0)) обозначен 'X по аналогии с операцией получения адреса переменной ' X. Обращение к процедуре применительно к имеющемуся в выполняемой программе вектору A длины N, не считая элемента A(0), производится в виде

```
N  0 ' A СОРТБАЙТ
```

```
[СОРТБАЙТ — упорядочить N байтов по возрастанию значений]
:: : СОРТБАЙТ [N, 'X] НАЧАЛО ['X, N] DO ВПЕРЕД D [ ] ;
: НАЧАЛО [N, 'X] 0 C2 !TB E2 ['X, 0, N] ;
: ВПЕРЕД ['X+J] ВЗЯТЬ ['X+J, X(J), X(J+1)] >
 IF+ ПРАВКА 1+ ['X+J+1] ;
: ВЗЯТЬ ['X+J] C @B C2 1+ @B ['X+J, X(J), X(J+1)] ;
: ПРАВКА ['X+J] C ['X+J, 'X+J] RP НАЗАД ['X+J,
 'X+K] D ['X+J] ;
: НАЗАД ['X+K] ВЗЯТЬ ['X+K, X(K), X(K+1)] >
 BR+ ОБМЕН EX 1— ['X+K—1] ;
: ОБМЕН ['X+J] ВЗЯТЬ ['X+J, X(J), X(J+1) ] ВЕРНУТЬ
 ['X+J] ;
: ВЕРНУТЬ ['X+J, X(J), X(J+1)] C3 !TB C2 1+ !TB
 ['X+J] ;
```

После отладки процедура может быть включена в какой-либо проблемно-ориентированный, а то и в базовый подсловарь. При этом с помощью операции CLEAR удаляются незафиксированные имена, точнее, соответствующие словарные входы, так что в словаре данная процедура будет представлена единственным словом СОРТБАЙТ. С этим словом связано скомпилированное во внутреннем представлении тело процедуры в целом, тела составляющих его процедур для отдельного использования теперь недоступны.

Включение новых процедур в базовый подсловарь ☼PRIME равносильно расширению языка ДССП. При этом язык может быть развит в том или ином направлении, адаптирован к определенной области применения. Таким образом, подобно естественным языкам, язык ДССП является развиваемым адаптивным языком (РАЯ). Развитие языка происходит путем пополнения его вновь определяемыми словами и словосочетаниями, представляющими объекты и понятия все более высокого уровня относительно примитивных типов данных и операций базового процессора. Адаптация к применениям осуществляется введением в язык слов и словосочетаний, представляющих объекты и процедуры, специфичные для данного применения. Могут быть созданы проблемно-ориентированные подсловари высокого уровня, рассчитанные на пользователей-непрограммистов. Существенно, что развитие и адаптация к применениям не связаны с необходимостью изменения или усложнения синтаксиса языка ДССП, важным достоинством которого является его простота.

Вследствие развиваемости языка архитектура ДССП-процессора не является замкнутой: ее также можно достраивать, модифицировать. Однако в любом случае неизменной должна сохраняться основа — исходный набор примитивов (форматов данных, элементарных операций) и правила его наращивания.

ГЛАВА 7
ПРИМЕР ПРИМЕНЕНИЯ МИКРОКОМПЬЮТЕРОВ

Едва ли найдутся другие технические средства, которые могли бы сравниться с микрокомпьютерами по многообразию, массовости и эффективности возможных применений. Универсальность, точность и автоматизм, свойственные компьютерам вообще, в микрокомпьютерах сочетаются с такими достоинствами, как невысокая стоимость, надежность, малогабаритность, незначительность энергопотребления. В совокупности эти качества поставили микрокомпьютер вне конкуренции среди известных средств управления, автоматизации и оптимизации процессов.

Но при всех своих исключительных возможностях и преимуществах микрокомпьютер — это все же только средство, применить которое можно как с пользой, так и без пользы, а то и во вред. Для успешного применения микрокомпьютеров требуется не только отчетливое понимание их возможностей и обстоятельное знание области применения, но также рациональная дисциплина, порядок, в котором должна быть построена работа.

По необходимости порядок должен быть тем же, что и в программировании — сверху вниз, от цели к средствам. Прежде всего надо четко определить цель применения, осознать, что должно быть достигнуто, а затем постепенно детализировать проблему, устанавливая подцели и прикидывая, как и какими средствами они могут быть достигнуты, отыскивая решения, позволяющие обойтись минимумом средств или обладающие какими-либо другими существенными преимуществами.

В случае сравнительно несложных применений микрокомпьютеров, таких как контроллеры с фиксированными, однозначно заданными функциями, работающие в автоматическом режиме при минимальном взаимодействии с человеком, указанные требования соблюсти нетрудно, и их соблюдают.

При компьютеризации сложных систем возникают трудности, вызванные тем, что функции компьютеров тесно переплетены с действиями людей, так что установить, насколько существенно и полезно участие компьютера в общем деле, непросто. При создании именно таких систем дисциплина особенно важна: прежде чем применить компьютер, надо понять, что и как он сможет делать в данном применении, чем будет полезен. При этом учитывать следует не

только положительные, но и отрицательные факторы, например, насколько усложнится работа людей в системе, не утратят ли они контроль над тем, за что отвечают, чем чреваты поломки и сбои и т. п. Другими словами, цель применения должна быть всесторонне обоснована и уяснена. Как это делается, мы покажем далее на конкретном примере применения микрокомпьютера в качестве средства, позволяющего существенно повысить эффективность и качество обучения различным предметам.

§ 1. Обучение с помощью компьютера

Первые попытки применить компьютер для обучения людей относятся к концу 50-х годов. Отправным пунктом явилось осуществление диалога человек — компьютер посредством подключенного к компьютеру телетайпа: телетайп печатает выдаваемые компьютером вопросы, предоставляя клавиатуру для ввода ответов. В зависимости от полученного ответа программа, которую выполняет компьютер, переключается на одну из предусмотренных в ней альтернатив, обеспечивая подходящее реагирование на данный ответ и дальнейшее продолжение диалога. Если учесть, что наряду с вопросами компьютер может выдавать на телетайп любую введенную в его память информацию, и предположить, что реагирование на ответы запрограммировано так, что ошибки учащегося будут своевременно разъясняться и корректироваться, а порядок прохождения и степень усвоения учебного материала — регламентироваться и контролироваться, то возможность обучения с помощью компьютера становится вполне очевидной.

Следует помнить, что, хотя непосредственным поводом для экспериментов по компьютерному обучению послужила техническая возможность диалога человек — компьютер, идея обучения при помощи машины существовала вне связи с компьютерами и задолго до их появления. Первоначальные опыты по созданию и применению обучающих машин приходятся на 20-е годы, а во второй половине 50-х годов с появлением программированного обучения конструирование реализующих его машин приняло массовый характер. В качестве целей этой деятельности, естественно, декларировались совершенствование системы образования, повышение эффективности обучения, модернизация школы на современной научно-технической основе и т. п. Цели эти не были достигнуты не только при помощи простых (бескомпьютерных) систем машинного обучения, но и с пришедшими на смену им компьютерными системами, создаваемыми на протяжении вот уже более четверти века.

Проведенные в середине 70-х годов в колледжах США испытания передовых в то время компьютерных систем PLATO IV и

TICCIT показали, что эти системы при всей их первоклассной технической оснащенности и богатых возможностях, предоставленных разработчикам обучающих программ и учебных материалов, практически не увеличивают дидактической эффективности процесса обучения, а вместе с тем неприемлемо дороги и сложны в применении. Предполагавшегося распространения этих систем не последовало.

С тех пор, благодаря развитию микрокомпьютеров, произошло значительное удешевление аппаратуры и немало сделано в отношении простоты и удобства компьютерных систем для пользователя. Сегодня персональные компьютеры пришли в школу — с ними работают или играют дети. Но надежды на компьютерную революцию в образовании, на решение при помощи компьютеров злободневной проблемы интенсификации обучения все еще остаются надеждами.

Потребность же в высокоэффективном машинном обучении исключительно велика и растет с каждым годом все быстрее. Ускоряющееся развитие науки, обновление техники и методов производства предъявляют все более и более строгие требования к системе образования, подготовки и переподготовки кадров, требования, удовлетворить которым, используя только традиционные средства обучения, невозможно. Необходима компьютерная система, пригодная для самого широкого применения по технико-экономическим параметрам и обеспечивающая возможность массового обучения в сжатые сроки и с гарантированным качеством. Эта система должна быть легко приспосабливаемой к динамично меняющимся условиям ее применения, оперативно реагирующей на зигзаги научно-технического прогресса быстрой переналадкой на широкомасштабное обучение специалистов по вновь возникающим актуальным направлениям, таким как информатика, микрокомпьютеры, роботы и т. п.

§ 2. Компьютер должен управлять

Поскольку мы намерены употребить микрокомпьютер для увеличения эффективности учебного процесса, то начинать надо с анализа этого процесса, с выявления его функциональной структуры и возможностей оптимизации микрокомпьютерными средствами процесса в целом или его компонент. Вернее же всего начать с уяснения того, как человек учится и каким образом микрокомпьютер может помочь учиться. По-настоящему научиться чему-нибудь человек может только делая то, чему учится, в ходе работы с предметом обучения, поэтому перепоручить эту работу компьютеру нельзя. Работа учителя в той ее главной части, которую справедливо расценивают как педагогическое искусство (раскрытие сущности предмета, пробуждение интереса к нему у учащихся, логичная

организация учебного материала, разработка упражнений), также не может быть возложена на компьютер, скажем, пока он не стал носителем высокоразвитого искусственного интеллекта.

Выходит, что компьютеру, а тем более маломощному микрокомпьютеру в процессе обучения по существу нечего делать. И понятно, таким образом, почему столь значительное место в компьютерных системах обучения занимает автоматизация предъявления учебного материала, прозванная в насмешку «электронным перелистыванием страниц». Другой вопрос, какова дидактическая ценность подобной автоматизации. Абсурдно полагать, что текст, прочитанный с дисплея, доходчивее того же текста, прочитанного по книге.

Можно, конечно, указать другие функции, посильные микрокомпьютеру в процессе обучения, такие как сверка ответов учащихся с эталонами, учет успеваемости, выполнение вычислений, редактирование и форматирование текстов, информационный поиск, графика и т. д. Однако все это хотя и находит применение в учебном процессе, но не является в нем главным и не может привести к значительному увеличению дидактического эффекта, не обеспечивает достижения преследуемой цели.

Решающая роль в обучении, как и во всяком деле, принадлежит функции управления. Учебный процесс не может быть эффективным, если он плохо организован, если действия учащихся не направляются компетентным руководителем, понимание и усвоение материала систематически не контролируется, ошибки не корректируются и т. д. Основной недостаток традиционной системы обучения заключается именно в слабости управления деятельностью учащихся в условиях большой группы (класса, аудитории). Эффективность и качество обучения значительно выше в малых группах, где преподаватель может уделить каждому ученику больше внимания.

Но дробление групп неприемлемо в широких масштабах по экономическим причинам. Проблема же обостряется по мере ужесточения требований к системе образования в ходе научно-технической революции. Все чаще предпринимаются попытки облегчить положение при помощи технических средств, в частности оборудованием так называемых аудиторий с обратной связью. В такой аудитории учащиеся могут отвечать на вопрос преподавателя нажатием имеющихся на партах кнопок, которым сопоставляются заранее предусмотренные варианты ответа, а преподаватель имеет возможность при помощи табло, отображающего состояние кнопок, оценить, скажем, степень понимания аудиторией материала, контролируемого заданным вопросом.

Нередко эта оценка производится при помощи компьютера, который обеспечивает также предъявление вопроса учащимся

посредством коллективного или индивидуальных, на каждой парте, дисплеев. Управление в подобных системах, несмотря на применение компьютера, оказывается слабым и неоперативным, поскольку обратная связь замкнута на человека-преподавателя, который к тому же реагирует на нее, воздействуя на аудиторию в целом, а не на каждого учащегося индивидуально. Компьютер употреблен не как средство управления, а лишь в качестве обработчика и передатчика данных и в какой-то мере как «советчик».

Не видно, однако, каких-либо серьезных возражений против того, чтобы реализация функции управления учебной деятельностью была полностью поручена компьютеру. Для достижения преследуемой цели эта функция является главной, удовлетворительно осуществить ее другими средствами не удается, а компьютер — наилучший исполнитель функций этого рода, поэтому и применять его в учебном процессе следует прежде всего для управления. Далее будет показано, что при надлежащем использовании даже сравнительно маломощный микрокомпьютер позволяет обеспечить действенное индивидуальное управление в большой группе учащихся и добиться высокой эффективности и качества обучения.

Разумеется, управление работой учащихся компьютер осуществляет путем выполнения разработанного человеком алгоритма проработки учебного материала. Предписываемые этим алгоритмом действия аналогичны действиям, совершаемым обычно преподавателями: определение последовательности прохождения материала, назначение упражнений, подтверждение правильности ответа учащегося или указание и разъяснение допущенных ошибок, задание дополнительных и вспомогательных упражнений, а также, в случае необходимости, упражнений на повторение ранее пройденного и т. п. Все эти действия должны быть направлены на то, чтобы обеспечить для каждого учащегося посильный для него и вместе с тем напряженный режим проработки материала при постоянном контроле правильности понимания и надежности усвоения, а также своевременной помощи в случае затруднений.

Конечно, осуществлять непрерывное индивидуальное управление одновременно многими учащимися компьютер сможет только в автоматическом режиме, т. е. при условии устранения из процесса человека-преподавателя, который должен вовлекаться в дело лишь в исключительных случаях или при необходимости произвести изменения в ходе занятия либо прервать его. Таким образом преподаватель получает возможность уделять внимание отдельным учащимся, в то время как работа всех остальных управляется компьютером в соответствии с предписанным алгоритмом, прорабатываемым предметом и индивидуальными достижениями каждого учащегося на текущем этапе проработки.

Учащиеся управляются компьютером так, что каждый добивается понимания и усвоения всего изучаемого материала, но работает при этом в свободном, посильном ему темпе, выполняя необходимое для овладения тем или иным вопросом количество упражнений и возвращаясь к этому вопросу столько раз, сколько потребуется для его уяснения. Чтобы осуществить это, надо наряду с индивидуальным управлением предоставить учащимся независимый доступ к учебному материалу, для чего обычно производят выдачу фрагментов («кадров») последнего на дисплеи, которыми в таком случае должны оборудоваться рабочие места учащихся. Речь идет об уже упоминавшемся «электронном перелистывании страниц», введение которого в десятки раз удорожает и усложняет необходимое компьютерное оборудование и вместе с тем существенно ухудшает гигиенические условия для учащихся. Но, имея в виду, что дидактический эффект порождается не в результате электронного предъявления материала, а главным образом в зависимости от добротности материала по существу и действенности управления его проработкой, можно с не меньшим успехом предоставлять учебный материал в печатном исполнении, в виде особым образом форматированных книг.

§ 3. Микрокомпьютерная система обучения «Наставник»

Изложенные в предыдущем параграфе рассуждения о роли компьютера в процессе обучения убедительно подтверждены опытом разработки и практического использования в Московском государственном университете им. М. В. Ломоносова автоматизированной системы обучения «Наставник». Эта система базируется на принципе, прямо противоположном обычному «чем больше технических возможностей, тем лучше». Чтобы сделать ее практичной, т. е. недорогой, легко обслуживаемой, надежной, несложной для освоения и использования как учащимися, так и преподавателями, решено было не злоупотреблять компьютером, применяя его для реализации только тех функций, которые, во-первых, безусловно существенны в отношении дидактики, а во-вторых, экономно и надежно осуществимы при нынешнем состоянии компьютерной техники. Кроме того, важное значение придавалось тому, чтобы система была предельно простой для пользователей, чтобы от обучаемых в ней не требовалось каких-либо компьютерных знаний или навыков, чтобы работать с ней мог преподаватель без специальной подготовки, чтобы разработку учебных материалов могли производить не программисты, а обыкновенные преподаватели и с меньшими трудозатратами, чем на известных системах компьютерного обучения.

В соответствии с таким подходом компьютер в «Наставнике» выполняет две функции: 1) управление учебной деятельностью учащихся, 2) протоколирование процесса обучения. Последнее необходимо не только для получения подробной информации об успеваемости и затруднениях каждого учащегося, но особенно для объективной оценки и направленного совершенствования учебного материала, а также для проведения дидактических экспериментов с целью исследования механизмов обучения и уточнения имеющихся методов и приемов. Таким образом, в «Наставнике» компьютер реализует две обратных связи: одну в контуре учащийся — учебный материал, корректирующую деятельность учащегося по овладению учебным материалом, другую — в контуре преподаватель/разработчик учебного материала — процесс обучения, способствующую совершенствованию учебных материалов и методик.

Учебный материал учащиеся в системе «Наставник» получают в виде специальным образом структурированных книг. Компьютер управляет обучением, выдавая номера подлежащих проработке секций материала, упражнений, а также справок, комментирующих ответы, которые поступают от учащегося в процессе выполнения упражнений. Построение системы по принципу «книга плюс компьютер» в противоположность «электронному перелистыванию страниц» позволило не только радикально упростить терминалы учащихся и сократить потребность в компьютерных ресурсах для их обслуживания (на 32 терминала достаточно микрокомпьютера с памятью 48 КВ), но явилось предпочтительным и в других отношениях.

Так, чтобы начать работать в «Наставнике», не нужно никакой предварительной подготовки — достаточно уметь читать, а остальному система научит при помощи книжки «Вступительный курс», проходимой за 15—30 мин. Это означает по меньшей мере три преимущества: во-первых, доступность системы — в ней могут работать даже школьники младших классов, во-вторых, пригодность ее для начального обучения, например, для подготовки к работе с компьютером, в-третьих, обучение в системе с самого начала может производиться практически без участия преподавателя.

Разработка учебного материала в виде книги, в противоположность программированию его для предъявления компьютером, значительно менее трудоемка (по некоторым оценкам в 5 раз) и может выполняться не программистами, а преподавателями соответствующих специальностей, не умеющими программировать. При этом внимание и усилия разработчика концентрируются на содержании материала и дидактике, а не на технических проблемах компьютеризации его. В «Наставнике» учебный материал полностью отделен от компьютерной программы, организующей его проработку, и разработчик учебного материала предоставляет для компьютера

лишь данные о структуре материала (номера секций и упражнений) и о соответствии справок возможным ошибкам. Выполняемая компьютером в процессе проработки учебного материала программа (так называемая подсистема обучения «Наставник») едина для всех проходимых курсов-предметов.

Аппаратура системы «Наставник» включает микрокомпьютер, доукомплектованный одноплатным контроллером терминалов учащихся, терминалы в количестве до 32 штук и 7-проводный кабель, связывающий терминалы с контроллером. Терминал учащегося подобен настольному микрокалькулятору с 11-клавишной клавиатурой и цифровым индикатором на 2 или на 8 десятичных цифр с запятыми. Терминал обеспечивает возможность посылки в компьютер десятичных цифр и отображения на цифровом индикаторе поступающих из компьютера сообщений. Посылка цифры осуществляется однократным нажатием клавиши, на которой эта цифра обозначена, и сопровождается немедленным появлением на индикаторе ответного сообщения. Контроллер терминалов постоянно следит за состоянием клавиатур, передавая компьютеру посланные цифры вместе с номерами терминалов, с которых эти цифры поступили, и отображая на индикаторы получаемые от компьютера сообщения.

Осуществление диалога учащийся—компьютер в цифровой форме в данном случае выгодно как в отношении компьютера, так и учащегося, поскольку всякий запрос или ответ последнего сводится к единственному нажатию клавиши. Кроме того, функционирование системы не зависит от языка, на котором ведется обучение: при наличии учебных материалов на разных языках прорабатывать их в «Наставнике» можно даже одновременно, на одном и том же занятии.

Функции, присваиваемые цифровым клавишам, могут варьироваться в зависимости от применения системы, а также назначения терминала. Например, при использовании терминала в качестве преподавательского его клавишам приписывается смысл, отличный от установленного для терминала учащегося. Но во всех случаях смысл клавиш естественный и легко усваиваемый. Так, в подсистемах обучения и тестирования клавиша 0 (нуль) уведомляет компьютер о том, что выданное им сообщение принято и все предписанное этим сообщением выполнено, что равносильно вызову следующего сообщения. «Ненулевые» клавиши при этом служат для указания номера ответа, полученного в результате выполнения упражнения. Компьютер повторяет данный номер, выдавая на индикатор его «эхо» и предоставляя возможность корректировать его указанием другого номера, пока не будет нажата клавиша 0, подтверждающая готовность ответа и вызывающая сообщение, которым компьютер реагирует на принятый ответ.

Диалог учащегося с компьютером при таких соглашениях получается предельно лаконичным и простым, т. е. на поддержание его учащийся затрачивает минимум времени и усилий. В самом начале занятия компьютер выдает на индикаторы всех терминалов сообщение о готовности к диалогу, после чего нажатие на данном терминале клавиши 0 вызовет на индикатор номер подлежащей проработке секции учебного материала, а еще нажатие — номер одного из упражнений в этой секции, которое следует выполнить.

Упражнения представлены в форме вопросов или задач с так называемым множественным выбором ответа — каждое содержит пронумерованный список ответов, из которых обычно только один верный, а прочие получаются в результате типичных ошибок. Учащийся указывает номер найденного им ответа нажатием соответствующей клавиши, проверяет «эхо» и клавишей 0 вызывает очередное сообщение. Если указанный ответ верен, компьютер выдает на индикатор подтверждение правильности ответа, а если допущена ошибка, то на индикаторе появляется номер справки, комментирующей эту ошибку.

Очередное нажатие клавиши 0 вызовет на индикатор либо номер упражнения в прорабатываемой секции, либо номер другой секции, в которой затем будет назначено упражнение. Таким образом, взаимодействие учащегося с компьютером внешне выражается в многократном повторении одной и той же последовательности действий:

— вызов номера упражнения, возможно, предворяемого номером секции;

— посылка номера ответа;

— вызов реакции компьютера на посланный ответ, т. е. номера справки или подтверждения правильности.

Учебный материал, ради простоты поиска в нем указываемых компьютером элементов, организован в виде последовательности пронумерованных секций. Как правило, секция посвящена отработке определенного вопроса и начинается сжатым изложением его (инструктивный текст на 2—3 страницах), после чего следует 10—15 упражнений, а в конце — список справок. В справках возможны ссылки на отдельные абзацы инструктивного текста, поэтому абзацы пронумерованы: а1, а2, а3 и т. д. Если необходима ссылка на абзац другой секции, то спереди добавляется ее номер, например: 13а9—9-й абзац секции 13. Секция может не содержать инструктивного текста, а включать только упражнения и справки.

Упражнения в пределах каждой секции нумеруются, начиная с 1. При этом различают основные, итоговые и вспомогательные упражнения. Основные размещаются в порядке номеров первыми, затем идут итоговые, а после них вспомогательные, которых может и

не быть. Общее число упражнений в секции не может превышать 15. Основные упражнения (числом 5—10) касаются отдельных элементов отрабатываемого вопроса, итоговые (числом 3—6) посвящены отработке вопроса в комплексе, а вспомогательные употребляются в тех случаях, когда по допускаемым учащимся ошибкам видно, что заданное упражнение ему прямо не дается и требуется более постепенный подход. В качестве вспомогательного может быть использовано любое из основных упражнений прорабатываемой секции.

В описании секции, предоставляемом компьютеру в составе управляющей информации по прорабатываемому курсу (т. е. в составе данных для программы, управляющей обучением), помимо номера секции, сообщаются количества основных и итоговых упражнений, а затем для всех упражнений в порядке их номеров описываются реакции на предусмотренные в них ответы. Установлено 4 типа реакций: 1) ответ правильный, 2) выдать номер справки, 3) то же и назначить вспомогательное упражнение, 4) то же и назначить упражнение на повторение в ранее пройденной секции.

Программное оснащение системы «Наставник» реализовано в виде совокупности независимо функционирующих компонент (подсистем), в число которых входят, в частности:

— подсистема обучения «Наставник»,
— подсистема контроля знаний «Экзаменатор»,
— подсистема тестирования умений «Тест»,
— подсистема подготовки управляющей информации,
— подсистема обработки протоколов занятий.

Все эти компоненты представляют собой программы, разработанные и выполняемые в диалоговой системе структурированного программирования ДССП. В готовом для использования виде каждая подсистема доступна как скомпилированный для загрузки при помощи начального загрузчика с дискеты, магнитной кассеты или с перфоленты программный модуль, включающий необходимые средства ДССП, программы собственно подсистемы, а также средства самоконтроля, диагностики и обеспечения диалога с оператором, например, при вводе управляющей информации и данных об учебной группе или при распечатке протокола занятия.

Компактность кода ДССП позволила осуществить каждую из подсистем полностью резидентной в главной памяти микрокомпьютера вместе с управляющей информацией и протоколом занятия (для подсистем обучения, контроля и тестирования). Внешняя память используется только для загрузки подсистем, ввода управляющей информации и сохранения протоколов, выводимых по окончании занятий. В случаях, когда протоколы не сохраняются для дальнейшей обработки, а только распечатываются в конце занятия, специализированная по отдельным предметам подсистема обучения может быть размещена вместе с управляющей информацией в постоянной памяти компьютера и функционировать вообще без внешней памяти, что существенно упрощает ее обслуживание.

Наряду с многоместным (многотерминальным) вариантом, «Наставник» может использоваться также в одноместном варианте, причем на простейших портативных микрокомпьютерах с минимальной клавиатурой и цифровым индикатором в качестве устройства вывода. Более того, не исключена возможность реализовать, например, подсистему обучения на программируемом карманном калькуляторе, дополненном постоянной памятью, в которой модуль для хранения управляющей информации будет сменным.

«Наставник» является системой обучения общего назначения — учиться в нем, при наличии соответствующих учебных материалов, можно всему, чему обычно учатся по книгам, при помощи лекций и семинаров. Опыт практических применений «Настаника» свидетельствует о том, что он обеспечивает существенно большую дидактическую эффективность и качество учебного процесса по сравнению с достигаемой традиционными средствами. Однако «Наставник» не следует противопоставлять иным применениям компьютера в учебном процессе, таким, как использование его для выполнения расчетов, моделирования, обработки текстов, учебного проектирования, а также в качестве основы разного рода тренажеров. Обучение в «Наставнике» должно сопровождаться практической работой с подобными компьютерными (и не только компьютерными) системами, так же как любой теоретический курс должен закрепляться практикой. В частности, при обучении программированию на 32 места в «Наставнике» достаточно иметь 3—4 микрокомпьютера, поочередно используемых учащимися под наблюдением преподавателя для практики по изучаемым компьютерным дисциплинам.

§ 4. Процедура УРОК в подсистеме обучения «Наставник»

Ведущее положение в системе «Наставник» занимает, разумеется, подсистема обучения, поскольку именно в этой подсистеме реализуется то оптимальное управление учебной деятельностью, которое является главной целью создания системы. Подсистемы «Экзаменатор» и «Тест» представляют собой примеры других полезных применений аппаратуры «Наставника» в учебном процессе. К числу таких применений относятся также автоматизированные практикумы, групповые психофизиологические эксперименты, анкетные опросы и т. п. Подсистемы подготовки управляющей информации и обработки протокольных данных являются служебными. Их назначение — обеспечивать использование подсистем, непосредственно работающих с учащимися.

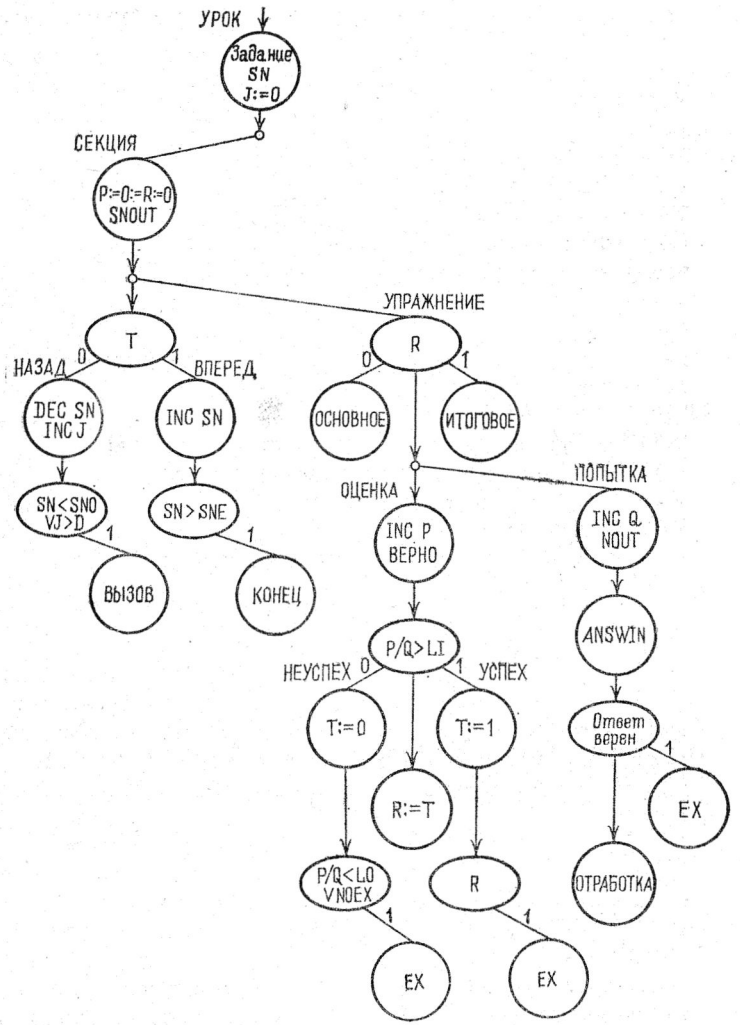


Рис. 4

Как можно было понять из общей характеристики «Наставника», подсистема обучения представляет собой программу, которая при наличии управляющей информации, подготовленной автором учебного материала, осуществляет индивидуальное руководство работой учащихся по освоению данного материала и подробно протоколирует ход этой работы. Центральным звеном в подсистеме обучения

является процедура проведения с учащимся одного занятия (процедура УРОК), представленная ниже в упрощенном варианте на языке РАЯ в ДССП и в виде блок-диаграммы на рис. 4.

Упрощенная процедура УРОК в подсистеме обучения «Наставник»:

```
[УРОК — процедура проведения занятия]
    VAR SN [номер прорабатываемой (текущей) секции]
    VAR J [счетчик возвратов в предыдущую секцию]
    [параметры курса: SN0 — номер начальной секции, SNE —
    номер заключительной секции, L0 — уровень успеваемости,
    снижение до которого приводит к возврату в предыдущую сек-
    цию, LI — уровень успеваемости, достижение которого приво-
    дит к продвижению вперед, D — максимально допустимое чис-
    ло возвратов в течение одного занятия]
: УРОК [ ] !0 J SNIN RP СЕКЦИЯ [ ] ;
    [SNIN — задание номера секции в начале занятия]
    [СЕКЦИЯ — процедура прохождения секции]
    VAR P [счетчик верных ответов]
    VAR Q [счетчик попыток ответа]
    VAR R [ранг упражнения: R = 0 — основное, R = 1 — ито-
    говое]
: СЕКЦИЯ [ ] !0 P !0 Q !0 R SNOUT RP УПРАЖНЕНИЕ
    [0/1] BR0 НАЗАД ВПЕРЕД [ ] ;
    [SNOUT — выдача на терминал номера секции SN, УПРАЖ-
    НЕНИЕ — выполнение упражнения, НАЗАД — возврат в пре-
    дыдущую секцию, ВПЕРЕД — переход в последующую секцию]
: ВПЕРЕД [ ] INCSN SNE SN < [SNE < SN] IF+ КОНЕЦ [ ];
    [INCSN — приращение номера секции SN, КОНЕЦ — конец
    курса]
: НАЗАД [ ] DECSN SN0 SN > !1+ J J D > &0 [SN0 > SN \/
    \/ J > D] IF+ ВЫЗОВ [ ]; [DECSN — убавление номера
    секции SN, ВЫЗОВ — вызов к преподавателю]
: УПРАЖНЕНИЕ [ ] R BR0 ОСНОВНОЕ ИТОГОВОЕ [N] RP
    ПОПЫТКА ОЦЕНКА [0/1] ; [ОСНОВНОЕ — выбор номера N
    основного упражнения, ИТОГОВОЕ — выбор номера N ито-
    гового упражнения, ПОПЫТКА — прием и обработка ответа,
    ОЦЕНКА — определение дальнейшего пути в зависимости от
    текущей успеваемости]
: ПОПЫТКА [N] !1+ Q NOUT ANSWIN [ответ верен — 1, невер-
    ен — 0] EX+ [выход из RP ПОПЫТКА, если ответ верен]
    ОБРАБОТКА [N] ; [NOUT — выдача N на терминал, ANSWIN —
    прием ответа, ОБРАБОТКА — выдача на терминал номера
    справки и вспомогательного упражнения, если оно предусмот-
    рено]
```

```
: ОЦЕНКА [N] D !1+ P ВЕРНО 100 P * L1 Q * > [P/Q > L1]
    BR+ УСПЕХ [1] НЕУСПЕХ [0] ! R [ ] ;
    [ВЕРНО — выдача на терминал подтверждения правильности
    ответа, УСПЕХ — продвижение вперед, НЕУСПЕХ — на
    прежнем уровне или возврат назад]
: УСПЕХ [ ] 1 R EX+ [если R = 1, то выход из RP УПРАЖНЕ-
    НИЕ, T = 1] [1] ;
: НЕУСПЕХ [ ] 0 100 P * L0 Q * < NOEX &0 EX+
    [если P/Q<L0 \/ NOEX, то выход из RP УПРАЖНЕНИЕ,
    T = 0]
    [0] ; [NOEX — запас упражнений исчерпан]
```

Занятием в подсистеме обучения «Наставник» называется промежуток времени (обычно 1,5—2 ч), в течение которого система предоставлена учебной группе для проработки материала по какому-либо предмету. Занятие может проводиться одновременно по двум-трем предметам, прорабатываемым соответствующим числом групп, общая численность которых не превышает количества терминалов учащихся. Для прохождения всего предмета может потребоваться то или иное число занятий в зависимости от объема учебного материала. Кроме того, время, затрачиваемое на проработку одного и того же материала, варьируется вследствие индивидуальных различий учащихся нередко в 1,5 раза даже в пределах одной учебной группы. Поэтому количества материала, пройденного отдельными учащимися на протяжении занятия, различны, и следующее занятие каждый начинает со своей, достигнутой на предыдущем занятии секции. Технически это обеспечивается тем, что по окончании занятия данные о текущем состоянии группы сохраняются на внешнем носителе, а в начале следующего занятия этой группы вводятся в компьютер, задавая исходный номер секции для каждого учащегося (в процедуре УРОК задание исходной секции осуществляет процедура SNIN).

Простоты ради процедура УРОК описана в одноместном варианте, т. е. применительно к управлению работой одного учащегося. В действительности она переключается на обслуживание того или иного учащегося при каждом нажатии им клавиши на терминале. Переключение заключается в том, что получаемый компьютером при нажатии клавиши код цифры и номера терминала, на котором произведено нажатие, запускает очередной для данного терминала этап процедуры УРОК с принадлежащими этому терминалу текущими значениями параметров. Для каждого терминала выделена область памяти, в которой хранятся текущие значения параметров работающего на нем учащегося, и при всяком обращении с данного терминала процедура УРОК применяется к его области с учетом ее текущего состояния и поступившей цифры. Первое с начала занятия нажатие клавиши 0 вызовет применительно к терминалу, на котором оно произведено, выполнение процедуры SNOUT (выдача на индикатор терминала номера секции) с переходом текущего состояния на этап выдачи номера упражнения, реализуемой при втором нажатии клавиши 0, и т. д.

Наряду с параметрами, имеющими отдельные значения для каждого терминала, такими, как номер текущей секции SN, счетчик верных ответов P, счетчик попыток ответа Q, счетчик возвратов в предыдущую секцию J, процедура УРОК работает также с параметрами, значения которых характеризуют учебный материал (курс) и заданы его автором в составе управляющей информации: номера начальной и заключительной секций SN0 и SNE, граничные уровни успеваемости L0 и L1, допустимое в течение занятия число возвратов в предыдущую секцию D. Кроме того, каждая секция характеризуется количествами имеющихся в ней основных и итоговых упражнений, а каждое упражнение — описаниями всех предусмотренных в нем ответов, содержащими номера справок и, если необходимо, вспомогательных упражнений.

Таким образом, инициируемая нажатиями клавиш на терминалах процедура УРОК работает всякий раз с принадлежащей обслуживаемому терминалу областью данных о текущем состоянии учащегося, привлекая необходимую управляющую информацию по прорабатываемому курсу. При этом текущее состояние модифицируется, а выполненная учащимся работа протоколируется путем фиксации номеров заданных ему упражнений и поступивших от него ответов.

Назначение процедуры УРОК — управлять проработкой учебного материала учащимся, добиваясь правильного понимания и усвоения изучаемого предмета. Процедура организована в форме цикла, осуществляющего прохождение секций в порядке возрастания номеров, если уровень успеваемости учащегося, оцениваемый как отношение P/Q числа правильных ответов к числу попыток, превышает верхнее граничное значение L1, и в порядке убывания, если уровень успеваемости оказывается ниже нижней границы L0. Возможность перемещения по курсу в порядке убывания номеров секций, т. е. путем возврата в предыдущую секцию, ограничена: в течение одного занятия допустимо не более D возвратов, где D — параметр прорабатываемого курса. Учащийся, превысивший допустимое число возвратов, снимается с обслуживания и направляется к преподавателю для выяснения причин неуспеваемости.

Проработка секции также организована в форме цикла — повторного выполнения процедуры УПРАЖНЕНИЕ, которая включает выдачу на индикатор терминала номера упражнения, предоставление учащемуся попыток ответа, пока не будет получен верный

ответ, отработку ошибок путем выдачи номеров справок и, может быть, вспомогательных упражнений, а при получении верного ответа — оценку текущей успеваемости и принятие решения о продвижении вперед, или выполнении еще одного упражнения того же уровня, или возврате назад. Приступив к новой секции, учащийся первым делом изучает инструктивный текст (если он имеется в начале секции), затем выполняет одно или несколько (в зависимости от количества совершаемых ошибок) из основных упражнений, после чего переходит к итоговым, однако в случае недостаточной успеваемости снова возвращается к основным, а то и в предыдущую секцию.

Номер назначаемого упражнения система определяет путем случайного выбора из имеющейся совокупности упражнений требуемого ранга: R = 0 — основное, R = 1 — итоговое. При повторном назначении упражнений того же ранга уже выполненные упражнения исключаются. Продвижение вперед, т. е. от основных упражнений к итоговым и от итоговых в следующую секцию, производится при условии $P/Q > L1$. Возврат назад, в предыдущую секцию, происходит при $P/Q < L0$, а также в случае исчерпания запаса упражнений. В случае $L0 \leqslant P/Q \leqslant L1$ дополнительно назначаются упражнения из текущей секции, если запас их не исчерпан.

Несмотря на несложность реализуемого алгоритма, процедура УРОК, как показала практика, характеризуется вполне удовлетворительной дидактической эффективностью и гарантированным качеством обучения всех учащихся. При этом она обладает важными преимуществами перед системами, основанными на сложных и детально разработанных стратегиях (впрочем, как правило, не более эффективных): 1) небольшая трудоемкость разработки учебных материалов, 2) минимальные требования к ресурсам компьютера — объем памяти, занимаемой подсистемой обучения «Наставник» вместе с поддерживающими ее средствами ДССП, не превышает 16 К байтов.

ПРИЛОЖЕНИЕ
АРХИТЕКТУРА ДССП-ПРОЦЕССОРА

Базовая конфигурация аппаратуры ДССП включает стековый диалоговый процессор, память с произвольным доступом к 16-битным словам и байтам (главная память), дисплейный с клавиатурой терминал, устройства ввода/вывода и внешней памяти — принтер, перфоратор и считыватель перфоленты, память на дисках или на кассетной магнитной ленте. Эту конфигурацию можно расширять, пополняя систему драйверами добавляемых устройств.

Выполняемая процессором последовательность операций (процесс) определяется *входным потоком* команд и данных, которые могут поступать либо с клавиатуры терминала, либо с внешних носителей, либо из буфера редактора текстов — особо организованной области главной памяти, используемой, в частности, для конструирования и модификации исходной программы. Синтаксически (формально) входной поток представляет собой последовательность слов — разделенных пробелами цепочек литер, т. е. букв, цифр и других знаков алфавита ввода/вывода.

Процессор интерпретирует поступающие на его вход слова и словосочетания как команды выполнить те или иные действия либо как данные, над которыми эти действия выполняются. Например, слово + означает операцию сложения, слово B10 предписывает установить режим десятичного ввода/вывода, слово —234 представляет отрицательное целое число, а словосочетание VAR X является командой образовать целочисленную переменную с именем X.

Имеется базовый набор слов, опознаваемых и выполняемых процессором как его собственные команды (примитивы). Слова же, состоящие только из цифр и, может быть, знака минус в качестве первой литеры, воспринимаются как числа (числовые литеры) и автоматически засылаются в стек, над которым производятся операции. Слово, не относящееся к примитивам и числам, может быть выполнено процессором или употреблено в качестве данного только при условии, что оно определено (описано) в уже предоставленной процессору части входного потока. Поступление неизвестного процессору слова W вызывает выдачу на экран терминала сообщения: «Не знаю W».

Определение слова, именующего действие (процедуру), включается во входной поток в виде команды «Скомпилировать опреде-

ление процедуры», представляемой словосочетанием, которое начинается словом : («скомпилировать») и оканчивается словом ; («конец определения»). Структуру этой команды можно описать в виде

: ⟨имя процедуры⟩ ⟨тело процедуры⟩ ;

где ⟨имя процедуры⟩ — определяемое слово, ⟨тело процедуры⟩ — определяющая последовательность слов, в которой могут быть слова, определенные в последующей части входного потока, однако до того, как определяемое слово будет употреблено в качестве команды.

Определение слов, именующих данные, производится командами образования соответствующих структур данных: VAR — создать переменную, VCTR — создать вектор и т. п. В ходе выполнения такой команды процессор резервирует в памяти место для значения поименованного данного и связывает адрес (внутренний указатель) этого места с именем, представляющим данное во входном потоке (с внешним именем). Например, при выполнении команды VAR X — «Создать переменную X» — резервируется 16-битная ячейка памяти и ее адрес связывается с именем X таким образом, что упоминание X в качестве ссылки вызывает копирование в стек текущего значения этой ячейки, команда ' X засылает в стек ее адрес, а команда !X пересылает в нее текущее значение вершины стека.

Имена определяемых процедур и структур данных заносятся в *словарь процессора* вместе с сопоставленными им внутренними указателями и в дальнейшем могут употребляться наряду с базовыми командами, пока не будут удалены в результате «забывания» или чистки соответствующей части словаря. Отдельные имена могут быть включены в подсловарь примитивов ⌼ PRIME и зафиксированы в нем в качестве неудаляемых. Таким образом осуществляется расширение языка ДССП и приспособление его к тому или иному применению. Это отражено в названии РАЯ — развиваемый адаптивный язык.

Типичная РАЯ-программа состоит из команд — описателей данных — и постепенно детализируемого определения процедуры, реализующей заданный алгоритм. При этом управление ходом программы, т. е. последовательностью выполнения ее частей, базируется не на переходах, а на *вложенности* (безусловной, условной, многократной, многоуровневой) подчиненных процедур. Условная вложенность программируется при помощи команд условного выполнения IF и выбора BR, многократная вложенность — при помощи команд повторения RP и DO, а также команд группы EX — выхода из цикла.

Во входном потоке одни и те же слова языка допустимо использовать как для прямого указания процедур или данных, так и в составе компилируемых определений. Однако имеются ограничения, а именно:

— недопустимы определения в определениях,

— невыполнимы прямые (нескомпилированные) команды группы BR.

Особыми литерами в алфавите ДССП являются квадратные скобки. Текст, заключенный в квадратные скобки, полностью игнорируется процессором и предназначен для пояснений (комментариев), адресованных человеку.

Команды преобразования и тестирования данных выполняются ДССП-процессором над стеком операндов, который служит также средством передачи параметров процедур. Элементами стека являются 16-битные слова, значения которых в числовой интерпретации трактуются как целые со знаком в диапазоне от —32768 до 32767 (от -2^{15} до $2^{15}-1$). При этом значения-результаты операций, выходящие за пределы данного диапазона, автоматически заменяются сравнимыми по модулю 2^{16} из заключенных в нем значений. Например, —32769 превращается в 32767, а 32769 — в —32767.

Стек организован как пословно адресуемая с помощью указателя q память s. Текущему значению q соответствует вершина стека s (q). При q = q0, где q0 — начальное значение указателя, стек пуст. Добавление элемента в стек связано с приращением значения q, а удаление элемента из стека — с убавлением значения q. Процедура приращения incq (increment q) выполняется как q:=q + 1, процедура убавления decq (decrement q) — соответственно как q:=q—1. С использованием данных процедур засылка в стек значения переменной X (push X) и обратная пересылка в X с удалением верхнего элемента стека (pop X) программируются в виде

```
push X: incq; s(q):=X;
pop X: X:=s(Q); decq;
```

Главная память ДССП-процессора, адресуемая в отличие от стека как пословно, так и побайтно, обозначается буквой m. Например, m(a,15:0) — 16-битная ячейка памяти с адресом a, m(a,7:0) — ячейка-байт с адресом a. В ней размещаются тела определяемых процедур и структур данных, словарь процессора, буфер редактора текстов и программы, эмулирующие ДССП-процессор. Физически вся эта память вместе со стеком операндов и невидимым пользователю стеком адресов возврата из подпрог-

рамм осуществлена в оперативном запоминающем устройстве компьютера, эмулирующего систему.

Приводимый ниже базовый набор команд ДССП включает главным образом примитивы процессора, т. е. соответствует набору команд компьютера в традиционном понимании. Поэтому в него не включены команды редактора текстов, файловой системы, отладчика, встроенного ассемблера, расширенной арифметики, графики и других расширений системы.

Комментарии в квадратных скобках представляют состояния стека операндов до и после выполнения описываемой команды.

Пояснение к средствам командного прерывания. Аппарат командного прерывания, представленный в базовом наборе командами TRAP, ON и EON, используется при обнаружении в процессе выполнения программы ошибок (например: неизвестная процедура, несоответствие типов данных, несовпадение контрольных сумм), а также при возникновении особых условий, таких как приостановка или окончание текущего процесса, исчерпание стека или буфера и т. п. Имя «сигнал» прерывания объявляется командой TRAP, сопоставляющей ему имя «конечной реакции»:

TRAP ⟨имя прерывания⟩ ⟨имя «конечной реакции»⟩

«Конечная реакция» — это процедура, обслуживающая объявленное командное прерывание, если его обслуживание не перехвачено командой ON или EON, подменяющей процедуру «конечной реакции» иной процедурой.

Встретившись в выполняемой программе, сигнал командного прерывания инициирует ближайшую из сопоставленных ему командами TRAP, ON и EON реакций. Если перехватов ON или EON нет, то инициируется «конечная реакция». Например, прерыванию EXERR, которое сигнализирует о неизвестной процедуре во входном потоке или о нечисловой литере при вводе числа, сопоставлена «конечная реакция» NOP, однако в процедуре выполнения очередного слова программы эта «пустая» реакция подменяется выдачей сообщения «Не знаю ⟨имя⟩» с последующим перезапуском системы (RESTART), а в процедуре ввода числа — соответствующей реакцией на ошибку при вводе.

Базовый набор команд ДССП

Команды манипулирования стеком

Мнемокод	Команда	Описание	Примеры
n1	Заслать в стек число-литерал n1	incq; s(q): = n1;	[ ] 256 [256]
# ch	Заслать в стек код литеры ch	incq; s(q): = # ch;	[ ] # 1 [49]
D	Удалить слово из стека	decq;	[53] D [ ]
DD	Удалить два слова из стека	decq; decq;	[7, 5] DD [ ]
DS	Очистить стек	q: = q0;	1..., 3, 9] DS [ ]
C	Скопировать вершину стека	incq; s(q): = s(q — 1);	[5] C [5, 5]
C2	Скопировать с глубины 2	incq; s(q): = s(q — 2);	[3,5] C2 [3, 5, 3]
C3	Скопировать с глубины 3	incq; s(q): = s(q — 3);	[2, 3, 5] C3 [2, 3, 5, 2]
C4	Скопировать с глубины 4	incq; s(q): = s(q — 4);	[0, 2, 3, 5] C4 [0, 2, 3, 5, 0]
CT	Скопировать с глубины, указанной вершиной T	t: = s(q); s(q): = s(q — t);	[7, 4, 5, 3] CT [7, 4, 5, 7]
E2	Обменять с глубины 2	s(q): = :s(q — 1);	[8, 5] E2 [5, 8]
E3	Обменять с глубины 3	s(q): = :s(q — 2);	[6, 8, 5] E3 [5, 8, 6]
E4	Обменять с глубины 4	s(q): = :s(q — 3);	[4, 6, 8, 5] E4 [5, 6, 8, 4]
ET	Обменять с глубины, указанной вершиной T	t: = s(q); decq; s(q): = :s(q — t + 1);	[6, 8, 5, 3] ET [5, 8, 6]
T0	Присвоить вершине значение 0	s(q): = 0;	[5] T0 [0]
T1	Присвоить вершине значение 1	s(q): = 1;	[5] T1 [1]
1+	Прибавить 1	s(q): = s(q) + 1;	[5] 1+ [6]
1—	Вычесть 1	s(q): = s(q) — 1;	[5] 1— [4]
2+	Прибавить 2	s(q): = s(q) + 2;	[5] 2+ [7]

Продолжение

Мнемокод	Команда	Описание	Примеры
2—	Вычесть 2	s(q):=s(q)—2;	[5] 2— [3]
+	Сложить	t:=s(q); decq; s(q):=s(q)+t;	[2, 3] + [5]
—	Вычесть	t:=s(q); decq; s(q):=s(q)—t;	[2,3] — [—1]
*	Перемножить	t:=s(q); decq; s(q):=s(q)*t;	[2, 3] * [6]
/	Поделить нацело	t:=s(q); decq; u:=s(q); s(q):=$(u/t)_{цел}$; incq; s(q):=u—s(q—1)*t;	[—7, 3] / [—2, —1] [7, —3] / [—2, 1]
MIN	Взять меньшее	t:=s(q); decq; s(q):=min(t, s(q));	[5, —8] MIN [—8]
MAX	Взять большее	t:=s(q); decq; s(q):=max(t, s(q));	[5,—8] MAX [5]
NEG	Изменить знак	s(q):=—s(q);	[—5] NEG [5]
ABS	Взять по абсолютной величине	**if** s(q)<0 **then** NEG;	[—5] ABS [5]
SGN	Взять знак	**if** s(q)<0 **then** s(q):=—1 **else if** s(q)>0 **then** s(q):=1;	[—5] SGN [—1] [5] SGN [1]
<	Меньше	t:=s(q); decq; **if** s(q)<t **then** s(q):=1 **else** s(q):=0;	[3, 5] < [1] [—3, —5] < [0]
=	Равно	t:=s(q); decq; **if** s(q)=t **then** s(q):=1 **else** s(q):=0;	[3, 5] = [0] [3, 3] = [1]

Продолжение

Мнемокод	Команда	Описание	Примеры
>	Больше	t:=s(q); decq; **if** s(q)>t **then** s(q):=1 **else** s(q):=0;	[3, 5] > [0] [3, —5] > [1]
NOT	НЕ (отрицание)	**if** s(q)=0 **then** s(q):=1 **else** s(q):=0;	[5] NOT [0] [0] NOT [1]
INV	Побитное отрицание (инверсия)	t:=s(q); **for** j:=0 **to** 15 **do** t(j):=¬t(j); s(q):=t;	[—1] INV [0] [1] INV [—2]

&	Побитное И (конъюнкция)	t:=s(q); decq; u:=s(q);
		for j:=0 **to** 15 **do**
		u(j):=u(j)∧t(j); s(q):=u;
			[1, 0] & [0]
			[—1, 1] & [1]
			[—2, 1] & [0]

&0	Побитное ИЛИ («конъюнкция нулей», дизъюнкция)	t:=s(q); decq; u:=s(q);
		for j:=0 **to** 15 **do**
		u(j):=u(j)∨t(j); s(q):=u;
			[1, 0] &0 [1]
			[—1, 1] &0 [—1]
			[—2, 1] &0 [—1]

'+'	Побитное сложение (неэквивалентность)	t:=s(q); decq; u:=s(q);
		for j:=0 **to** 15 **do**
		u(j):=u(j)⊕t(j); s(q):=u;
			[1, 0] '+' [1]
			[—1, 1] '+' [—2]
			[—2, 1] '+' [—1]

SHL	Сдвиг влево	t:=s(q); **for** j:=1 **to** 15 **do** t(j):=t(j—1); t(0):=0; s(q):=t;	[1] SHL [2] [2] SHL [4] [—1] SHL [—2]
SHR	Сдвиг вправо	t:=s(q); **for** j:=0 **to** 14 **do** t(j):=t(j+1); t(15):=0; s(q):=t;	[1] SHR [0] [2] SHR [1] [—1] SHR [—1]
SWB	Перестановка байтов	t:=s(q); t(15:8):=:t(7:0); s(q):=t;	[1] SWB [256] [—1] SWB [31744]
NOP	Никакой операции		[ ] NOP [ ]

Продолжение

Мнемокод	Команда	Описание	Примеры
	Команды, управляющие последовательностью выполнения процедур		
IF— p	Выполнение по минусу	t:=s(q); decq; **if** t<0 **then** p;	[5, 0] IF— 2+ [5] [5, —4] IF— 2+ [7]
IF0 p	Выполнение по нулю	t:=s(q); decq; **if** t=0 **then** p;	[5, 4] IF0 D [5] [5, 0] IF0 D [ ]
IF+ p	Выполнение по плюсу	t:=s(q); decq; **if** t>0 **then** p;	[5, 0] IF+ 2— [5] [5, 4] IF+ 2— [3]
BR— p1 p2	Выбор по минусу	t:=s(q); decq; **if** t<0 **then** p1 **else** p2;	[5, —4] BR— T0 D [0] [5, 35] BR— T0 D [ ]
BR0 p1 p2	Выбор по нулю	t:=s(q); decq; **if** t=0 **then** p1 **else** p2;	[—5, 0] BR0 D ABS [ ] [—5, 7] BR0 D ABS [5]
BR+ p1 p2	Выбор по плюсу	t:=s(q); decq; **if** t>0 **then** p1 **else** p2;	[5, 7] BR+ T0 T1 [0] [5, 0] BR+ T0 T1 [1]
BRS p1 p2 p3	Выбор по знаку	t:=s(q); decq; **if** t<0 **then** p1 **else if** t=0 **then** p2 **else** p3;	[5, —3] BRS 1+ D 1— [6] [5, 0] BRS 1+ D 1— [ ] [5, 3] BRS 1+ D 1— [4]
BR a1 p1 a2 p2 ... ... aJ pJ ... ELSE pN	Выбор из N	t:=s(q); decq; **if** t=a1 **then** p1 **else if** t=a2 **then** p2 **else** ... **if** t=aJ **then** pJ **else** ... **else** pN;	[5, 7] BR 4 TO 3 T1 7 NEG ELSE D [—5]
RP p	Выполнять p бесконечное число раз	p; p; p; ...	[0] RP 2+ [2], [4], [6] ...
DO p	Выполнить p указанное вершиной число раз	t:=s(q); decq; p; p; ... p; (t раз)	[0, 3] DO 1+ [1], [2], [3]
EX	Выход из цикла		
EX—	Выход по минусу	t:=s(q); decq; **if** t<0 **then** EX;	

Продолжение

Мнемокод	Команда	Описание	Примеры
EX0	Выход по нулю	t:=s(q); decq; **if** t=0 **then** EX;	
EX+	Выход по плюсу	t:=s(q); decq; **if** t>0 **then** EX;	
EXT	Выход из цикла с вложенностью, указанной вершиной T	DO EX	
	Команды, порождающие процедуры и структуры данных		
:	Скомпилировать определяемую процедуру	: ⟨имя процедуры⟩ ⟨тело⟩;	: SGN BRS — 1 0 1 ;
;	Закончить компиляцию (конец определения)		
::	Зафиксировать в словаре имя определяемой далее процедуры	:: определение процедуры	:: :NOT BR0 1 0 ;
VAR x	Создать 16-битную переменную с именем x	x(15:0)	VAR TIME
n VCTR x	Создать (n + 1)-компонентный вектор 16-битных слов с именем x	x(0:n,15:0)	[8] VCTR ROW [ ]
n BVCTR x	Создать (n + 1)-компонентный вектор байтов с именем x	x(0:n,7:0)	24 BVCTR TAGS
CNST w w0 w1... wk ;	Создать вектор 16-битных констант с именем w	w(0:k,15:0)	CNST Q 37 —4 78 ;

Продолжение

Мнемокод	Команда	Описание	Примеры
BCNST b b0 ... bk ;	Создать вектор байтов-констант с именем b	b(0:k,7:0)	BCNST SW 3 5 2 9 15 ;
"⟨текст⟩"	Создать текстовый литерал, обеспечив засылку в стек адреса и длины текста	incq; s(q):=⟨адрес текста⟩ incq; s(q):=⟨длина текста⟩	"NAME" [⟨адрес⟩, 4]
k1 k2 ... kn n ARR x	Создать n-мерный массив 16-битных слов с именем x	x(0:k1,0:k2, ..., 0:kn,15:0)	4 9 2 ARR MTRX [ ]
FIX	Последующую структуру создать в области данных, сохраняемых вместе с кодом программы	FIX определение структуры	FIX VAR X [8] FIX VCTR ROW [ ]

Команды манипулирования именуемыми данными и памятью

x	Заслать в стек значение переменной x	incq; s(q):=x;	[ ] A2 [⟨значение A2⟩]
[j, k] x	Заслать в стек значение элемента массива x	k:=s(q); decq; j:=s(q); s(q):=x(j,k);	3 5 M [⟨значение M(3, 5)⟩]
' x	Заслать в стек адрес-указатель переменной x	incq; s(q):=addrx	[ ] ' A2 [⟨адрес A2⟩]
[j, k] ' x	Заслать в стек адрес-указатель элемента массива x	k:=s(q); decq; j:=s(q); s(q):=addrx(j, k);	4 8 ' G [⟨адрес G(4, 8)⟩]
" p	Заслать в стек адрес-указатель процедуры p	incq; s(q):=addrp;	[ ] " +[⟨адрес+⟩]

Продолжение

Мнемокод	Команда	Описание	Примеры
EXEC	Выполнить процедуру, указываемую вершиной стека t	t:=s(q); decq; call t;	[⟨адрес⟩] EXEC
@	Заменить адрес значением слова	t:=s(q); s(q):=m(t, 15:0)	[⟨адрес⟩] @ [⟨значение слова⟩]
@B	Заменить адрес значением байта	t:=s(q); s(q):=m(t,7:0)	[⟨адрес⟩] @B [⟨значение байта⟩]
[a,i] @BI	Заменить адрес значением бита, указанного вершиной	i:=s(q); decq; a:=s(q); s(q):=m(a, i);	[⟨адрес⟩, i] @BI [⟨значение бита⟩]
! x	Присвоить переменной x значение удаляемой из стека вершины	x:=s(q); decq;	[5] !MU [ ]
!T	Переслать значение подвершины по адресу, указанному вершиной	t:=s(q); decq; m(t,15:0):=s(q); decq;	[5, 2030] !T [ ]
!TB	Переслать младший байт подвершины по адресу, указанному вершиной	t:=s(q); decq; m(t,7:0):=s(q,7:0); decq;	[5, 635] !TB [ ]

[a, n, b] !SB	Скопировать n байтов, начиная с адреса a, в область с начальным адресом b	b:=s(q); decq; n:=s(q);
		decq; a:=s(q); decq;
		for i:=0 to n−1 do m(b+i):=m(a+i);
			[275, 9, 635] !SB [ ]

[a, i] !BI0	Присвоить 0 биту i по адресу a	i:=s(q); decq; a:=s(q); decq; m(a, i):=0;	[⟨адрес⟩, i] !BI0 [ ]
[a, i] !BI1	Присвоить 1 биту i по адресу a	i:=s(q); decq; a:=s(q); decq; m(a, i):=1;	[⟨адрес⟩, i] !BI1 [ ]

Продолжение

Мнемокод	Команда	Описание	Примеры
		Команды управления словарем	
GROW ¤v	Открыть для наращивания ¤v	Открыть для наращивания подсловарь ¤v	[ ] GROW ¤VT [ ]
FORGET ¤v	Удалить наращённое	Удалить наращённое, начиная с последнего GROW ¤v	[ ] FORGET ¤VT [ ]
CLEAR ¤v	Удалить незафиксированные имена из ¤v	В подсловаре ¤v удалить незафиксированные имена	[ ] CLEAR ¤VT [ ]
USE ¤v	Открыть для использования ¤v	Открыть для использования подсловарь ¤v	[ ] USE ¤WV [ ]
SHUT ¤v	Закрыть подсловарь ¤v	Сделать недоступным для использования подсловарь ¤v	[ ] SHUT ¤WV [ ]
ONLY ¤v	Только подсловарь ¤v	Сделать доступным для использования только подсловарь ¤v	[ ] ONLY ¤KEY [ ]
CANCEL	Отменить	Отменить последнее ONLY	[ ] CANCEL [ ]
?¤	Состояние словаря	Выдать на экран терминала данные, характеризующие текущее состояние словаря	[ ] ?¤ [ ]
UNDEF	Неопределенные слова	Выдать на экран терминала неопределенные слова	[ ] UNDEF [ ]

Продолжение

Мнемокод	Команда	Описание	Примеры
		Команды управления процессором	
RESTART	Перезапуск	Запуск процессора снова	[ ] RESTART [ ]
\G	Продолжить выполнение	Продолжение работы после останова на неопределенном слове	[ ] \G [ ]
E	Редактировать	Вызов редактора текстов	[ ] E [ ]
PF	Выполнить содержимое буфера	Скопировать содержимое буфера редактора текстов на вход процессора	[ ] PF [ ]
INT	Предназначить для прерывания	Определяемую далее процедуру предназначить для обработки прерывания	[ ] INT [ ]
[a] LINK p	Связать с вектором прерывания	Предназначенную для обработки прерывания процедуру p связать с адресом a вектора прерывания	[30] LINK SRV [ ]
TRAP g r	Объявить командное прерывание («ловушку») g, сопоставив ему процедуру реагирования r	При получении g, если не предписано иное, выполнить процедуру r	[ ] TRAP OV NOP [ ]
ON g p	Перехват командного прерывания g	Подменить установленную для g реакцию r процедурой p	[ ] ON OV RESTORE [ ]
EON g p	Перехват командного прерывания g с выходом из процедуры, содержащей EON	Сохранить текущее значение q указателя стека, подменив установленную для g реакцию выходом из процедуры, непосредственно содержащей EON с восстановлением q и выполнением процедуры p	[ ] EON OV STOP [ ]

Продолжение

Мнемокод	Команда	Описание	Примеры
?I	Информация о системе	Выдать на терминал дату генерации системы и объем свободной памяти	[ ] ?I [ ]
B2	Двоичный ввод/вывод	Установить режим двоичного ввода/вывода	[ ] B2 [ ]
B8	Восьмеричный ввод/вывод	Переключение на восьмеричный ввод/вывод	[ ] B8 [ ]
B10	Десятичный ввод/вывод	Переключение на десятичный ввод/вывод	[ ] B10 [ ]
B16	Шестнадцатеричный ввод/вывод	Переключение на шестнадцатеричный ввод/вывод	[ ] B16 [ ]
		Команды ввода/вывода с терминала	
.	Отобразить на экран копию значения вершины	displ (s(q));	[5] . [5]
..	Отобразить на экран копии значений элементов стека, начиная с вершины	displ(s(q), s(q—1), s(q—2), ...);	[... 5, 4] .. [... 5,4]
."⟨текст⟩"	Отобразить текст на экран	displ(⟨текст⟩);	[ ] ."YES" [ ]
TIN	Ввод числа с терминала в стек с отображением на экран	incq; s(q):=⟨число⟩; displ (s(q));	[ ] TIN [—487]
TIB	Ввод байта-литеры с терминала в стек с отображением на экран	incq; s(q):=0; s(q, 7:0):=⟨код литеры⟩; displ (⟨литера⟩);	[ ] TIB [64]
TRB	Ввод байта-литеры с терминала в стек без отображения на экран	incq; s(q):=0; s(q,7:0):=⟨код литеры⟩	[ ] TRB [95]

Продолжение

Мнемокод	Команда	Описание	Примеры
[u, t] TON	Вывод на экран терминала числа из стека в поле, длина которого указана вершиной	t:=s(q); decq; u:=s(q); decq; displ (⟨u в поле t⟩);	[47, 6] TON [ ]
[t] TOB	Вывод на экран терминала литеры, код которой содержится в младшем байте вершины	t:=s(q); decq; displ(⟨литера t(7:0)⟩);	[98] TOB [ ]
CR	Возврат каретки	Перевод курсора в начало очередной строки	[ ] CR [ ]
SP	Пробел	Перевод курсора на одну позицию вправо	[ ] SP [ ]
BELL	Звонок	Звуковой сигнал на терминале	[ ] BELL [ ]
[a, n] TIS	Ввод с терминала n байтов в память, начиная с адреса a	for i:=0 to n—1 do m(a +i,7:0):=⟨литера⟩;	[3064,6] TIS [ ]
[a, n] TOS	Вывод на экран терминала n байтов из памяти, начиная с адреса a	for i:=0 to n—1 do displ (m(a+i,7:0));	[4450, 8] TOS [ ]
		Команды управления памятью на гибких дисках	
DX0	Работа с дисководом 0	Переключиться на дисковод 0	[ ] DX0 [ ]
DX1	Работа с дисководом 1	Переключиться на дисковод 1	[ ] DX1 [ ]
DNP	Начальная установка диска	Установить в исходную позицию	[ ] DNP [ ]
[k] DSET	Установка на начало сектора k	Установить на сектор k	[24] DSET [ ]

Продолжение

Мнемокод	Команда	Описание	Примеры
DOON	Переключение в режим вывода	Переключиться на вывод	[ ] DOON [ ]
DOOFF	Отмена режима вывода	Отменить режим вывода	[ ] DOOFF [ ]
DIB	Ввод байта в стек	incq; s(q):=⟨байт⟩;	[ ] DIB [124]
DOB	Вывод байта из стека	t:=s(q,7:0); decq; discout(t);	[56] DOB [ ]
[a, n] DIS	Ввод с диска n байтов в память, начиная с адреса a	**for** i:=0 **to** n—1 **do** m(a+i, 7:0):=байт;	[789, 64] DIS [ ]
[a, n] DOB	Вывод на диск n байтов из памяти, начиная с адреса a	**for** i:=0 **to** n—1 **do** discout(m(a+i, 7:0));	[643, 80] DOS [ ]
[f] NUST	Установка на начало файла f	Установить на файл f	[23] NUST [ ]
DIE f	Вывод в буфер редактора из файла f	Скопировать содержимое файла f в буфер редактора	[ ] DIE [9]
DOE f	Вывод в файл f из буфера редактора	Скопировать содержимое буфера редактора в файл f	[ ] DOE [32]
LOAD f	Загрузка файла f в качестве входного потока	Скопировать файл f на вход процессора	[ ] LOAD [25]
SAVE	Сохранить систему на диске	Вывести на диск систему в коде	[ ] SAVE [ ]
CAT	Выдать каталог диска	Вывести на экран терминала каталог диска, находящегося в работе	[ ] CAT [ ]

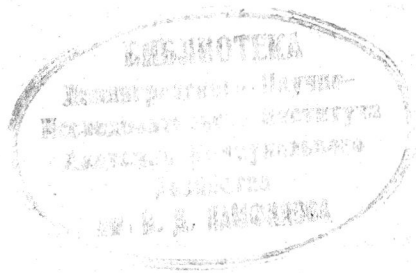
Продолжение

Мнемокод	Команда	Описание	Примеры
	Команды перфолентного ввода/вывода		
RDT	Ввести первый байт с ПЛ	Считать первый значащий байт с перфоленты; incq; s(q):=⟨байт⟩;	[ ] RDT [5]
PNT	Вывести t незначащих байтов на ПЛ	t:=s(q); decq; вывести на перфоленту t незначащих строк-байтов	
RDB	Ввести байт с ПЛ	incq; s(q):=⟨байт⟩;	[20] PNT [ ]
PNB	Вывести байт на ПЛ	t:=s(q); decq; punch(t);	[ ] RDB [85]
[a, n] RDS	Ввести с ПЛ n байтов в память, начиная с адреса a	**for** i:=0 **to** n—1 **do** m(a+i, 7:0):=⟨байт⟩;	[93] PNB [ ]
[a, n] PNS	Вывести на ПЛ n байтов из памяти, начиная с адреса a	**for** i:=0 **to** n—1 **do** punch(m(a+i,7:0));	[600, 24] RDS [ ]
RDE	Ввод с ПЛ в буфер редактора	Ввести с ПЛ в буфер редактора	[600, 24] PNS [ ]
PNE	Вывод из буфера редактора на ПЛ	Вывести содержимое буфера редактора на ПЛ	[ ] RDE [ ]
TLOAD	Загрузка с ПЛ в качестве входного потока	Ввести с ПЛ на вход процессора	[ ] PNE [ ]
TSAVE	Сохранить систему на ПЛ	Вывести на ПЛ систему в коде	[ ] TLOAD [ ]
			[ ] TSAVE [ ]

Продолжение

Команды вывода на принтер

Мнемокод	Команда	Описание	Примеры
[u, t] LPN	Вывод числа из стека на принтер в поле, длина которого указана вершиной	t:=s(q); decq; u:=s(q); decq; print(<u в поле t>);	[475, 8] LPN [ ]
[t] LPB	Вывод литеры из стека на принтер	t:=s(q); decq; print(<литера t(7:0)>);	[93] LPB [ ]
LPCR	Новая строка	Печатать с начала очередной строки	[ ] LPCR [ ]
LPSR	Пробел	Перевод печатающей головки на одну позицию вправо	[ ] LPSP [ ]
LPFF	Новая страница	Печатать с начала очередной страницы	[ ] LPFF [ ]
[n] LPT	Позиция n	Печатать с позиции n текущей строки	[43] LPT [ ]
[a, n] LPS	Вывод на принтер n байтов-литер	Вывести на принтер n байтов-литер из памяти, начиная с адреса a	[844, 40] LPS [ ]
LPE	Распечатать буфер редактора	Вывести на принтер содержимое буфера редактора текстов	[ ] LPE [ ]



Николай Петрович Брусенцов

Микрокомпьютеры

Редактор *Л. Г. Силкова*
Художественный редактор *Т. Н. Кольченко*
Технический редактор *И. Ш. Аксельрод*
Корректор *Л. С. Сомова*

ИБ № 12574

Сдано в набор 17.04.85. Подписано к печати 13.08.85
Т-16699. Формат 84×108¹/₃₂. Бумага тип. № 2.
Гарнитура обыкновенная. Печать высокая.
Усл. печ. л. 10,92. Усл. кр.-отт. 11,13. Уч.-изд. л. 12,87.
Тираж 74 000 экз. Заказ № 1344. Цена 50 коп.

Ордена Трудового Красного Знамени издательство «Наука»
Главная редакция физико-математической литературы
117071 Москва В-71, Ленинский проспект, 15

2-я типография издательства «Наука».
121099 Москва Г-99, Шубинский пер., 6

ИЗДАТЕЛЬСТВО «НАУКА»
ГЛАВНАЯ РЕДАКЦИЯ
ФИЗИКО-МАТЕМАТИЧЕСКОЙ ЛИТЕРАТУРЫ
117071 Москва В-71, Ленинский проспект, 15

ГОТОВИТСЯ К ПЕЧАТИ

ВОЕВОДИН В. В. **Математические модели и методы в параллельных процессах.**

Посвящена одной из самых молодых областей современной прикладной математики — исследованию математических моделей параллельных вычислительных систем и построению параллельных численных методов. Рассмотрены общие принципы отображения алгоритмов на архитектуру вычислительных систем. Исследованы параллельные структуры алгоритмов и программ. Построены математические модели некоторых параллельных систем. Изучены общие структуры информационных потоков, проходящих через такие системы. Указаны особенности реализации параллельных вычислительных алгоритмов. Обсуждены математические проблемы освоения супер-ЭВМ.

Для специалистов, решающих задачи на современных ЭВМ, и для студентов вузов.