PDP-11 и VAX-11

PDP-11 и VAX-11

АРХИТЕКТУРА ЭВМ

и программирование
на языке ассемблера

PDP-11 и VAX-11
Архитектура ЭВМ и программирование на языке ассемблера

Computer Organization and Assembly Language Programming for the PDP-11 and VAX-11

Wen C. Lin
University of California, Davis

1817

HARPER & ROW, PUBLISHERS, New York
Cambridge, Philadelphia, San Francisco,
London, Mexico City, São Paulo, Singapore, Sydney

PDP-11 и VAX-11
Архитектура ЭВМ и программирование на языке ассемблера

В. Лин

Перевел с английского **В. М. Северьянов**

МОСКВА «РАДИО И СВЯЗЬ» 1989

ББК 32.973
Л59
УДК 519.68

Редакция переводной литературы

Л $\frac{2404000000\text{-}094}{046\,(01)\,\text{-}89}$ 137-89

ISBN 5-256-00299-6 (рус.)
ISBN 0-06-04406L-9 (англ.)

ПРЕДИСЛОВИЕ К РУССКОМУ ИЗДАНИЮ

Как видно из названия, книга посвящена архитектуре и программированию на языке ассемблера двух широко распространенных семейств ЭВМ фирмы DEC: PDP-11 и VAX-11. Следует заметить, что эти ЭВМ имеют архитектуру традиционного фон-неймановского типа, а программирование на языке ассемблера представляет собой трудоемкий процесс, требующий достаточно высокой квалификации. Поэтому может возникнуть вопрос, актуальна ли подобная книга в настоящее время?

Читателю, без сомнения, известно, что становятся доступными новые языки программирования высокого уровня, позволяющие в существенной степени избежать зависимости от ЭВМ и облегчить процесс программирования. Вслед за Алголом, Фортраном и Коболом распространились такие языки нового поколения, как Паскаль, Си, Ада. Наряду с традиционным процедурным подходом к программированию развиваются методы объектно-ориентированного взаимодействия с ЭВМ, функциональное и логическое программирование. Например, языки Лисп и Пролог очень удобны при создании систем искусственного интеллекта. Разрабатываются соответствующие архитектуры ЭВМ, в том числе для систем параллельной обработки и распределенных вычислительных систем. Широкое распространение персональных компьютеров (профессиональных, учебных, бытовых) привлекает к использованию ЭВМ непрофессиональных программистов. Еще более простыми в использовании будут ЭВМ пятого поколения, с которыми можно будет взаимодействовать и на естественном языке.

Тем не менее, интерес к архитектуре ЭВМ и программированию на уровне машинных кодов и языка ассемблера не только не угасает, но явно растет по двум причинам. Первая причина заключается в том, что доступность использования ЭВМ для непрофессиональных программистов достигается ценой соответствующих (и весьма немалых) усилий профессиональных программистов, создающих системное программное обеспечение, для которых знание архитектуры ЭВМ и умение программировать на языке ассемблера просто необходимы. Происходящий ныне процесс компьютеризации неизбежно увеличивает число специалистов очень высокой квалификации – системных программистов. Вторая причина обусловлена бурным развитием микропроцессорной техники. Современные микропроцессоры обладают такими возможностями, которые соизмеримы, а порою и превосходят возможности вычислительных систем, созданных недавно, причем обладают значительно меньшими габаритами, потреблением энергии и стоимостью. Программирование микропроцессоров и однокристальных ЭВМ, встраиваемых в какое-либо оборудование, выполняется, как правило, на языке ассемблера, а применение их требует детального знания архитектуры ЭВМ. Выпускаются, в частности, и микропроцессоры с набором команд ЭВМ PDP-11.

За последние годы в нашей стране издано несколько книг по архитектуре и программированию ЭВМ семейства PDP-11. Эти книги моментально исчезают с прилавков магазинов. Продолжается производство в странах СЭВ целого ряда ЭВМ, программно совместимых с ЭВМ PDP-11, а спрос на литературу по их использованию остается не-

удовлетворенным. Налаживается производство ЭВМ, подобных ЭВМ VAX-11, книг же по этой тематике на русском языке практически нет. Поэтому, без сомнения, данная книга найдет своего читателя.

Книгу отличают лаконичность изложения и большое число примеров. Это делает ее удобной для тех читателей, которые сталкиваются с языком ассемблера не впервые и хотят освоить еще одну, новую для них ЭВМ или испытывают потребность в обновлении уже имеющих знаний. Новичкам одной только этой книги будет, по-видимому, недостаточно, хотя и им она принесет несомненную пользу.

Основной материал книги касается семейства ЭВМ PDP-11, ЭВМ типа VAX-11 отведена одна глава. Рассматривая ЭВМ этих двух типов, автор старается показать, что знание ЭВМ одного типа облегчает освоение ЭВМ другого типа. ЭВМ VAX-11, которая имеет центральный процессор большей разрядности, более мощный набор команд и развитые средства поддержки виртуальной памяти, представляет собой логическое продожение и развитие ЭВМ PDP-11, что дает пример последовательного и плодотворного применения системного подхода к созданию семейств совместимых ЭВМ.

В. М. Северьянов

Моей жене Шун-Лин, родителям, миссис Б. Лин и старшей сестре Хи-Кван

ПРЕДИСЛОВИЕ

С появлением микропроцессоров потребовалось включить вопросы по организации ЭВМ и программированию на языке ассемблера в учебные программы для студентов старших курсов – будущих инженеров по вычислительной технике и электротехническим дисциплинам. По этой теме был выпущен целый ряд книг. В одних книгах упор делается на программирование для какой-либо одной мини- или микроЭВМ, в других описывается техника программирования применительно к различным типам микроЭВМ.

Однако при чтении этих книг новички сталкиваются с рядом проблем. Первая: в большинстве книг отсутствует информация об элементарных принципах работы базовой ЭВМ; вторая: в большей части книг нет согласованного и последовательного изложения взаимосвязи между системными аппаратными средствами, системным программным обеспечением и прикладными программами пользователей; третья: отсутствует подробное изложение основ процесса разработки пользовательских программ (не рассматриваются, например, режимы адресации машинных инструкций, форматы документирования программ или функции псевдоинструкций); четвертая: приводимые примеры часто недостаточно ясны, хотя, по моему убеждению, понятные, пусть даже тривиальные примеры весьма быстро и эффективно способствуют тому, что студенты приобретают уверенность в собственных силах.

Конечно, при обучении этому предмету нельзя оговорить абсолютно все детали. Студенты (если они закончат обучение) в конце концов сами в них разберутся. Но я обнаружил, что детали и основы сводят к минимуму ненужную путаницу, особенно в самом начале курса. Если курс по программированию на языке ассемблера и организации ЭВМ должен стать центральным в учебной программе по электротехнике и вычислительной технике, то мы должны обучать студентов основам теории, начиная с режимов адресации и арифметики дополнений. Кроме того, как правило, изучив конкретную ЭВМ (предпочтительно пользующуюся популярностью и достаточно сложную и мощную, такую как ЭВМ семейства PDP-11), студент легко может освоить любую другую машину, ознакомившись с руководствами для пользователей. А что касается новичков, то одна книга, описывающая многие ЭВМ различных типов, только больше запутает их. По этой причине, чтобы помочь студентам в изучении новейшей ЭВМ, завоевывающей сейчас популярность, мы включили в текст вводную главу (гл 10), описывающую систему VAX-11.

Материал книги построен так, чтобы дать детальное изложение основ теории программирования и уделить равное внимание архитектуре аппаратных средств, системному программному обеспечению и программам, разрабатываемым пользователем. Мы не ставим перед собой цель научить студентов разнообразным методам программирования на языке ассемблера (по этой теме уже есть несколько книг), а стремимся перекинуть мостик между начинающими, которым нужны основы, и опытными студентами, интересующимися методами прикладного программирования на языке ассемблера. Содержание этой книги использовалось в качестве конспекта курса по структуре ЭВМ и про-

граммированию на языке ассемблера. Этот курс преподавался на базе лаборатории, оборудованной вычислительной сетью на основе ЭВМ PDP-11/34 и LSI-11 с операционной системой RSX-11M, где студенты приобретали опыт практической работы.

В гл. 1 изложение материала начинается с основных понятий программирования. С помощью простого примера поясняется, как работает простая цифровая ЭВМ. В этом примере дается понятие инструкций, закодированных посредством двоичных чисел. На основе этого примитивного понятия вводится базовая архитектура ЭВМ фон-неймановского типа. Далее следует обзор вычислительной системы, дополненной диском и терминалом и загруженной системным и прикладным программным обеспечение, и краткое описание того, каким образом все это работает вместе. В гл. 2 дана системная организация ЭВМ PDP-11 с точки зрения пользователя. В гл. 3 описаны основы представления информации в цифровой ЭВМ, такой как PDP-11. Представление информации подразделяется на представление чисел и представление инструкций. В этой главе описывается арифметика дополнений для двоичных, восьмеричных и шестнадцатеричных чисел, а также некоторое подмножество из набора инструкций ЭВМ PDP-11. Глава завершается простым примером, в котором все компоненты увязываются вместе с целью проиллюстрировать, каким образом базовые элементы работают вместе как одна "бригада". Глава 4 посвящена набору инструкций. В ней сделана попытка детально описать мощные режимы адресации, применяемые в семействе PDP-11. Глава 5 посвящена программированию подпрограмм. Она начинается с основного понятия подпрограммы, а далее рассматриваются различные форматы написания подпрограмм. О ключевых проблемах, таких как передача параметров и связывание главной программы с подпрограммой, рассказывается с помощью примеров. В гл. 6 вводится понятие макросов, объясняется различие между подпрограммами и макроинструкциями, и, чтобы показать, как определенные в системе макросы, определенные пользователем, и условные макросы могут применяться при разработке прикладного программного обеспечения, снова прибегаем к примерам. В гл. 7 дается описание важных методов программирования ввода-вывода. В книге продемонстрирована согласованная работа аппаратной платы ввода-вывода и программы ввода-вывода на языке ассемблера. Чтобы показать взаимодействие между внешним устройством или реальным внешним миром и цифровой ЭВМ, используются примеры. Главы 8 и 9 посвящены соответственно прерываниям, ловушкам и прямому доступу в память (ПДП). Каждая тема иллюстрируется практическими примерами, в которых тщательно разбираются наиболее важные атрибуты. Глава 10 – это введение в систему VAX-11. В заключительную главу 11 помещены лабораторные упражнения и приведены примеры задач.

Автор признателен профессору Х. Х. Лумису младшему, который впервые несколько лет тому назад познакомил его с этой интересной темой и предоставил ему возможность заниматься рядом проблем в лаборатории Калифорнийского университета (г. Дейвис). Благодарности заслуживают также мои друзья студенты, участвовавшие в преподавании этого курса. В подготовку рукописи этой книги внесли свой вклад, в частности, Л. Соуренсон, С. Цзю и мисс Мэй-Ксин Цзяо. Глубокую признательность выражаю Д. Уиллигу из фирмы Hatpet & Row и профессору Р. Алгази из Калифорнийского университета за оказанную ими поддержку. Наконец, автор хотел бы выразить свою благодарность фирме Digital Equipment Corporation за сотрудничество и разрешение использовать информацию из публикаций фирмы по ЭВМ типа PDP-11. В частности, материал, помещенный в приложениях Б–Г (информация о наборе инструкций ЭВМ PDP-11), является частичной перепечаткой из серии руководств по микроЭВМ "Процессоры микроЭВМ (1978–79)" и "Справочная карта по LSI-11 и PDP-11/03" этой фирмы.

В. Лин

ГЛАВА 1

ВВЕДЕНИЕ

1.1. ОСНОВНЫЕ ПРИНЦИПЫ РАБОТЫ ЭВМ

ОРГАНИЗАЦИЯ ЭВМ

Организация ЭВМ напоминает физическую организацию человека. Хотя люди обладают разными личными качествами или различаются цветом кожи, у каждого человека есть мозг, глаза, уши, рот. Фактически головной мозг функционирует как некий банк памяти, мозжечок служит устройством управления, а уши и рот обеспечивают взаимодействие с внешним миром. Вычислительную машину обычно представляют состоящей из четырех основных частей: памяти, арифметического и логического устройства, устройства управления и устройства ввода-вывода. Эти устройства связываются друг с другом с помощью электрических сигналов, передаваемых по шине, а шина – это не что иное, как пучок проводов, используемых для переноса электрических сигналов. Через устройство ввода-вывода они взаимодействуют с внешним миром, а из внешнего мира взаимодействие с ними осуществляется с помощью периферийного оборудования, такого как телевизионные камеры (глаза), громкоговорители (рот), микрофоны (уши) или чаще всего терминал с клавиатурой и электронно-лучевой трубкой (ЭЛТ), похожей на телевизионную, или перфоратор-считыватель перфокарт либо перфоленты. Поскольку внешнее оборудование зависит от области приложения и варьируется от системы к системе, в этой книге оно рассматриваться не будет.

На рис. 1.1 показана блок-схема некоторой базовой цифровой ЭВМ и четыре основных ее компонента. **Устройство управления** совместно с арифметическим и логическим устройством обычно называют **центральным процессором** (ЦП); это сердце всей системы. Устройство управления определяет последовательность операций и устанавливает пути транспортировки данных внутри системы. **Арифметико-логическое устройство** (АЛУ) выполняет арифметические и логические операции. **Устройство памяти** хранит информацию. Оно функционирует подобно множеству почтовых ящиков. Информация сохраняется в ячейке, обозначаемой числом, которое называется **адресом** этой ячейки, а память обычно определяется адресом и содержимым. **Содержимое** – это строка двоичных чисел, единиц и нулей, называемая машинным словом. Чтобы запомнить информацию в определенной ячейке или извлечь ее оттуда, ЦП посылает определенный адрес и для выполнения операции генерирует командный сигнал записи или чтения данных соответственно. При записи данных ЦП поставляет содержимое, при чтении он его получает. Устройство ввода-вывода обеспечивает порты для связи с внешним миром. Благодаря значительному прогрессу в области твердотельной технологии базовая система, содержащая память, устройство ввода-вывода и центральный процессор, может быть размещена всего в одной или двух полупроводниковых электронных микросхемах.

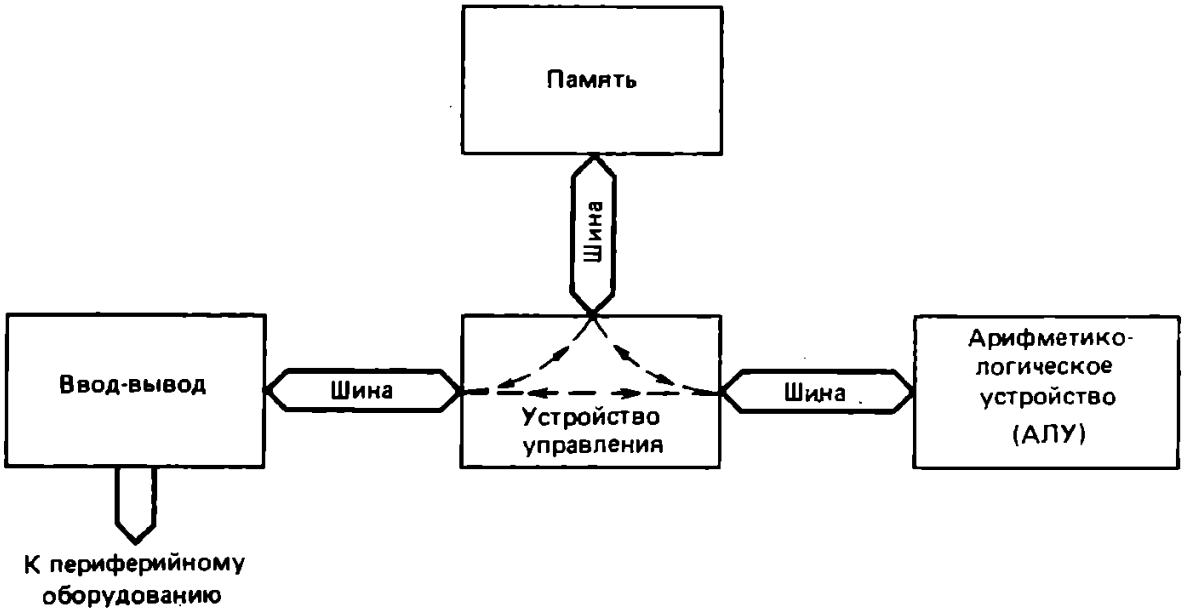


Рис. 1.1. Блок-схема базовой ЭВМ

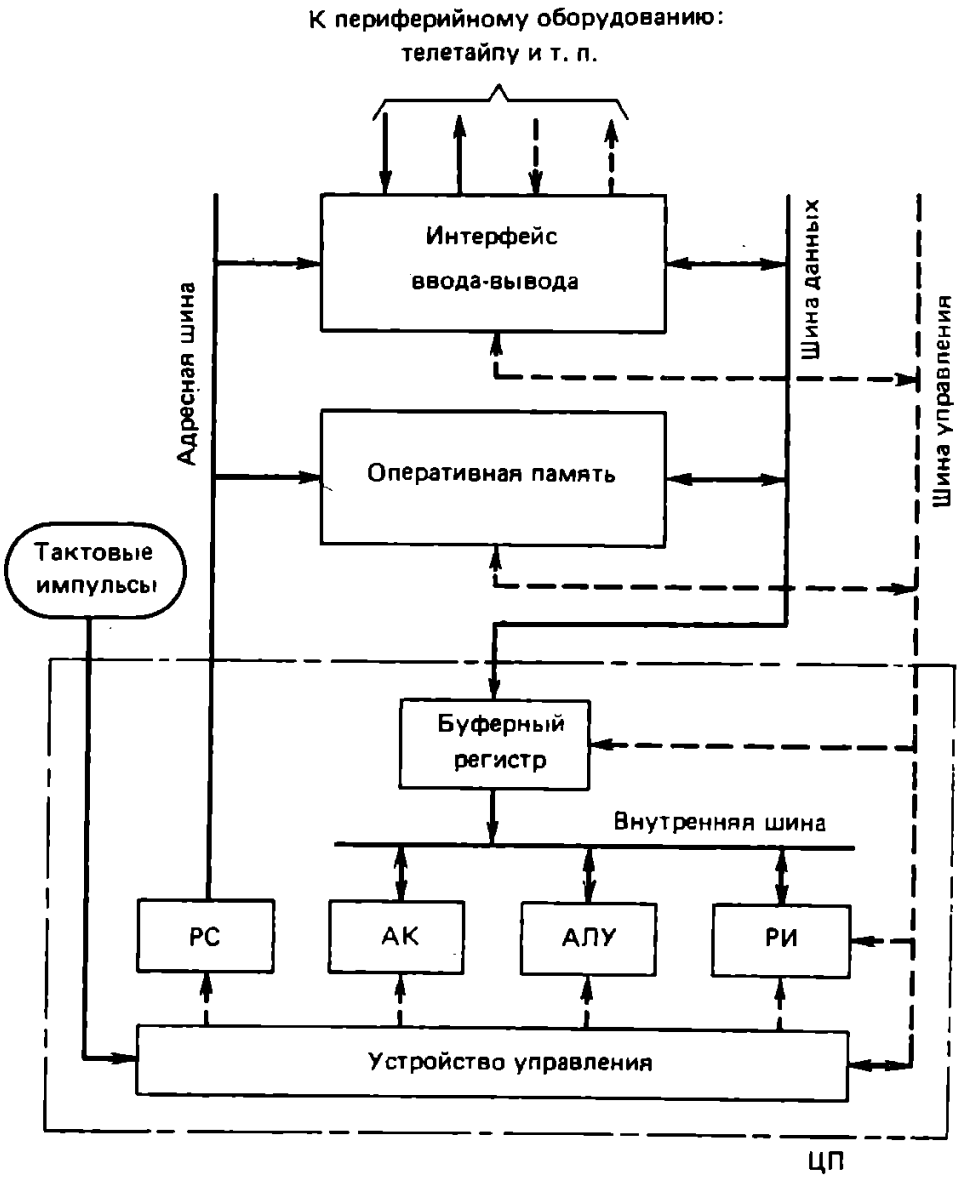


Рис. 1.2. Элементарная операционная блок-схема базовой ЭВМ

На рис. 1.2 представлена фактически такая же блок-схема, как на рис. 1.1, но пути данных и линии управления показаны раздельно (управление обозначено штриховыми линиями), а АЛУ и устройство управления собраны вместе в штрихпунктирном квадрате под названием ЦП. Чтобы удобнее было описывать работу, ЦП показан более подробно. Обратите внимание, что имеются четыре **регистра**. Они представляют собой наборы сгруппированных триггеров, используемые для запоминания двоичных данных. О регистре можно думать, что это "почтовый ящик", который применяется для временного хранения информации. Поскольку каждый регистр выполняет определенную функцию, ему присваивается индивидуальное имя: РС – программный счетчик; АК – аккумулятор; РИ – регистр инструкции; БР – буферный регистр. Большинство центральных процессоров располагают более чем шестью регистрами, но для простоты мы здесь используем минимальное число регистров – четыре.

Устройство управления – это нерегулярная логическая схема, синхронизированная тактовым генератором. Она генерирует управляющие импульсы и потенциалы для всей системы, открывает пути транспортировки данных в нужной последовательности в соответствии с информацией в РИ. Центральный процессор получает из оперативной памяти информацию и помещает ее в РИ. Затем устройство управления интерпретирует эту информацию, генерирует соответствующие электрические сигналы и выполняет инструкцию. Регистры АК и БР используются под временные данные или для сохранения информации с целью обеспечения этих действий. Работу ЭВМ поясним на простом примере.

Пример

Задача. Мы хотим разработать управление главным входом в здание так, чтобы вход открывался только для человека по имени Джон К.Смит.

Разработка. Воспользуемся ЭВМ, операционная блок-схема которой показана на рис. 1.2, с телетайпом в качестве единственного периферийного устройства ввода-вывода. Как отмечалось выше, устройство памяти подобно банку почтовых ящиков. Каждому ящику в качестве его адреса приписывается некоторое число. Внутри ящика находится слово, называемое содержимым этого почтового ящика. Давайте теперь загрузим память так, чтобы она содержала сообщения, перечисленные в табл. 1.1. Если посетителю необходимо пройти в здание, то он (или она) должен напечатать на клавиатуре телетайпа сообщение "Открыть вход".

Пусть аппаратура будет разработана так, что при включении питания ЭВМ автоматически сбрасывает или очищает все свои внутренние регистры. Тогда мы можем ожидать, что регистр РС содержит нуль. Текущее значение, находящееся в регистре РС, – это текущий адрес ящика памяти, из которого ЭВМ будет извлекать сообщение. Регистр РС устроен так, чтобы его содержимое увеличивалось на единицу при приеме командного импульса увеличения от устройства управления (для удобства устройство управления далее будем называть **контроллером**). Он сбрасывается (или очищается) в 0 при приеме командного импульса сброса. Содержимое регистра РС – это всегда адресное значение слова памяти (почтового ящика), из которого ЦП будет считывать сообщение или в которое он будет производить запись. Именно поэтому этот регистр называют программным счетчиком или указателем.

Давайте теперь проследим работу контроллера. Первоначально регистр РС содержит 0. Содержимое памяти с адресом 0 будет переслано в регистр БР, затем в регистр РИ. Контроллер интерпретирует сообщение, предписывающее в данном случае извлечь сообщение по следующему адресу 1, помещает его в аккумулятор АК и выполняет его. Другими словами, контроллер получает данные по адресу 1 и сохраняет их в регистре АК. В результате в регистре АК находятся данные "Открыть вход". В то же время

Таблица 1.1

Адрес	Содержимое
0	Загрузить регистр АК данными из следующего ящика
1	"Открыть вход"
2	Вычесть данные (сообщение), находящиеся в регистре интерфейса ввода-вывода, из данных регистра АК
3	Если результат равен нулю, извлечь содержимое следующего ящика; иначе извлечь содержимое ящика с адресом 0
4	Загрузить регистр АК данными из следующего ящика
5	"Напечатайте, пожалуйста, свое имя"
6	Дать команду телетайпу распечатать сообщение, находящееся в регистре АК
7	Загрузить регистр АК данными из следующего ящика.
8	"Джон К. Смит"
9	Ожидать 30 с
10	Вычесть входные данные из регистра АК
11	Если результат равен нулю, извлечь содержимое следующего ящика; иначе извлечь содержимое ящика с адресом 0
12	Включить мотор, который откроет входную дверь
13	Ожидать 30 с
14	Включить мотор, который закроет входную дверь
15	Извлечь содержимое ящика с адресом 0

контроллер посылает к регистру РС импульс увеличения, и содержимое регистра РС увеличивается до 2. Извлекая информацию по адресу 2, контроллер обнаруживает, что ему нужно сделать операцию ввода-вывода. Тогда предпринимаются следующие действия. Содержимое регистра РС увеличивается до 3, и входные данные "Открыть вход", которые были напечатаны на телетайпе посетителем, вычитаются из содержимого регистра АК. Вспоминаем, что мы уже поместили в регистр АК сообщение "Открыть вход", поэтому результатом вычитания будет 0. Это вычитание выполняется АЛУ, а результат (в данном случае 0) запоминается в регистре АК. Теперь извлекается и запоминается в регистре РИ сообщение по адресу 3. Выполняя текущее сообщение, находящееся в РИ, контроллер проверяет содержимое регистра АК. Если оно равно нулю, то извлекается и запоминается в регистре РИ информация по адресу 4. В противном случае содержимое регистра РС сбрасывается в 0. Обратите внимание, что это сообщение либо вызывает продолжение работы ЭВМ, либо приводит к возврату назад к исходной точке и к повторению процесса проверки того, что напечатал посетитель.

Предположим, что результатом последнего вычитания является 0. Тогда будет извлечено и выполнено сообщение по адресу 4. Обратите внимание, что сообщения в ячейках с адресами 4 и 5 близки к сообщениям в ячейках 0 и 1, за исключением того, что после выполнения в регистре АК появляется сообщение "Напечатайте, пожалуйста, свое имя". Процесс продолжается до тех пор, пока содержимое регистра РС не станет равным 6. Просматривая сообщение по адресу 6, контроллер находит, что он опять должен сделать операцию ввода-вывода, на этот раз вывести данные, которые были запомнены в регистре АК. Он выводит данные из регистра АК (теперь это сообщение "Напечатайте, пожалуйста, свое имя"), дав команду телетайпу распечатать это сообщение. В то же время содержимое регистра РС было увеличено до 8. Сообщение в ящике 8 предписывает ЭВМ загрузить в регистр АК данные "Джон К. Смит". Сообщение в ящике 9 задерживает работу машины на 30 с, чтобы дать возможность посетителю напечатать свое имя. Сообщение по адресу 10 проверяет, зовут ли посетителя Джон К. Смит. Если да, то процесс продолжается. В противном случае регистр РС сразу будет установлен в 0, а весь процесс повторится с самого начала.

ПРОГРАММНОЕ ОБЕСПЕЧЕНИЕ

Очевидно, что большую часть работы выполняет контроллер. Он декодирует и интерпретирует сообщение из регистра РИ и генерирует необходимые управляющие импульсы в соответствующие моменты времени для транспортировки данных из одного места в другое или для инициализации последовательностей операций, таких как арифметические операции или операции ввода-вывода. С точки зрения разработчика аппаратуры (или логики), контроллер – это не что иное, как последовательная логическая схема. И хотя процедуры разработки могут быть сложными и утомительными, если есть ясное представление о проблеме, то конструкция контроллера вполне очевидна. Кроме того, контроллер обычно разрабатывается и изготавливается как некое аппаратное средство, поэтому пользователю приходится заботиться лишь о том, что должно быть записано в оперативную память.

Положим для удобства, что контроллер разработан только для декодирования (интерпретации) шестнадцати записанных в памяти сообщений из вышеприведенного примера. Нам известно, что содержимое памяти – это либо единицы, либо нули, поэтому наши сообщения должны быть закодированы с помощью двоичных комбинаций. Поскольку у нас только 16 сообщений, для их кодирования можно воспользоваться четырехбитовыми двоичными числами. Другими словами, если каждая ячейка составляется из четырех элементов памяти, нам нужно только 16 ячеек (слов) памяти. Такая память в присущей вычислительной технике терминологии называется памятью 16 X 4. Но среди сообщений имеются и дубликаты: сообщения в ячейках 0, 4 и 7; 2 и 10; 3 и 11; 9 и 13. Следовательно, их двоичные коды должны быть идентичными. Предположим, что разработчик контроллера закодировал сообщения так, как показано в табл. 1.2.

Таблица 1.2

Память		Комментарии
Адрес в двоичном коде	Содержимое в машинном коде	Содержимое на русском языке
0 0 0 0	0 0 0 1	Загрузить регистр АК данными из следующего ящика
0 0 0 1	0 0 1 0	"Открыть вход"
0 0 1 0	0 0 1 1	Вычесть данные (сообщение), находящиеся в регистре интерфейса ввода-вывода, из данных регистра АК
0 0 1 1	0 1 0 0	Если результат равен нулю, извлечь содержимое следующего ящика; иначе извлечь содержимое ящика с адресом 0
0 1 0 0	0 0 0 1	Загрузить регистр АК данными из следующего ящика
0 1 0 1	0 1 0 1	"Напечатайте, пожалуйста, свое имя"
0 1 1 0	0 1 1 0	Дать команду телетайпу распечатать сообщение, находящееся в регистре АК
0 1 1 1	0 0 0 1	Загрузить регистр АК данными из следующего ящика
1 0 0 0	0 1 1 1	"Джон К. Смит"
1 0 0 1	1 0 0 0	Ожидать 30 с
1 0 1 0	0 0 1 1	Вычесть входные данные из регистра АК
1 0 1 1	0 1 0 0	Если результат нулевой, извлечь содержимое следующего ящика; иначе извлечь содержимое ящика с адресом 0
1 1 0 0	1 0 0 1	Включить мотор, который откроет входную дверь
1 1 0 1	1 0 0 0	Ожидать 30 с
1 1 1 0	1 0 1 0	Включить мотор, который закроет дверь
1 1 1 1	1 0 1 1	Извлечь содержимое ящика с адресом 0

Это означает, что есть 16 сообщений, но для их кодирования необходимо только 11 четырехбитовых комбинаций. Контроллер декодирует двоичные сообщения и предпринимает соответствующие действия. Но существует очевидное противоречие. Человек предпочитает видеть сообщения в их словесном выражении, как это показано в табл. 1.2, но ЭВМ "понимает" только "язык" двоичных бит. В нашем примере ЭВМ понимает 11 и только 11 "слов" в виде двоичных комбинаций и выполняет только эти 11 действий.

Совершенно ясно, что когда потребуется выполнить действия или задачи, отличные от этих 11, то контроллер придется разработать заново. Если хотим просто изменить последовательность действий или реализовать другой набор операций при использовании лишь 11 четырехбитовых сообщений (или команд, или инструкций), то нам придется только изменить порядок содержимого памяти, показанного в табл. 1.2. Поскольку мы не можем записать в память содержимое, выраженное словами русского языка, то должны транслировать сообщения с русского языка в четырехбитовые двоичные слова и затем записать их по правильным адресам в соответствии с желаемой последовательностью действий. К счастью, люди не столь механистичны, как машины. Мы в состоянии понимать русский язык с сокращениями или даже с ошибками, так что использованные нами длинные сообщения на русском языке могут быть значительно сокращены. Например, первое сообщение из табл. 1.2 можно заменить следующим сообщением:

LDI $\triangleq$ Загрузить регистр АК данными или содержимым, непосредственно следующим за текущей ячейкой памяти.

В двоичном коде сообщение может быть представлено кодом 0001. (Символ $\triangleq$ означает "определено как".) Теперь можно определить несколько терминов. Сокращенное обозначение известно как мнемоника, а комбинация 0001 или другое двоичное слово — как машинный код. **Мнемоники** предназначаются для людей, тогда как эквивалентные двоичные коды нужны для машин. Записанное в память содержимое в своей совокупности называется **программой**. Поскольку последовательность операций можно менять посредством записи содержимого в различные ячейки (или по разным адресам) без изменения электрической распайки машины, то этот материал называют **программным обеспечением** (или **программными средствами**)[1]. Содержимое памяти в зависимости от его природы называют **мнемоническими инструкциями** или **данными**. Например, LDI будет инструкцией, а "Джон К. Смит" — данными.

Поскольку контроллер разрабатывается и реализуется как составная часть ЦП и не предполагается, что он будет подвергаться изменениям со стороны пользователя, то распознаваемый машиной набор инструкций должен разрабатываться настолько гибким, насколько это возможно, чтобы машину можно было использовать для решения самых разнообразных задач. Реальные ЭВМ располагают намного большим числом инструкций, чем в нашем примере. Для ЭВМ PDP-11 существует около 70 инструкций общего назначения.

Вышеописанный основной процесс разработки программного обеспечения имеет следующие четыре шага:

1. Для конкретного приложения выбрать соответствующие мнемонические инструкции из набора мнемонических инструкций, обеспечиваемого данной машиной.
2. Выстроить выбранные инструкции в желаемом порядке или последовательности.
3. Транслировать мнемонические инструкции в соответствующие им двоичные коды, определенные производителем машины.
4. Записать двоично-кодированные инструкции в память.

---

[1] Используемый здесь английский термин software буквально означает "нечто гибкое". — *Прим. перев.*

Первые два шага известны как **программирование на языке ассемблера**. Третий шаг – это **ассемблирование**, а четвертый – **загрузка памяти**.

Поскольку мнемонические инструкции обычно очень примитивны и тесно взаимосвязаны с аппаратурой машины, программы на языке ассемблера, как правило, машинно-зависимы. Другая машина будет располагать другим набором мнемонических инструкций и, вообще говоря, другой аппаратной организацией. Поэтому хорошему программисту, прежде чем он сможет приступить к программированию на языке ассемблера, необходимо детально ознакомиться с аппаратной организацией машины. К счастью, человеку, овладевшему программированием на языке ассемблера для какой-либо достаточно мощной машины, перейти на другую машину не составит большого труда. В следующем разделе мы изучим в деталях вычислительную систему общего характера и познакомимся с ее структурой. Мы увидим, что программные и аппаратные компоненты вычислительной системы функционируют как одна "бригада".

1.2. БАЗОВАЯ СТРУКТУРА ЦИФРОВОЙ ВЫЧИСЛИТЕЛЬНОЙ СИСТЕМЫ

Описанная выше ЭВМ – просто игрушка. Она оказалась бы безполезной при любом практическом применении. На практике объем памяти никогда не ограничивается 16 словами. Благодаря постоянно снижающейся стоимости полупроводниковой памяти вычислительные системы сегодня могут обладать памятью по крайней мере в 16 000 слов (что выражают как 16 Кслов) размером 8 или 16 бит или больше вместо 4 бит в нашем примере. Для программ, состоящих из ста или более инструкций, процесс ручной трансляции мнемонических инструкций в двоичные коды для занесения в память оказывается неприемлемым. Чтобы быть полезной, машина должна иметь разумный объем памяти, быть способной принимать мнемонические инструкции непосредственно от пользователя и транслировать их в двоичный код автоматически.

На рис. 1.3 изображена системная организация одной популярной цифровой ЭВМ. Если исключить три блока, названные приводом диска, терминалом с электронно-лучевой трубкой и печатающим устройством, то эта система похожа на ту, которая показана на рис. 1.2. Три дополнительных блока, именуемые периферийными устройствами, добавлены для удобства пользователя. Обратите внимание, что система обладает тремя шинами, с помощью которых ЦП может считывать информацию от любого из показанных блоков или записывать ее в любой блок.

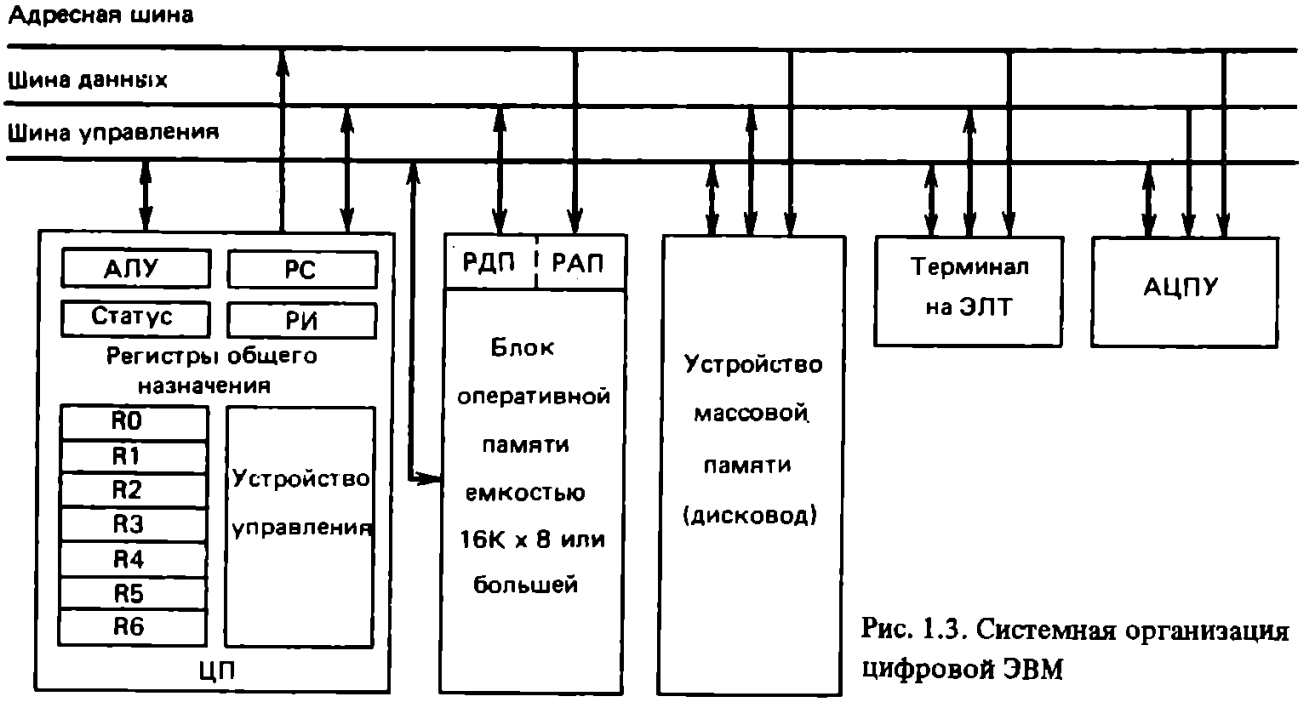


Рис. 1.3. Системная организация цифровой ЭВМ

Память. Обратите внимание, что с блоком памяти *функционально* сопряжены два регистра, помеченные РДП (регистр данных памяти) и РАП (регистр адреса памяти). Физически же эти два регистра обычно располагаются в ЦП. Когда ЦП хочет прочитать из памяти инструкцию или данные, он сначала устанавливает в РС желаемое значение адреса и посылает его в РАП, а затем к устройству памяти. После этого управляющий сигнал **чтение** от ЦП приводит к тому, что память помещает содержимое слова памяти, адресованного в РАП, в регистр РИ или в один из регистров общего назначения ЦП в зависимости от того, является это содержимое инструкцией или данными. Если вместо сигнала чтение используется сигнал запись, то содержимое некоторого регистра ЦП будет переслано в ячейку памяти, указанную в регистре РС, соответственно через РДП и РАП. Связь между ЦП и периферийными устройствами осуществляется аналогично. Каждое периферийное устройство в общем случае подключается к шинам через интерфейсную плату, на которой есть несколько специфических регистров. С этими регистрами ЦП взаимодействует аналогично тому, как он связывается с регистрами РАП и РДП и памятью.

Центральный процессор. Как показано на рис. 1.3, ЦП содержит арифметико-логическое устройство, группу регистров и устройство управления. Из названия понятно, что устройство АЛУ ответственно за все арифметические и логические операции. Регистры общего назначения R0, R1, . . . , R6 используются как временная память для данных АЛУ. Регистр РС служит программным счетчиком, который всегда указывает на тот адрес памяти, содержимое которого будет прочитано или записано в следующий раз. Регистр инструкции РИ удерживает извлеченную из памяти инструкцию, чтобы устройство управления могло декодировать ее и сгенерировать соответствующие сигналы для ЦП. Регистр состояния – это индикатор возникающих в ЦП условий или результатов действий АЛУ.

Например, один из двоичных бит этого регистра может быть определен как индикатор результата арифметической операции, выполненной АЛУ. Если результат нулевой, то индикаторный бит будет равен 1, иначе он будет равен 0. Такой индикатор называют *флагом*. В регистре состояния есть много двоичных бит, и все они могут использоваться для отображения текущего состояния ЦП, что позволяет реализовать на ЭВМ процессы с принятием решений. Весь процесс подобен кукольному театру. Программист, который разрабатывает содержимое памяти, – это режиссер кукольного представления, устройство управления функционирует как протянутые к кукле нити, а регистры выступают в качестве кукол. Представление идет в соответствии с программой или инструкциями, записанными в памяти или в мозгу режиссера.

Шинная структура. Шина в вычислительной системе – это среда, через которую системные компоненты связываются друг с другом. Это информационное "шоссе", образованное пучком проводов. Большинство ЭВМ имеют шинную структуру, как показано на рис. 1.3. Шина здесь функционально делится на три группы: адресную шину, шину данных и шину управления. Адресная шина переносит информацию о том, где искать инструкции или данные; шина данных переносит эти данные или инструкции для ЦП; шина управления обеспечивает информацию о том, как и когда должна выполняться операция.

Периферийные устройства. Наиболее часто в вычислительных системах применяются следующие периферийные устройства: 1) **устройства массовой памяти**, такие как приводы магнитных дисков и лентопротяжные механизмы для магнитной ленты; 2) **терминал с электронно-лучевой трубкой**, содержащий подобную телетайпу клавиатуру для ввода информации и экран на телевизионной трубке для вывода информации; 3) **печатающее устройство**, которое производит пригодные для чтения копии документов, чтобы пользователи могли проверять свою работу и сохранять результаты

для использования в будущем. Как следует из названия, устройство массовой памяти может применяться для хранения миллионов байт (8-битовых) или слов (16-битовых) информации временно или постоянно. Однако из-за более медленного доступа по сравнению с полупроводниковой памятью такое устройство используется, как правило, в качестве вторичного банка памяти. Когда возникает необходимость, информация сначала пересылается в оперативную память и после этого ЦП работает с оперативной памятью, а не с диском или лентой. Оперативная память действует как информационный резервуар, в котором ЦП может читать и записывать информацию с высокой скоростью в соответствии с желаниями пользователя. Терминал с экраном на электронно-лучевой трубке обычно используется как порт для ввода данных и команд пользователя. С помощью клавиатуры пользователь может выдавать команды или вводить данные; экран показывает ответную информацию.

ПРОГРАММНЫЕ ЭЛЕМЕНТЫ СИСТЕМЫ

Цифровая ЭВМ без программного обеспечения никогда не сможет работать, независимо от того, насколько сложной является ее аппаратура. Это напоминает человека, у которого есть дорогой автомобиль, но нет бензина, и поэтому он не может им воспользоваться. Давайте же рассмотрим программные элементы.

Машинные или мнемонические инструкции. Каждая машина, как правило, обладает уникальным набором инструкций, определенным ее производителем. Программист использует такой набор инструкций для написания программ, предписывающих машине выполнять ту задачу, которую определяет программист. Вообще говоря, инструкция сообщает машине, когда и где взять данные, что сделать с ними и куда поместить результат операции. Детальный формат инструкций варьируется от машины к машине. Но в общем случае в инструкции есть два существенных сегмента: один содержит операционный код, другой – адресную информацию об операнде или операндах, вовлеченных в операцию. Под операндом мы понимаем некоторый "предмет", над которым ЦП выполняет операцию.

Есть четыре общих типа формата инструкции: трехадресная, двухадресная, одноадресная и безадресная. Поясним эти понятия на примерах.

Пример 1. Трехадресная инструкция

```
ADD  A B C
```

предписывает ЦП сложить содержимое A с содержимым ячейки B и занести результат в ячейку C. Инструкция имеет четыре секционные зоны – **поля инструкции.** Зона для оператора ADD определяется как поле операционного кода; зоны для символов A и B определяются как поля исходных операндов; зона для символа C – как поле операнда назначения. Хотя обычно поля операндов содержат адресную информацию, иногда в них помещаются данные или константы. Для данного примера формальный формат инструкции будет

```
ADD  A, B, C
```

Поле операции отделяется от полей операндов пробелом, адресные поля разделяются запятыми.

Пример 2. Двухадресная инструкция

```
ADD  A, B
```

осуществляет следующее действие:

$$(A) + (B) \rightarrow B$$

где (A) – содержимое адреса A;
(B) – содержимое адреса B.

Эта инструкция предписывает ЦП прибавить содержимое ячейки А к содержимому ячейки В и занести результат в слово памяти с адресом В. Скобки символизируют "содержимое . . . ".

Обратите внимание, что если нужно поместить результат в ячейку памяти С, то необходима другая инструкция. Следующие две двухадресные инструкции выполняют ту же работу, что и одна трехадресная инструкция. Вторая инструкция здесь копирует (пересылает) содержимое ячейки В в ячейку С.

Инструкция	**Действие**
ADD A, B	(A) + (B) → B
MOVE B, C	(B) → C

Пример 3. Одноадресная инструкция

ADD A

осуществляет следующее действие:

(A) + (АК) → АК

Такая инструкция предписывает ЦП прибавить содержимое ячейки А к содержимому регистра с меткой АК и поместить результат в регистр АК. Может возникнуть вопрос, где же находится регистр АК. Вспоминаем, что в ЦП, изображенном на рис. 1.2, имеется регистр с меткой АК, т. е. показанная на этом рисунке машина является одноадресной, в ее ЦП есть регистр, предназначенный для использования в качестве аккумулятора. Чтобы выполнить такую же операцию, как в примере 1, этой машине потребуются следующие инструкции:

Инструкция	**Действие**
LOAD A	(A) → АК
ADD B	(B) + (АК) → АК
STORE C	(АК) → C

Второй операнд, АК, здесь явным образом не показан. Поскольку это всегда один и тот же регистр (АК) то ЦП "понимает" это и нет нужды показывать его явно как часть инструкции.

Пример 4. Безадресная инструкция.

Вычислительную машину, которая использует безадресные инструкции, называют еще стековой машиной. В машинах такого типа обычно есть целый ряд аккумуляторных регистров, располагающихся в ЦП и организованных в виде стека. Иногда они называются стековой памятью. **Стековая память** разрабатывается таким образом, что доступ к ней всегда осуществляется по правилу "**последним вошел – первым вышел**". Набор данных может быть поэлементно протолкнут (или занесен) в стек, начиная с его дна, так что последний протолкнутый элемент данных становится верхним слоем стека. При извлечении данных доступ в первую очередь осуществляется к верхнему слою стека, т. е. верхний элемент выталкивается из стека первым. Таким образом, указывать адресную информацию нет необходимости. На рис. 1.4 показан числовой пример для выполнения операции из примера 1. Пусть для ячеек А, В и С

(A) = 2, (B) = 3, (C) – не имеет значения или любое число. Для выполнения операции потребуется следующий набор инструкций:

Инструкция	**Действие**
LOAD A	(A) → СТЕК 4
LOAD B	(B) → СТЕК 3
ADD	(СТЕК 3) + (СТЕК 4) → СТЕК 4
STORE C	(СТЕК 4) → C

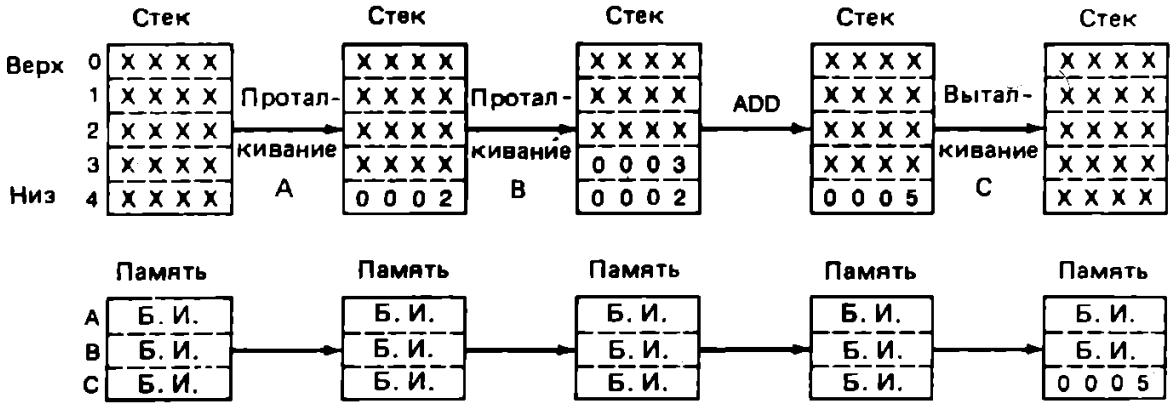


Рис. 1.4. Выполнение безадресной инструкции

На рис. 1.4 показана серия "моментальных снимков" (или "картинок движения") вместе с соответствующими инструкциями. Крестики на рисунке означают "не имеет значения" или "может быть любым числом"; Б. И. означает "без изменения". Обратите внимание, что ключевая инструкция, инструкция сложения ADD, не имеет адреса, поэтому такая машина называется безадресной.

Из приведенных выше примеров видно, что для выполнения одной и той же заданной операции на машине, имеющей инструкции с меньшим числом адресуемых операндов, требуется большее число инструкций. Однако когда инструкция содержит большое число адресуемых операндов, для ее кодирования требуется слово памяти большей длины или большее число бит в слове. И поскольку при большем числе инструкций для решения задачи требуется больше времени, то вся проблема сводится к поиску компромисса между скоростью, с которой машина выполняет операции, и объемом или длиной слова памяти.

1.3. СИСТЕМНОЕ ПРОГРАММНОЕ ОБЕСПЕЧЕНИЕ И ЯЗЫКИ ДЛЯ ЭВМ

Ранее мы указывали, что для людей необходимы мнемонические инструкции, похожие на сокращения в английском языке, а для машины нужен кодированный язык. Конечно, теоретически человек способен выучить двоичный машинный язык и выдавать машине команды или инструкции на этом языке, но на практике вряд ли разумно требовать, чтобы каждый пользователь осваивал такой примитивный язык по очевидным соображениям. В английском языке – 26 букв и 10 цифр. В двоичном машинном языке только две цифры, а именно: 1 и 0. Хотя двоичный язык прост, для человека он утомителен и сопряжен с ошибками. Идеальным было бы иметь такого специалиста, который способен транслировать мнемоники в двоичный язык для машины. К счастью, между мнемоническими инструкциями и двоично-кодированными битовыми комбинациями имеется взаимно однозначное соответствие, поэтому процесс трансляции представляется весьма механическим. А значит, нет нужды, чтобы трансляцию выполнял специалист. Необходимо только записывать последовательности инструкций на очень примитивном уровне, в мнемоническом языке, и располагать специальной программой, функционирующей как транслятор. Если на машине установлено программное обеспечение для выполнения трансляции, то пользователь может взаимодействовать с ней на языке, похожем на английский. Поэтому, чтобы система была полезной, в ней надо иметь программные средства для обеспечения разнообразного сервиса, и чем сильнее развито такое программное обеспечение, тем более мощной и полезной будет система.

Программное обеспечение бывает двух типов: встроенное и прикладное. Первое разрабатывается с целью создать удобства для пользователя, второе разрабатывается с помощью встроенного программного обеспечения для выполнения тех специфических

задач, которые пользователь хочет решать с помощью ЭВМ. Программа для управления главным входом в здание, рассмотренная выше, относится к прикладному программному обеспечению, тогда как транслятор мнемонических инструкций в двоичный язык — это пример программного обеспечения другого типа. Встроенное программное обеспечение известно как **системное программное обеспечение**. Оно разрабатывается и поставляется производителем ЭВМ и представляет собой совокупность программных модулей, каждый из которых подобно штатному сотруднику фирмы или корпорации выполняет специфические функции по обслуживанию пользователя системы.

АППАРАТНО-ПРОГРАММНАЯ "БРИГАДА"

Давайте познакомимся теперь с несколькими системами, относящимися к разным уровням развития ЭВМ. На рис. 1.5 показана аппаратная организация трех типичных вычислительных систем. На рис. 1.5, *а* представлена наиболее примитивная из них.

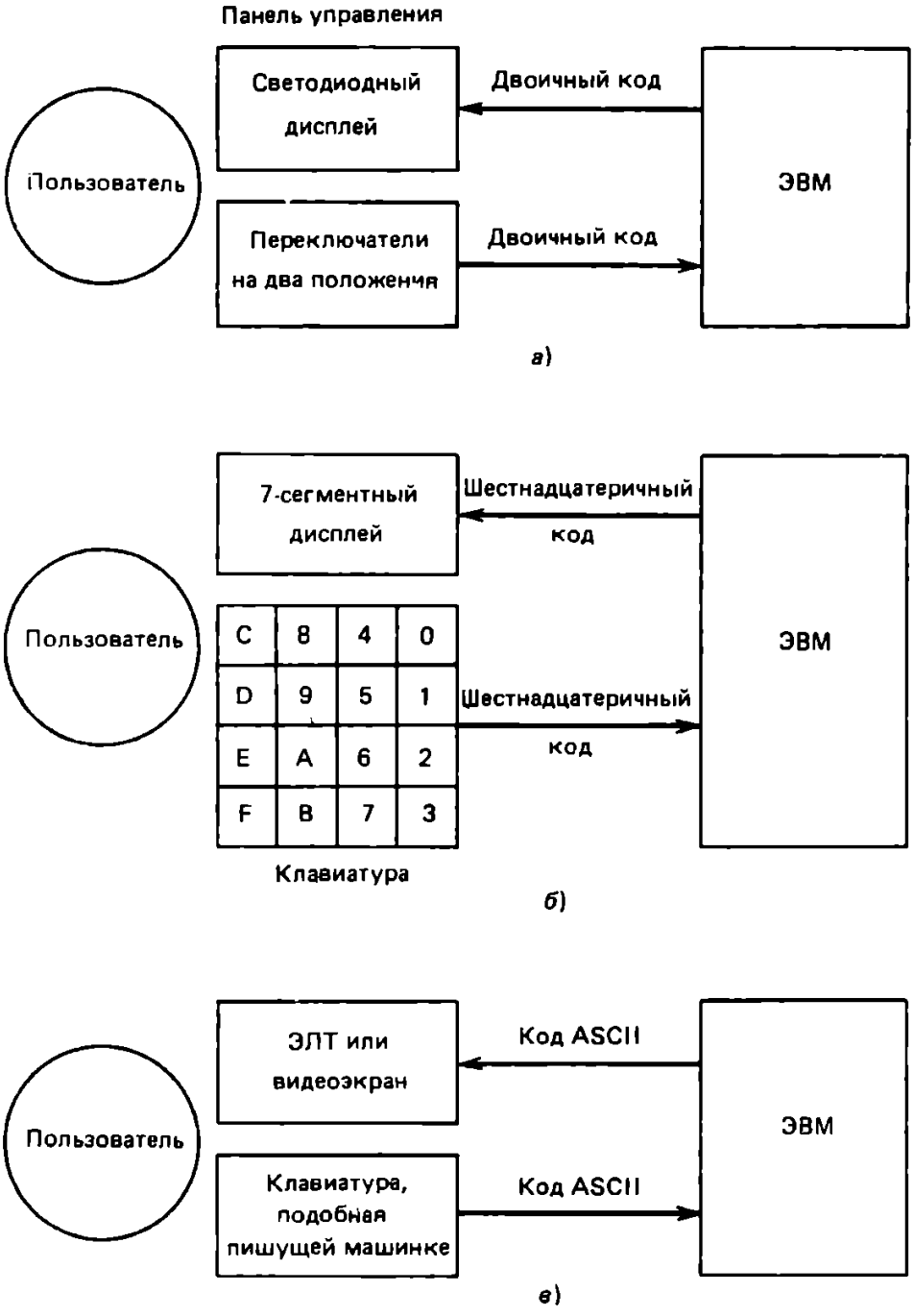


Рис. 1.5. Три типа вычислительных систем

Поскольку здесь требуется, чтобы пользователь взаимодействовал с ЭВМ посредством двоичного машинного языка, потребность в системном программном обеспечении отсутствует. Такая система недорога, но работать с ней трудно. На рис. 1.5, *б* показана более сложная система, содержащая шестнадцатеричную клавиатуру. Всякий раз, когда пользователь нажимает на какую-нибудь клавишу, генерируется и посылается в машину 4-битовый двоичный код. Двоичный код, соответствующий каждой клавише шестнадцатеричной клавиатуры, приведен в табл. 1.3. Сравнивая табл. 1.3 и 1.2, можно увидеть, что шестнадцатеричная клавиатура может быть использована для генерации и ввода в ЭВМ адресного кода и инструкций (или сообщений), чтобы машина функционировала в качестве контроллера главного входа в здание из приведенного выше примера. Если ЭВМ предназначается для такого применения, то системное программное обеспечение может и не потребоваться. Но если система должна быть более гибкой, то нужно хотя бы простое системное программное обеспечение для управления клавиатурой и дисплеем.

Таблица 1.3. **Шестнадцатерично-двоичный код**

Клавиша шестнадцатеричной клавиатуры	Двоичный код	Клавиша шестнадцатеричной клавиатуры	Двоичный код
0	0 0 0 0	8	1 0 0 0
1	0 0 0 1	9	1 0 0 1
2	0 0 1 0	A	1 0 1 0
3	0 0 1 1	B	1 0 1 1
4	0 1 0 0	C	1 1 0 1
5	0 1 0 1	D	1 1 0 1
6	0 1 1 0	E	1 1 1 0
7	0 1 1 1	F	1 1 1 1

На рис. 1.5, *в* пользователю предоставляется терминал, содержащий два независимых устройства: видеоэкран ЭЛТ и подобную пишущей машинке клавиатуру. Обратите внимание, что прямой аппаратной связи между ЭЛТ и клавиатурой нет. В противоположность обычной пишущей машинке то, что печатается на клавиатуре, на экране автоматически отображаться не будет (если только этого не обеспечит работающая ЭВМ с помощью специально написанной и установленной в машине системной программы, выдающей на экран эхо при нажатии клавиши). Так же как и шестнадцатеричная клавиатура, такая клавиатура генерирует уникальный двоичный код для каждой нажимаемой клавиши. Есть несколько стандартных кодов, которые могут использоваться для клавиатуры. Наиболее популярен **стандартный американский код для обмена информацией** ASCII. В табл. 1.4 приводятся символы кода ASCII в двоичном представлении для каждой клавиши стандартной клавиатуры. С помощью такой клавиатуры, дополненной соответствующей электронной интерфейсной платой, мы можем генерировать код ASCII любого знака и пересылать его в ЭВМ.

Таким образом, теоретически возможно с помощью клавиатуры вводить в ЭВМ сообщения или команды и взаимодействовать с ней на английском языке. К несчастью, способность распознавать каждый введенный знак не означает, что ЭВМ в состоянии понимать английские сообщения или команды. Ей пришлось бы "выучить" английскую грамматику, структуру предложений, а также правописание и значение каждого слова, прежде чем она смогла бы понимать наш язык. Поэтому нам приходится пользоваться тем, что доступно в настоящее время. Хотя язык с мнемоническими инструкциями уже на несколько порядков удобнее для нашего использования, все же он машинно-зависим и труден в изучении. Вследствие этого появился целый ряд других,

так называемых языков высокого уровня (более похожих на английский); на многих вычислительных системах доступны такие языки, как Бейсик, Паскаль, Фортран, Алгол и Кобол. Языки высокого уровня обычно машинно-независимы и их намного легче осваивать, но они не столь быстры и эффективны, как язык ассемблера.

Таблица 1.4. **Алфавитно-цифровой двоичный код ASCII**

Знак	Двоичное представление символов кода ASCII	Знак	Двоичное представление символов кода ASCII
0	X 0 1 1 0 0 0 0	I	X 1 0 0 1 0 0 1
1	X 0 1 1 0 0 0 1	J	X 1 0 0 1 0 1 0
2	X 0 1 1 0 0 1 0	K	X 1 0 0 1 0 1 1
3	X 0 1 1 0 0 1 1	L	X 1 0 0 1 1 0 0
4	X 0 1 1 0 1 0 0	M	X 1 0 0 1 1 0 1
5	X 0 1 1 0 1 0 1	N	X 1 0 0 1 1 1 0
6	X 0 1 1 0 1 1 0	O	X 1 0 0 1 1 1 1
7	X 0 1 1 0 1 1 1	P	X 1 0 1 0 0 0 0
8	X 0 1 1 1 0 0 0	Q	X 1 0 1 0 0 0 1
9	X 0 1 1 1 0 0 1	R	X 1 0 1 0 0 1 0
A	X 1 0 0 0 0 0 1	S	X 1 0 1 0 0 1 1
B	X 1 0 0 0 0 1 0	T	X 1 0 1 0 1 0 0
C	X 1 0 0 0 0 1 1	U	X 1 0 1 0 1 0 1
D	X 1 0 0 0 1 0 0	V	X 1 0 1 0 1 1 0
E	X 1 0 0 0 1 0 1	W	X 1 0 1 0 1 1 1
F	X 1 0 0 0 1 1 0	X	X 1 0 1 1 0 0 0
G	X 1 0 0 0 1 1 1	Y	X 1 0 1 1 0 0 1
H	X 1 0 0 1 0 0 0	Z	X 1 0 1 1 0 1 0

Примечание. Буква X в двоичном виде кода ASCII – это двоичная переменная, которая служит битом четности и позволяет обнаруживать кодовые ошибки.

Для каждого языка в системе должна присутствовать системная программа, функционирующая как специальный транслятор для этого языка. Транслятор для языка ассемблера называется ассемблером, а транслятор для языка высокого уровня называют компилятором или интерпретатором в зависимости от его структуры или режима работы. Все системное программное обеспечение пишется обычно на языке ассемблера или на каком-либо промежуточном языке.

Теперь уже стало очевидным, что любая полезная или программируемая вычислительная система должна по меньшей мере включать терминал, системное программное обеспечение и аппаратуру ЭВМ. Располагая всем этим, пользователь может разрабатывать прикладные программы под конкретные требования. Функциональное соотношение между пользователем, системным программным обеспечением и аппаратурой ЭВМ показано на рис. 1.6. На этом рисунке выделены некоторые системные программы, функционирующие как стандартная программа ввода-вывода, компилятор, ассемблер и т. п. Фактически же системное программное обеспечение включает намного большее число программ, выполняющих функции самых разных типов.

ОПЕРАЦИОННАЯ СИСТЕМА

Совокупность системного программного обеспечения называется **операционной системой** ЭВМ. Далее для удобства операционную систему будем называть общим термином *системное программное обеспечение.* Операционная система ЭВМ напоминает административную систему ресторана. В ресторане разные сотрудники выполняют разные функции. Администратор отвечает за работу ресторана; официанты или официантки обслуживают клиентов. И хотя при поверхностном взгляде клиенты имеют дело

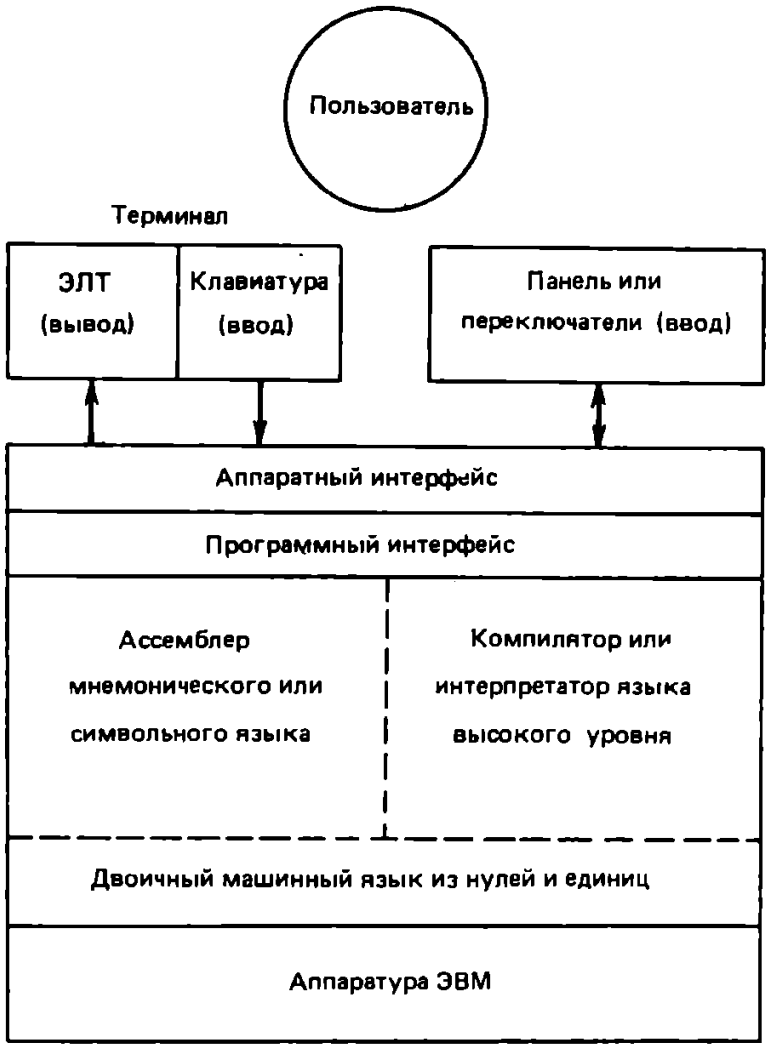


Рис. 1.6. Функциональное отношение между пользователем, системным программным обеспечением и аппаратурой

только с официантами или официантками, в действительности качество обслуживания зависит от административной системы. Аналогично, несмотря на то, что пользователь имеет дело только с переключателями на панели ЭВМ и с терминалами, ключевым фактором, обеспечивающим производительность всей системы, является операционная система, которая выполняет роль "администратора" ЭВМ.

Если рассматривать пользователя как некоего ремесленника, стремящегося создать произведение искусства (прикладную программу), то операционная система представляется своеобразным ящиком, в котором разнообразные инструменты ожидают, когда их используют по назначению. Следовательно, необходимо, чтобы пользователь понимал важность операционной системы и знал, как ее применять. Операционная система обычно содержит системные программы для управления файлами, выполнения отладки и другие служебные программы (утилиты). Как правило, они хранятся на устройстве массовой памяти, подобном диску, и называются **дисковой операционной системой** (ДОС). Хотя в настоящее время существует много разнообразных операционных систем, доступных на различных вычислительных системах, и они во многом отличаются друг от друга, если пользователь хорошо изучит одну из них, то будет в состоянии легко переключаться с одной на другую. В конце книги мы познакомим читателя с одной из широко распространенных и мощных операционных систем для семейства ЭВМ PDP-11 – системой RSX-11M.

1.4. ВЗАИМОДЕЙСТВИЕ ЧЕЛОВЕКА С ЭВМ

До сих пор мы пытались дать представление о базовой цифровой ЭВМ – от примитивной ЭВМ специального назначения до более развитой универсальной машины. Мы показали, каким образом аппаратура и системное программное обеспечение работают совместно как одна "бригада", специально указав на различие между системным и прикладным программным обеспечением и подчеркнув ту важную роль, которую играет операционная система. В последующих главах мы будем изучать конкретную систему, а именно вычислительную систему PDP-11. Но прежде чем завершить эту главу, нам необходимо узнать, как пользователь взаимодействует с вычислительной системой с помощью терминала.

Рис. 1.7. Взаимодействие пользователя и ЭВМ

На рис. 1.7, *а* показаны функциональные связи между пользователем и вычислительной системой. Пользователь взаимодействует с ЭВМ с помощью терминала, у которого есть клавиатура, функционирующая как устройство ввода, и видеоэкран на ЭЛТ, работающий как устройство вывода. С терминала пользователь выдает команды системному программному обеспечению на выполнение обслуживания и разрабатывает прикладные программы путем ввода последовательностей инструкций и соответствующих данных или чисел. На экране появляются ответы системных программ на команды или эхо инструкций и данных, вводимых с клавиатуры. На этом рисунке отражены две важные концепции:

1. Команды, не относящиеся к прикладной программе, поступают прямо к системному программному обеспечению, так что пользователь имеет возможность взаимодействовать с операционной системой или требовать от нее обслуживания. Эту концепцию можно пояснить также с помощью аналогии с рестораном. Предположим, что клиенту нужно заказать обед из десяти блюд для группы друзей. Сначала он "отдает команду" официантке принести копию меню. Затем с помощью меню он может проконсультироваться с официанткой, составить "программу" обеда из десяти блюд и заказать его. Если клиент является пользователем или программистом ЭВМ, то он выдает "команды" системному программному обеспечению (официантке) об оказании помощи при разработке прикладной программы (обеда из десяти блюд), которая должна быть выполнена (обед приготовлен) другой системной программой (шеф-поваром). Вспоминаем, что операционная система представляет собой совокупность программных модулей, подобную совокупность людей, таких как официантки, шеф-повар и администратор. Ясно, что "команды" не предназначаются для прикладной программы, а сама прикладная программа содержит только последовательность инструкций и данные или числа, но не команды. К несчастью, новички часто путают команды системной программы с инструкциями прикладной программы.

2. Выходная информация с клавиатуры – это строка двоичных кодов ASCII. Всякий раз, когда нажимается какая-нибудь клавиша, клавиатура генерирует уникальный

8-битовый (включая бит четности) код ASCII, соответствующий этой клавише (по табл. 1.4). Например, если с помощью клавиатуры вводится инструкция ADD # 2, A, то системное программное обеспечение "увидит" такую последовательность символов в коде ASCII:

Битовая позиция символов в коде ASCII

Символы	7 6 5 4 3 2 1 0	Ожидаемое действие
A	0 1 0 0 0 0 0 1	2 + (A) →A
D	0 1 0 0 0 1 0 0	
D	0 1 0 0 0 1 0 0	
пробел	1 0 1 0 0 0 0 0	
#	1 0 1 0 0 0 1 1	
2	1 0 1 1 0 0 1 0	
,	1 0 1 0 1 1 0 0	
A	0 1 0 0 0 0 0 1	

При выполнении инструкция предпишет ЦП прибавить число (#) два (2) к содержимому ячейки памяти с адресом A и поместить результат в ячейку A. Обратите внимание, что числовой знак 2 представлен своим кодом ASCII, который не является реальным двоичным числом. Реальное двоичное число 2 есть 00000010. Следовательно, системное программное обеспечение должно преобразовывать вводимые числа в коде ASCII в соответствующие им "истинные" двоичные числа, чтобы ЦП мог с ними работать. Все ЭВМ, большие и малые, разрабатываются для работы только с истинными двоичными числами, но не с кодами ASCII. Аналогично если пользователь хочет видеть на экране ЭЛТ полученный в ячейке A результат, то системное программное обеспечение должно преобразовать его обратно в код ASCII и отобразить на экране.

Внимательное рассмотрение изображенного на рис. 1.7, *а* процесса взаимодействия человека с ЭВМ выявляет, что от системного программного обеспечения следует ожидать выполнения намного большего числа услуг, нежели преобразование кода ASCII в двоичные числа и обратно. Например, системные программы могут распознавать, является ли вводимая информация командой, инструкцией или числом в коде ASCII. Кроме того, на каком-то этапе разработки программы пользователю может просто потребоваться увидеть на экране то, что было напечатано на клавиатуре. В этом случае все, что необходимо сделать системной программе, — это обеспечить на экране эхо от клавиатуры. Если на клавиатуре нажимается клавиша A, то системная программа, выполняющая вывод, должна послать на экран код ASCII буквы A, чтобы для введенного знака A на экране было эхо.

Поскольку мы ожидаем от вычислительной системы выполнения многих функций, то очевидно, что для обеспечения взаимодействия с ЭВМ простой стандартной клавиатуры телетайпа недостаточно. Именно поэтому клавиатура терминала содержит намного больше клавиш, чем обычный телетайп. Новичков такая разнообразная клавиатура нередко приводит в замешательство. Фактически же (независимо от сложности) клавиатура может быть разбита на три группы клавиш: алфавитно-цифровые знаки и символы, клавиши управления форматом и устройством и специальные функциональные клавиши. Клавиши знаков и символов — это обычные клавиши телетайпа. Клавиши управления форматом, такие как перевод строки (LF), подача бланка (FF), звонок (BEL), табуляция (TAB) и возврат каретки (CR), используются для формирования вывода на экране ЭЛТ или распечатки на АЦПУ. Функции специальных функциональных клавиш определяются системным программным обеспечением. Поскольку каждой клавише соответствует уникальный код ASCII, то системный программный модуль ввода легко может идентифицировать ее и осуществить "переход" к определенной

подпрограмме для выполнения специфической функции. Например, для активизации какой-либо специфической функции может быть нажата клавиша с меткой управление (CONTROL) одновременно с другой, алфавитно-цифровой клавишей, например Z. К сожалению, такие двухклавишные группы в большинстве своем не стандартизованы, так что их определения от системы к системе варьируются. Поэтому пользователю приходится знакомиться с руководствами, поставляемыми производителем ЭВМ, и заучивать конкретную функцию каждой клавиши.

Обратите внимание, что большинство кодов управления форматом и функциональных кодов являются непечатными; если желательно эхо при нажатии клавиш таких кодов, то требуется специальная подпрограмма. Например, если нажимаются одновременно клавиши управления и Z, то можно написать подпрограмму, которая отобразит на экране один символ ^ и один знак Z. Какой или какие символы при этом отображаются, не является существенным; важно только, чтобы пользователь знал, что происходит при нажатии функциональной клавиши.

В следующей главе содержится подробное описание конкретной вычислительной системы – ЭВМ PDP-11. Вам необходимо изучить принципы работы, аппаратную архитектуру и операционную систему ЭВМ PDP-11 и, кроме того, отправиться в лабораторию для разработки прикладных программ, а также их отладки и выполнения. Экспериментирование в лаборатории напоминает поход в бассейн для того, чтобы научиться плавать. Невозможно научиться плавать без реального плавания в бассейне, реке или озере. Пользователь должен детально изучить работу на вычислительной системе, но, ограничиваясь только этим, он никогда не станет хорошим программистом.

1.5. УПРАЖНЕНИЯ

1. Разработайте ЭВМ специального назначения, аналогичную контроллеру защиты входа в здание, описанному в этой главе, которая должна функционировать как автоматический банковский кассир. Вы можете располагать 16 или большим числом ячеек памяти, и каждое слово памяти может быть четырехбитовой или большей длины. Наилучшей будет, конечно, разработка, требующая меньшего числа слов памяти и более сжатого по ширине слова памяти. Представьте свою разработку в формате, подобном табл. 1.2, но как с двоичными, так и с шестнадцатеричными кодами и комментариями на русском языке. Вы можете исходить из того, что ваша система располагает терминалом с ЭЛТ и подобной телетайпу клавиатурой и что терминал способен вводить-выводить сообщения на английском языке, если вы как разработчик предварительно определите двоичный код для каждого сообщения. Желательны опять ясные, короткие и недвусмысленные сообщения. Ваша система может потребовать от клиента представить его (или ее) полное имя, пароль, девичью фамилию матери; дату рождения и т. п. Затем она проверит, хочет ли клиент снять деньги со счета или увеличить счет, и определит точную сумму. Наконец, она может ответить сообщением такого типа: "Операция произведена, благодарю за пользование нашим банком".

2. Опишите кратко собственными словами, что представляет собой операционная система и почему вычислительная система в ней нуждается.

3. Кратко обсудите различия по структуре и характеристикам между трехадресной, двухадресной, одноадресной и безадресной машинами.

4. Кратко опишите функции программного обеспечения ввода-вывода для интерфейса между терминалом с ЭЛТ и ЭВМ.

5. Кратко обсудите различия между командами и прикладным программным обеспечением.

ГЛАВА 2

СИСТЕМНАЯ ОРГАНИЗАЦИЯ ЭВМ PDP-11 С ТОЧКИ ЗРЕНИЯ ПОЛЬЗОВАТЕЛЯ

2.1. ФУНКЦИОНАЛЬНОЕ ОПИСАНИЕ СИСТЕМЫ

Структура вычислительной системы PDP-11 похожа на ту, что показана на рис. 1.3. С помощью операционной системы пользователь может разрабатывать прикладное

программное обеспечение на терминале с ЭЛТ. С клавиатуры терминала пользователь может вводить команды или запрашивать у конкретных системных программ помощь при разработке целевой программы, предназначенной для выполнения той задачи, которую пользователю хотелось бы решать на ЭВМ. Все системные программы первоначально располагаются на диске, за исключением терминального (или консольного) монитора (системный программный модуль), который обычно после активизации (или раскрутки) системы оказывается в оперативной памяти. Консольный монитор взаимодействует с пользователем, интерпретирует команды и передает запросы на выполнение операций другим системным программам. После завершения полученного от пользователя задания монитор возвращается в режим готовности выполнить любое новое пожелание пользователя. Пользователь является ведущим (главой), а монитор – администратором, распоряжающимся своей "бригадой" (системными программными модулями), каждый член которой обслуживает некоторые специфические функции. Чтобы поддерживать правильное функционирование вычислительной системы, ведущий (пользователь) должен прежде всего знать, как осуществляется взаимодействие с администратором (монитором), а также знать функции каждого модуля, работающего под наблюдением монитора. Затем уже следует основное: последовательность инструкций и данные прикладной программы. В противоположность программированию на языке высокого уровня при программировании на языке ассемблера требуется, чтобы программист знал аппаратную структуру, а также инструкции и представление данных для конкретной вычислительной системы.

2.2. ОРГАНИЗАЦИЯ СИСТЕМНЫХ АППАРАТНЫХ СРЕДСТВ

Типичная система ЭВМ PDP-11 состоит обычно из центрального процессора, оперативной памяти и периферийных устройств, таких как терминалы, дисковый привод и АЦПУ. Глобальные функции периферийных устройств мы обсудили в гл. 1. Что касается программирования на языке ассемблера, то все периферийные устройства можно рассматривать как некоторое множество регистров. Детали мы обсудим в гл. 7, а сейчас сосредоточим наше внимание на ЦП и оперативной памяти. На рис. 2.1 показана упрощенная системная диаграмма ЭВМ PDP-11, содержащая ЦП, шину и оперативную память.

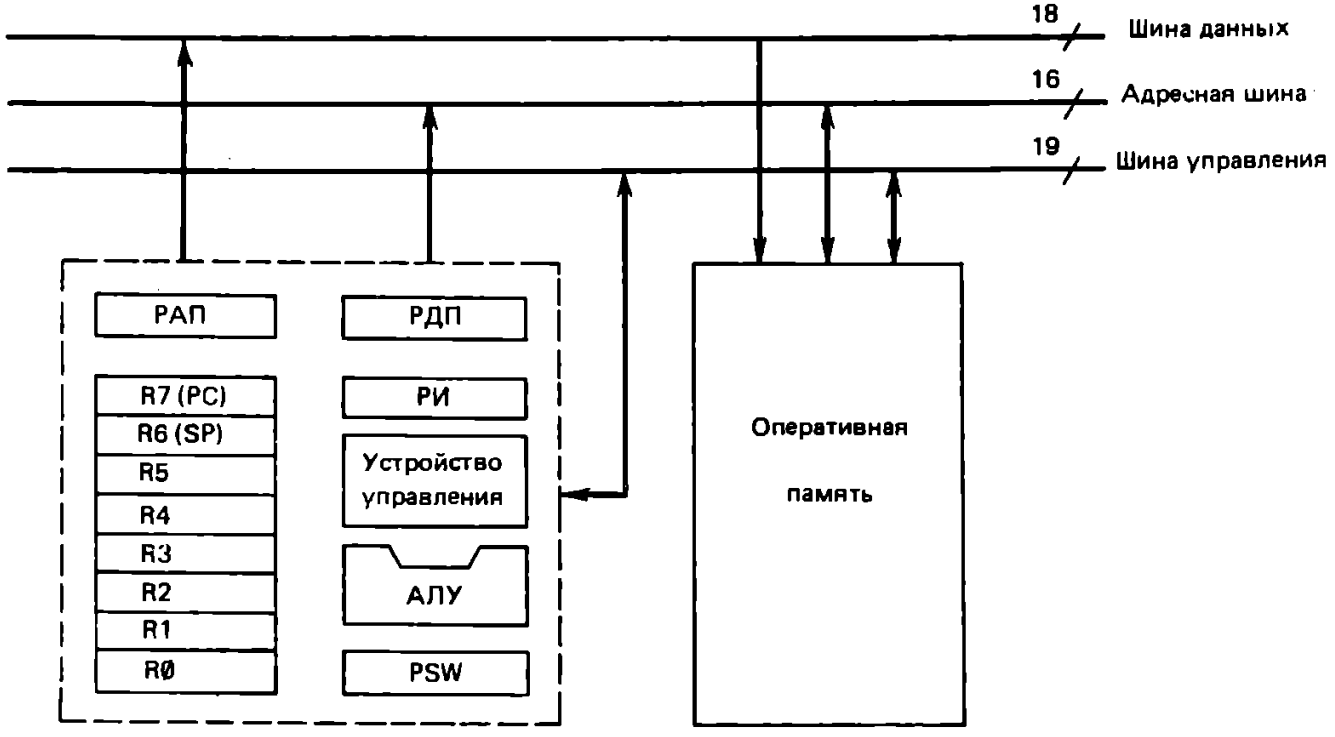


Рис. 2.1. Упрощенная системная диаграмма ЭВМ PDP-11

ЦЕНТРАЛЬНЫЙ ПРОЦЕССОР (ЦП)

Как показано на рис. 2.1, ЦП содержит РАП (регистр адреса памяти), РДП (регистр данных памяти), РИ (регистр инструкции), восемь регистров общего назначения (R0, R1, . . . , R7), PSW (слово состояния процессора), устройство управления и арифметико-логическое устройство (АЛУ). Функции всех этих элементов были обсуждены в предыдущей главе, за исключением PSW и восьми регистров общего назначения, роль которых будет выяснена в дальнейшем.

Поскольку ЭВМ PDP-11 – это 16-битовая машина, все регистры ЦП 16-битовой ширины, причем эти биты обычно определяются так, что справа располагается **самый младший бит (СМБ)**, а слева – **самый старший бит (ССБ)**:

$b_{15}, b_{14}, \ldots, b_1, b_0$,

где b_i – это двоичная переменная при $i = 0, 1, \ldots, 15$.

Считается, что каждый регистр состоит из **слова** памяти. Слово здесь обычно эквивалентно двум **байтам**, а каждый байт состоит из восьми бит. Это напоминает нашу монетную систему. Здесь нет никакой мистики, есть лишь забота об удобстве.

В регистре PSW четыре младших бита отражают значения **истина** (когда $b_i = 1$) или ложь (когда $b_i = 0$) текущего состояния АЛУ или результата, следующего сразу за операцией АЛУ. Они известны как **флаги** состояния АЛУ. Формат PSW определяется следующим образом:

Битовая позиция	Мнемонический символ	Описание
b_0	C	Перенос
b_1	V	Переполнение
b_2	Z	Нуль
b_3	N	Отрицательный результат
b_4	T	Захват
$b_5 - b_7$	–	Приоритет прерывания
$b_8 - b_{15}$	–	Не используются

Если, например, выполняется инструкция

SUB R0, R1

то АЛУ вычтет содержимое регистра R0 из содержимого регистра R1 и поместит результат в регистр R1. Если результат равен нулю, то бит b_2 в регистре PSW будет автоматически установлен в единицу устройством управления ЦП. Если результат отрицательный или меньше нуля, в единицу будет установлен бит b_3. Заметим, что PSW – очень важный регистр ЦП, поскольку он обеспечивает информацию для принятия решений в процессе выполнения программы, а без возможности принятия решений ЭВМ почти бесполезна.

Значение бит C, Z, N очевидно, но смысл бит V (переполнение), T (захват) и бит приоритета прерывания пока неясен. В дальнейшем студенты узнают, что если бит V имеет значение **истина**, то это показывает, что результат последней операции АЛУ неправильный и выполнение программы должно быть остановлено. К несчастью, переполнение часто путают с переносом, сопряженным с битом C, истинное значение которого подразумевает, что в результате последней операции был перенос. Результат тем не менее правильный, и выполнение останавливать не нужно. Смысл бита T и бит приоритета прерывания будет рассмотрен в гл. 8.

Теоретически восемь регистров общего назначения находятся в распоряжении прог-

раммиста, т. е. программист может использовать любой из этих регистров для любой цели. Однако регистры R7 и R6 обычно назначаются соответственно как программный счетчик (PC) и указатель стека (SP).

Как описано в предыдущей главе, регистр РАП сохраняет адрес инструкции, находящейся в стадии выполнения, а функция регистра PC – содержать адрес той инструкции, которая будет выполняться следующей. Функция регистра SP нами еще не исследовалась. Во многих случаях программисту желательно зарезервировать в оперативной памяти специальную область, называемую стековой памятью, для временного запоминания информации во время выполнения программы. Если так, то программист может использовать регистр SP в качестве адресного указателя (подобного регистру PC) на эту специальную область для временного хранения данных. Эта специальная область похожа на бумагу для черновиков, которую мы нередко используем, когда хотим, чтобы какой-то сложный процесс оставался систематическим и ясным. В главах, касающихся подпрограмм и прерываний, мы опишем использование регистра SP и области стековой памяти более детально.

Остальные шесть регистров общего назначения, R0, . . . , R5, действительно имеют общее назначение. Эти регистры могут использоваться программистом для любой цели. Важно отметить, что в ЦП помимо АЛУ и устройства управления имеется еще двенадцать регистров. Однако среди них только R0, . . . , R7 и PSW "видимы" или доступны программисту, т. е. программист может осуществлять доступ только к этим девяти регистрам.

ОПЕРАТИВНАЯ ПАМЯТЬ

Устройство оперативной памяти, показанное на рис. 2.1, является другим важным элементом вычислительной системы. В пятидесятых и начале шестидесятых годов устройства оперативной памяти изготовлялись из крошечных магнитных сердечников. Но с развитием полупроводниковой технологии почти экспоненциально снижалась стоимость полупроводниковой памяти, и в настоящее время оперативную память в большинстве случаев делают из полупроводниковых устройств. Как описано в предыдущей главе, оперативная память служит прежде всего как область для сохранения программ и данных, с которыми взаимодействует ЦП при разработке и выполнении программ.

Хотя в аппаратном смысле структура оперативной памяти весьма сложна, с точки зрения программиста использовать ее очень просто. Программисту необходимо иметь дело только с адресом и содержимым этого адреса. Как показано на рис. 2.2, концептуально структуру оперативной памяти можно представить как оперативную память с двумя подсоединенными регистрами, РАП и РДП, первый из которых предназначен для адресации, а второй – для адресованного содержимого. Содержимое может быть инструкцией или данными. Адрес – это обычно некоторое числовое значение (вспомним аналогию с почтовым ящиком). Однако конкретному адресному значению программист может присвоить некий символьный адрес, состоящий из одного или нескольких печатных знаков, и в дальнейшем ссылаться на этот адрес по этому символьному адресу. Например, если по улице Линкольна, дом 1000 расположена гостиница, то на нее можно ссылаться как на отель по имени "гостиница" или по номеру 1000 на улице Линкольна.

Рис. 2.2. Концептуальная структура оперативной памяти

Символьный адрес (необязательный)[1]	8-битовый числовой адрес								16-битовое содержимое													
									Старший байт (нечетный)								Младший байт (четный)					
	a_7	a_6	a_5	a_4	a_3	a_2	a_1	a_0	b_{15}	b_{14}	b_{13}	b_{12}	b_{11}	b_{10}	b_9	b_8	b_7	b_6	b_5	b_4	b_3	b_2	b_1	b_0
	0	0	0	0	0	0	0	0	0	0	1	0	1	1	0	0	0	1	0	0	1	0	1	1
	0	0	0	0	0	0	1	0	1	0	1	0	0	0	0	0	0	1	1	0	1	0	1	0
	0	0	0	0	0	1	0	0	0	1	0	1	0	1	0	0	0	1	1	1	1	0	0	0
	0	0	0	0	0	1	1	0	1	0	1	0	1	0	0	0	0	1	0	1	0	1	0	0
≃	—	—	—	—	—	—	—	—	—	—	—	—	—	—	—	—	—	—	—	—	—	—	—	—
	1	1	1	1	1	0	0	0	1	0	1	0	1	0	1	1	1	1	0	0	1	0	1	0
	1	1	1	1	1	0	1	0	0	1	1	1	0	0	0	0	0	0	1	1	0	1	0	1
	1	1	1	1	1	1	0	0	0	0	0	1	1	0	0	0	0	1	1	0	1	0	1	0
	1	1	1	1	1	1	1	0	0	1	0	1	0	1	0	0	1	0	1	0	1	1	0	0

[1] Определяется программистом

Рис. 2.3. Организация оперативной памяти ЭВМ PDP-11

Числовой адрес (восьмеричный)			Содержимое в восьмеричном представлении на байтовой основе					
			Старший байт			Младший байт		
a_7a_6	$a_5a_4a_3$	$a_2a_1a_0$	$b_{15}b_{14}$	$b_{13}b_{12}b_{11}$	$b_{10}b_9b_8$	b_7b_6	$b_5b_4b_3$	$b_2b_1b_0$
0	0	0	0	5	4	1	1	3
0	0	2	2	4	0	1	5	2
0	0	4	1	2	4	1	7	0
0	0	6	2	5	0	1	2	4
.	.	.	.	.	.	.	.	.
	.	.	.	.	.	.	.	.
.	.	.	.	.	.	.	.	.
3	7	0	2	5	3	3	1	2
3	7	2	1	6	0	0	6	5
3	7	4	0	3	0	1	5	2
3	7	6	1	2	4	2	5	4

			Содержимое в восьмеричном представлении на пословной основе					
			b_{15}	$b_{14}b_{13}b_{12}$	$b_{11}b_{10}b_9$	$b_8b_7b_6$	$b_5b_4b_3$	$b_2b_1b_0$
0	0	0	0	2	6	1	1	3
0	0	2	1	2	0	1	5	2
0	0	4	0	5	2	1	7	0
0	0	6	1	2	4	1	2	4
.	.	.	.	.	.	.	.	.
.	.	.	.	.	.	.	.	.
.	.	.	.	.	.	.	.	.
3	7	0	1	2	5	7	1	2
3	7	2	0	7	0	0	6	5
3	7	4	0	1	4	1	5	2
3	7	6	0	5	2	2	5	4

Рис. 2.4. Оперативная память ЭВМ PDP-11, показанная на байтовой и пословной основе

Оперативная память ЭВМ PDP-11 логически организована на базе байт и слов, как это показано на рис. 2.3. Обратите внимание, что показанная на рисунке структура памяти называется картой памяти и имеет 8-битовую длину адреса. Поскольку ЭВМ PDP-11 — машина с адресацией байт, основной единицей памяти является байт. Следовательно, показанная на рисунке память имеет объем 2^8, равный 256 слов или 512 байт. Для удобства содержимое памяти заполнено произвольными двоичными числами. Если теперь кому-то захочется прочитать (READ) слово в ячейке 100, то в ЦП будет счи-

тано содержимое 01 01 01 00 01 11 10 00. Если же захочется прочитать байт из ячейки 100, то вместо этого в ЦП будет считано содержимое младшего байта из ячейки 100: 01 11 10 00. Аналогично можно записать (WRITE) в байт желаемое содержимое, например 11 11 11 11, в ячейку 101. В результате содержимое ячейки 101 (старший байт слова в ячейке 100) станет равным 11 11 11 11.

Хотя сама концепция представляется вполне понятной, программист быстро обнаружит, что оперировать с двоичным представлением данных – очень неудобный (или "нецивилизованный") способ работы. Мы снова сталкиваемся с проблемой, разделяющей человека и машину. В качестве компромисса программист может использовать вместо двоичного **шестнадцатеричное** или **восьмеричное** представление информации. Шестнадцатеричное – это числовое представление 4-битовых групп из двоичных чисел (см. табл. 1.3), а восьмеричное – числовое представление 3-битовых групп из двоичных чисел. Причина, по которой вместо десятичной системы выбирается восьмеричная или шестнадцатеричная, состоит в том, что обе они обеспечивают простую группировку двоичных чисел. В ЭВМ PDP-11 используется восьмеричная система представления. В качестве примера на рис. 2.4 дано восьмеричное представление памяти, показанной на рис. 2.3, сделанное на байтовой и пословной основе.

Поскольку на байтовой и пословной основе двоичные числа группируются по-разному, одни и те же двоичные данные на байтовой основе несколько отличаются от тех же самых данных на пословной основе. Начинающим следует тщательно разобраться в этом различии, поскольку в системе ЭВМ PDP-11 мы при написании программ нередко смешиваем байтовую и пословную адресацию.

Когда определена восьмеричная система представления, программист может вводить данные в восьмеричном представлении с помощью клавиатуры терминала и читать данные на экране ЭЛТ также в восьмеричном представлении. Преобразование восьмеричного представления в коде ASCII в двоичное представление при вводе и двоичного представления в восьмеричное представление в коде ASCII при выводе должно осуществлять системное программное обеспечение.

2.3. ПОРЯДОК РАБОТЫ ЭВМ

Предположим, что некая прикладная программа уже разработана и записана в оперативную память, показанную на рис. 2.1, начиная с ячейки 2000. Область памяти между ячейкой 000000 и стартовой ячейкой прикладной программы, например 2000, обычно резервируется для системного программного обеспечения. Точный стартовый адрес для выполнения прикладной программы от одной машины к другой может варьироваться. Для удобства мы произвольно выбрали значение 2000.

Порядок работы ЭВМ кратко можно описать следующим образом. Сначала пользователь может выдать операционной системе или системному программному обеспечению команду 2000G0. Работа начинается с того, что значение 2000 помещается в регистр PC и затем переносится в регистр РАП, а одновременно с этим содержимое регистра PC автоматически увеличивается на 2 (байта). Затем извлекается содержимое ячейки 2000 и через шину, регистр РДП оно в конце попадает в регистр РИ. Устройство управления интерпретирует содержимое регистра РИ и определяет, требуется ли еще информация. Если да, то оно помещает содержимое регистра PC в регистр РАП, увеличивает (PC) на 2 и извлекает содержимое ячейки памяти, на которую теперь указывает регистр РАП. Процесс такого извлечения повторяется до тех пор, пока в ЦП не окажется вся информация, необходимая для одной инструкции. Затем ЦП выполняет инструкцию и переходит к извлечению следующей подлежащей выполнению инструкции. Этот процесс извлечения с последующим выполнением повторяется до тех пор, пока процессор не "увидит" инструкцию останова HALT. Тогда он останавливает выполнение, а содержимое регистра PC будет указывать на ячейку, где расположено слово, следующее за тем, в котором находится инструкция HALT. Порядок работы поясним на примере.

ПОРЯДОК РАБОТЫ ПРОСТОЙ ПРОГРАММЫ

Предположим, что, начиная с ячейки 2000, в памяти записана простая программа, карта памяти которой выглядит так:

Символьный адрес, определенный пользователем	Числовой адрес	Инструкции	Комментарии
START	2000	MOV #2, R0	Примечание 1
	2002		
	2004	MOV #4, R1	Примечание 2
	2006		
	2010	ADD R1, R0	Примечание 3
	2012	HALT	Примечание 4
	2014	? ? ? ? ?	Примечание 5

Примечание 1. В дальнейшем студент узнает, что эта инструкция занимает два слова памяти, 2000 и 2002. В ячейке 2000 располагается инструкция пересылки MOV, а в ячейке 2002 – число 2. Эта инструкция заставляет ЦП скопировать число 2 в регистр R0.

Примечание 2. Аналогично предыдущей инструкции, за исключением того, что здесь инструкция MOV находится в ячейке 2004, а число 4 помещено в ячейке 2006. Это приводит к тому, что ЦП копирует число 4 в регистр R1.

Примечание 3. Эта инструкция занимает одно слово памяти. Она заставляет ЦП прибавить содержимое регистра R1 к содержимому регистра R0 и поместить сумму, которая должна быть равна 6, в регистр R0. Первоначальное содержимое регистра R0 теряется.

Примечание 4. Эта инструкция занимает одно слово памяти. Она вынуждает ЦП остановить выполнение.

Примечание 5. Содержимое этой ячейки не имеет значения.

Чтобы начать работу, пользователь печатает на клавиатуре команду 2000GO. ЭВМ отвечает следующим образом:

1. 2000 → PC.
2. (PC) → РАП; содержимое регистра PC, обозначенное как (PC), в данном случае 2000, помещается в регистр РАП.
3. (PC) + 2 → PC; содержимое регистра PC увеличивается на 2, т. е. оно становится равным 2002.
4. Инструкция пересылки MOV, расположенная в ячейке 2000, извлекается через регистр РДП и помещается в регистр РИ.
5. ЦП интерпретирует инструкцию и обнаруживает, что необходима еще информация, и тогда помещает 2002 в регистр РАП и вновь увеличивает (PC) на 2. В результате (PC) равно 2004.
6. ЦП выполняет инструкцию, помещая 2 в R0.
7. ЦП извлекает следующую инструкцию, помещая (PC) в регистр РАП, в данном случае 2004, и корректирует содержимое регистра PC до значения 2006.
8. В ЦП попадает инструкция MOV. ЦП производит еще одно извлечение. В результате выполнения инструкции в R1 оказывается число 4, а (PC) становится равным 2010.
9. ЦП извлекает следующую инструкцию, помещая 2010 в регистр РАП, увеличивает (PC) до 2012; инструкция сложения извлекается и помещается в регистр РИ.

10. ЦП выполняет инструкцию сложения и помещает сумму, число 6, в регистр R0.
11. ЦП помещает содержимое регистра PC, которое теперь 2012, в регистр РАП, увеличивает содержимое регистра PC до 2014 и извлекает из памяти и заносит в регистр РИ инструкцию останова.
12. ЦП интерпретирует и выполняет инструкцию останова HALT, и машина прекращает работу.

Содержимое регистра PC указывает на ячейку 2014, а машина останавливает работу из-за инструкции HALT. Поскольку мы никогда ничего в ячейку 2014 не заносили, она имеет неопределенное содержимое, которое можно считать не чем иным, как мусором.

2.4. УПРАЖНЕНИЯ

1. Кратко опишите своими словами функции АЛУ, устройства управления и оперативной памяти в вычислительной системе PDP-11.
2. Кратко опишите функцию каждого из следующих регистров центрального процессора ЭВМ PDP-11: РАП, РДП, R0, R1, R2, R3, R4, R5, R6, R7, РИ, PSW.
3. Функции регистров общего назначения R0, R1, . . . , R5 аналогичны функциям слов оперативной памяти; зачем они нужны нам в ЦП?
4. В чем различие применения символьного адреса и числового адреса?
5. Используя приложение А, оттранслируйте в двоичные числа и запишите их в память ЭВМ PDP-11 в правильном порядке, начиная с адреса 0, последовательность символов: HELLO, PDP-11;
   а) покажите содержимое памяти по отношению к ее собственным адресам в двоичном представлении;
   б) оттранслируйте это же содержимое памяти в восьмеричные числа на байтовой основе;
   в) оттранслируйте это же содержимое памяти в восьмеричные числа на пословной (16-битовой) основе.

ГЛАВА 3

ПРЕДСТАВЛЕНИЕ ИНФОРМАЦИИ

Мы показали, что записанное в память содержимое представляет собой информацию. Эта информация может быть последовательностью инструкций или совокупностью данных или чисел. Инструкции сообщают ЦП, что и с какими данными он должен делать. Прежде чем научиться программировать, необходимо узнать, как представляются инструкции, данные или числа и каким образом ЦП манипулирует числами. В следующих разделах мы познакомимся сначала с представлением чисел и с тем, как ЦП работает с отрицательными числами или как реализуются в ЦП сложение и вычитание. Поскольку программисты используют восьмеричные числа, наряду с манипуляцией двоичными числами мы изучим манипуляцию восьмеричными числами. Затем мы кратко познакомимся с некоторыми простыми, но существенными инструкциями (детально набор инструкций будет изучаться в последующих главах).

3.1. ПРЕДСТАВЛЕНИЕ ЧИСЕЛ БЕЗ ЗНАКА ПО РАЗЛИЧНЫМ ОСНОВАНИЯМ СИСТЕМЫ СЧИСЛЕНИЯ

В течение столетий люди использовали десятичную систему счисления. Например, под трехцифровым представлением десятичного числа, скажем 789_{10}, мы в действительности понимаем представление значения

$$V_{10} = 7 \times 10^2 + 8 \times 10^1 + 9 \times 10^0.$$

Аналогично двоичное представление числа 1011_2 имеет значение

$$V_2 = 1 \times 2^3 + 0 \times 2^2 + 1 \times 2^1 + 1 \times 2^0.$$

Восьмеричное число 761_8 имеет значение

$$V_8 = 7 \times 8^2 + 6 \times 8^1 + 1 \times 8^0,$$

а шестнадцатеричное число $A1C_{16}$ имеет значение

$$V_{16} = 10 \times 16^2 + 1 \times 16^1 + 12 \times 16^0.$$

Нижний индекс показывает **основание системы счисления** числа. В общем случае значение n-цифрового числа определяется следующим образом:

$$V_b = (x_{n-1}, x_{n-2}, \ldots, x_1, x_0)_b = x_{n-1} b^{n-1} + x_{n-2} b^{n-2} \ldots + x_1 b^1 + x_0 b^0, \qquad (3.1)$$

где b – основание числа, а x_i – i-тая цифра при $i = 0,1, \ldots, n-1$ и $0 \leqslant x_i \leqslant b-1$.

Однако вне вычислительной системы принято использовать десятичное представление, поэтому для обслуживания машинного ввода-вывода необходимо располагать стандартным способом кодирования десятичных чисел в двоичном формате. Такой код называется двоично-кодированным десятичным (BCD). Он определяется так:

Десятичное представление	Эквивалент в BCD
0	0 0 0 0
1	0 0 0 1
2	0 0 1 0
3	0 0 1 1
4	0 1 0 0
5	0 1 0 1
6	0 1 1 0
7	0 1 1 1
8	1 0 0 0
9	1 0 0 1

Важно не путать представление BCD с двоичным значением десятичного числа. Например, десятичное число 17_{10} обладает эквивалентным двоичным значением 10001_2, т.е.

$$17_{10} = 10001_2.$$

Однако представление BCD этого десятичного числа будет

0001 0111,

где 0111 – код для первой цифры – 7, а 0001 – код для следующей цифры – 1. Заметим, что значения кодов этих чисел не совпадают со значением всего числа. Обратите внимание, что в первом случае нужно пять бит, а во втором – восемь бит.

Поскольку в вычислительной системе PDP-11 стандартным является восьмеричное представление, индекс восьмеричного числа обычно опускают. Десятичное же число вместо индекса 10 всегда сопровождается точкой. Например:

Десятичное число: $17. = 1 \times 10 + 7 \times 10^0$;

Восьмеричное число: $17 = 1 \times 8 + 7 \times 8^0$.

В табл. 3.1 показаны числовые представления в различных системах счисления для значений от 0 до 15_{10}.

Таблица 3.1

Десятичное представление	Восьмеричное представление	Шестнадцатеричное представление	Двоичное представление	Представление BCD	
0 0	0 0	0	0 0 0 0	0 0 0 0	0 0 0 0
0 1	0 1	1	0 0 0 1	0 0 0 0	0 0 0 1

Окончание табл. 3.1

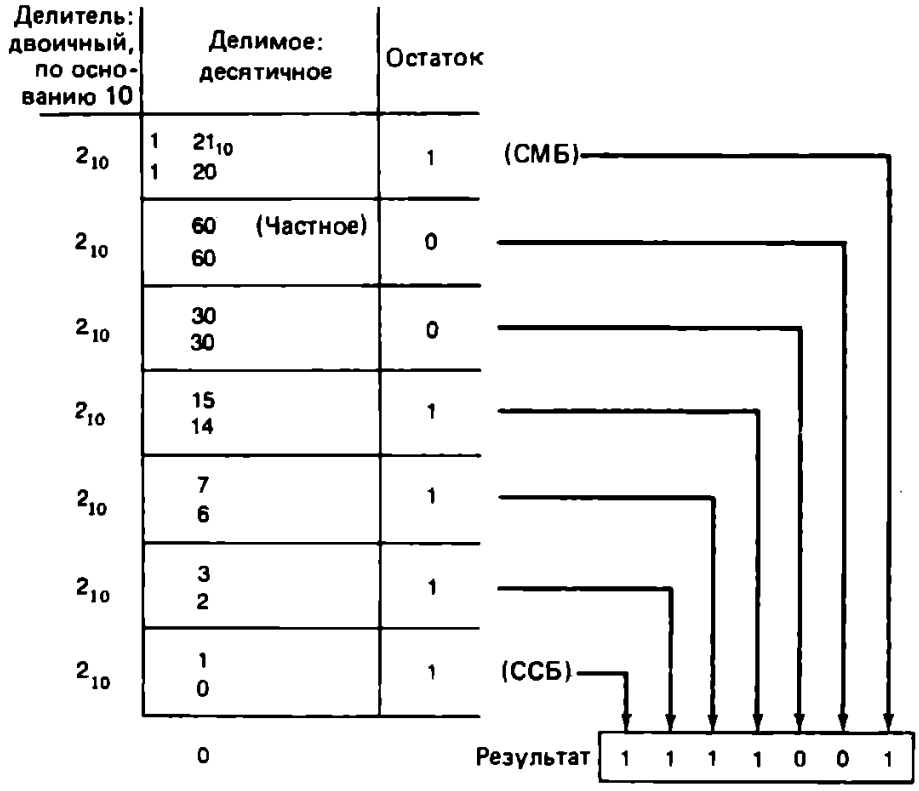
Десятичное представление	Восьмеричное представление	Шестнадцатеричное представление	Двойное представление	Представление BCD	
02	02	2	0010	0000	0010
03	03	3	0011	0000	0011
04	04	4	0100	0000	0100
05	05	5	0101	0000	0101
06	06	6	0110	0000	0110
07	07	7	0111	0000	0111
08	10	8	1000	0000	1000
09	11	9	1001	0000	1001
10	12	A	1010	0001	0000
11	13	B	1011	0001	0001
12	14	C	1100	0001	0010
13	15	D	1101	0001	0011
14	16	E	1110	0001	0100
15	17	F	1111	0001	0101

3.2. ПРЕОБРАЗОВАНИЕ ПРЕДСТАВЛЕНИЙ ЧИСЕЛ ПО РАЗЛИЧНЫМ ОСНОВАНИЯМ СИСТЕМЫ СЧИСЛЕНИЯ

Поскольку при программировании на языке ассемблера используются представления чисел по разным основаниям, то часто необходимо осуществлять преобразование от одной системы к другой. Процесс преобразования иллюстрируется приводимым ниже набором числовых примеров. В этих примерах для преобразования используется в основном способ деления вручную. Как видно из иллюстраций, мы для удобства изменили обычный формат деления вручную, переместив остаток вправо, а на то место, которое обычно отводится под остаток, поместили частное.

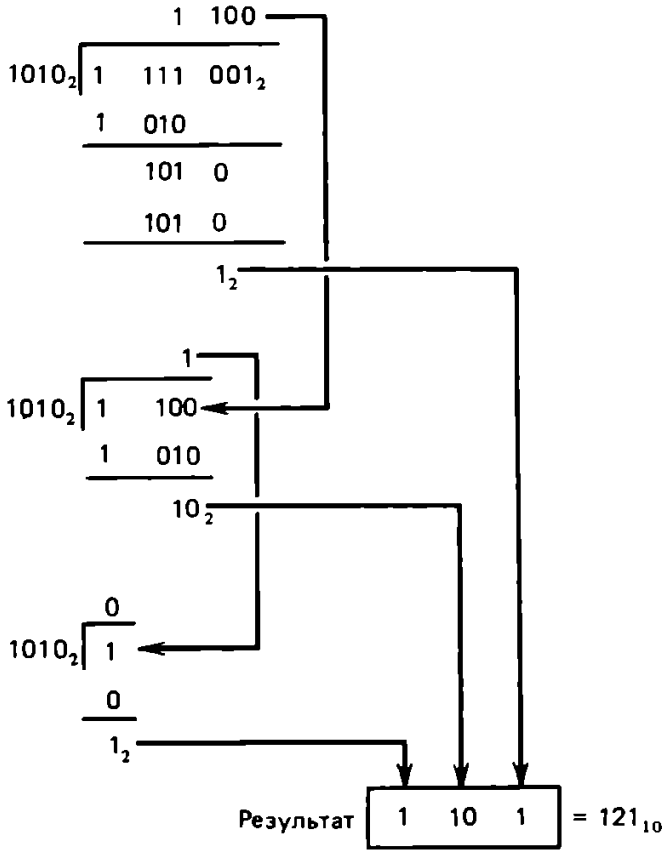
ПРЕОБРАЗОВАНИЕ ДЕСЯТИЧНЫХ ЧИСЕЛ В ДВОИЧНЫЕ ЧИСЛА

(На рисунке мы используем необычное размещение позиций для частного и остатка)



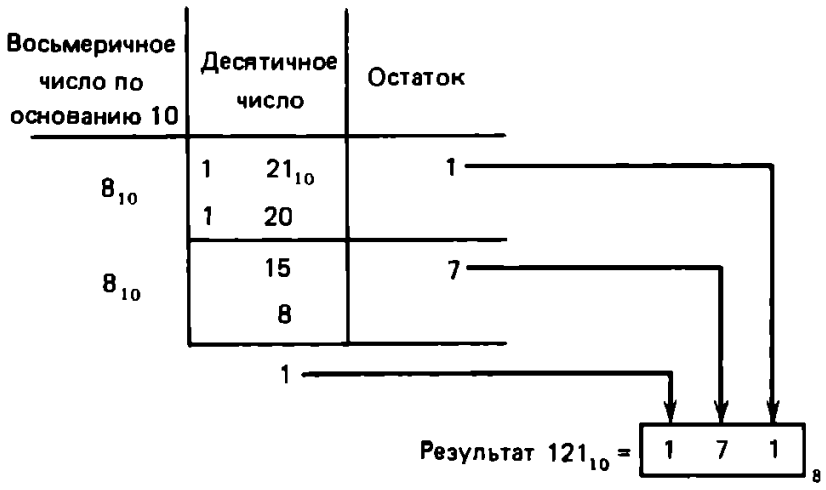
ПРЕОБРАЗОВАНИЕ ДВОИЧНЫХ ЧИСЕЛ В ДЕСЯТИЧНЫЕ ЧИСЛА

Поскольку двоичное представление основания 10 есть 1010_2, то мы разделим заданное число на 1010_2:

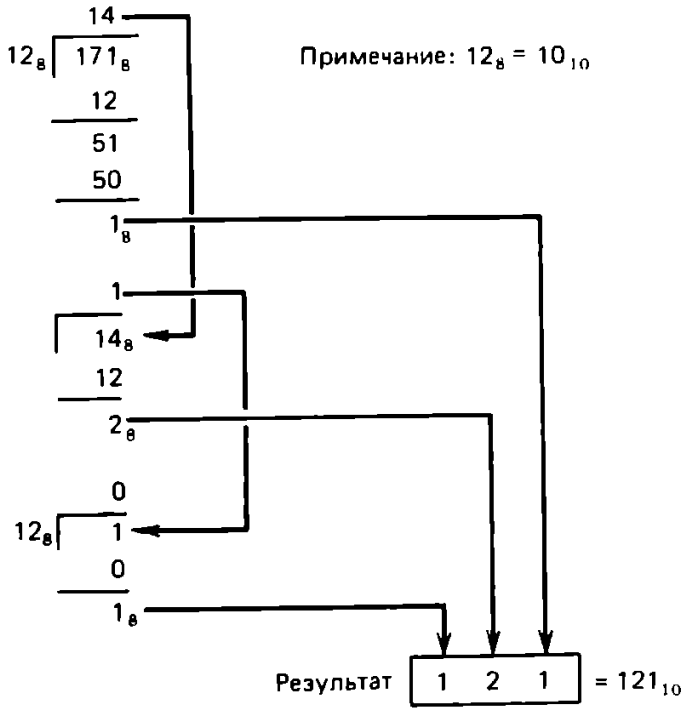


Обратите внимание, что в этом примере делитель, являющийся основанием 10, записан в двоичном виде потому, что делимое представлено в двоичном виде. Процесс вычитания в этом делении, выполняемом вручную, аналогичен такому же процессу в десятичной системе, за исключением того, что заем из левой цифры имеет значение 2, а не 10, как в десятичной системе.

ПРЕОБРАЗОВАНИЕ ДЕСЯТИЧНЫХ ЧИСЕЛ В ВОСЬМЕРИЧНЫЕ ЧИСЛА



ПРЕОБРАЗОВАНИЕ ВОСЬМЕРИЧНЫХ ЧИСЕЛ В ДЕСЯТИЧНЫЕ ЧИСЛА



ПРЕОБРАЗОВАНИЕ ВОСЬМЕРИЧНЫХ ЧИСЕЛ В ДВОИЧНЫЕ ЧИСЛА И ОБРАТНО

Хотя в данном случае применим и рассмотренный выше способ деления, это преобразование легче выполнить, если найти способ двусторонних преобразований. Из табл. 3.1 видно, что преобразование для этого примера возможно выполнить следующим образом:

$1\ 7\ 1_8 \quad 1\ 111\ 001_2$;

$1\ 111\ 001_2 \rightarrow 1\ 7\ 1_8$.

Теперь можно обобщить алгоритм деления:

$$\begin{aligned} X_{b1} &= (b2)_{b1}\, Q_0 + R_0; \\ Q_0 &= (b2)_{b1}\, Q_1 + R_1; \\ Q_i &= (b2)_{b1}\, Q_{i+1} + R_{i+1}; \\ Q_j &= (b2)_{b1}\, 0 + R_{j+1}, \end{aligned} \qquad (3.2)$$

где b_1 – основание системы счисления первоначального числа; b_2 – основание числовой системы, в которую производится преобразование; Q_0 – частное начального уровня деления; R_0 – остаток начального уровня деления; Q_i – частное i-го уровня деления; R_i – остаток i-го уровня деления.

Последовательно вычитая нижнюю строку из следующей верхней строки в (3.2), получаем

$$X_{b1} = R_{j+1}\, b_2^{j+1} + \ldots + R_1\, b_2^1 + R_0\, b_2^0.$$

Интересно отметить, что преобразование десятичного числа в двоичное легче выполнить с помощью преобразования десятичного числа в восьмеричное.

3.3. ПРЕДСТАВЛЕНИЯ ОТРИЦАТЕЛЬНЫХ ЧИСЕЛ

Есть два способа представления отрицательных чисел: с помощью **значения со знаком** и с помощью **дополнения**.

ПРЕДСТАВЛЕНИЕ С ПОМОЩЬЮ ЗНАЧЕНИЯ СО ЗНАКОМ

Для 16-битовой машины (цифровой ЭВМ) самый старший бит назначается знаковым битом (1 означает отрицательное значение, 0 – положительное), а остальные биты представляют значение числа. Например при 16-битовом двоичном числе $b_{15}, \ldots, b_1, b_0$ бит b_{15} будет знаковым битом, а биты $b_{14}, \ldots, b_0$ – значением. Значение числа можно оценить так:

значение равно $(1 - 2\,b_{15})\ (b_{14}\,2^{14} + \ldots + b_0\,2^0)$.

Например, $1\,000\,000\,000\,000\,110_2$ – отрицательное число со значением

$$V = (1 - 2 \times 1)\ (1 \times 2^2 + 1 \times 2^1 + 0 \times 2^0) = -6_{10}.$$

ПРЕДСТАВЛЕНИЕ С ПОМОЩЬЮ ДОПОЛНЕНИЯ

Представление с помощью дополнения может показаться несколько усложненным, но при его использовании для выполнения как сложения, так и вычитания нужен только один аппаратный сумматор. Более того, представление с помощью дополнения положительного числа есть то же самое, что его значение со знаком; поэтому в большинстве цифровых ЭВМ применяется представление с помощью дополнения. В настоящее время в цифровых ЭВМ используются два типа представления с помощью дополнения: дополнение до двух и дополнение до единицы. Но существуют также представления через дополнение до семи, восьми, девяти и десяти, доступные для использования программистами. Сначала мы дадим представление с помощью дополнения до двух, а затем с помощью других представлений.

Дополнение до двух

Математический вывод. Для n-битового числа, представленного с помощью значения со знаком, дополнение до двух может быть выведено по следующей формуле:

$$\boxed{b_{n-1}}\ b_{n-2}, \ldots, b_0 \xrightarrow{\bar{2}} -b_{n-1}\,2^{n-1} + b_{n-2}\,2^{n-2} + \ldots + b_0\,2^0. \qquad (3.3)$$

Здесь $\bar{2}$ над стрелкой символизирует операцию получения дополнения до двух; знаковый бит заключен в прямоугольник для облегчения его идентификации.

Пример 1. Нужно преобразовать число, значение со знаком которого равна –6, в его эквивалентное дополнение до двух.

Величина со знаком числа –6 в двоичном представлении = $\boxed{1}\ 110$

Здесь $n = 4$. Из уравнения (3.3) имеем

$$-1 \cdot 2^3 + 1 \cdot 2^2 + 1 \cdot 2^1 + 0 \cdot 2^0 =$$
$$= -8 + 4 + 2 = -2 \longrightarrow \boxed{1}\ 010$$

Таким образом, дополнение до двух числа, значение со знаком которого равна –6, в двоичном представлении будет

$$\boxed{1}\ 110 \xrightarrow{\bar{2}} \boxed{1}\ 010$$

Пример 2. Нужно преобразовать число, значение со знаком которого равна +6, в его эквивалентное дополнение до двух. Значение со знаком +6 равна 0110. Из уравнения (3.3) имеем

$$0 \cdot 2^3 + 1 \cdot 2^2 + 1 \cdot 2^1 + 0 \cdot 2^0 = +6 \longrightarrow \boxed{0}\ 110$$

Обратите внимание, что дополнение положительного числа в представлении через значение со знаком совпадает с самим собой. Это очевидно из уравнения (3.3), поскольку b_{n-1} (знаковый бит) равен нулю и, следовательно, не изменяет значения числа.

Из процесса получения дополнения становится ясно, что вычисления могут быть длинными. Тогда возникает законный вопрос, имеет ли смысл вообще избирать путь, связанный с дополнениями. К счастью, этот процесс можно легко реализовать логически следующим образом.

Получение дополнения с помощью логической процедуры

Шаг 1. Инвертировать значение числа, оставив знаковый бит нетронутым.

Шаг 2. Прибавить единицу к инвертированному числу.

Шаг 3. Добавить первоначальный знаковый бит к результату, полученному на шаге 2.

Пример. Преобразование числа –6 в его дополнение до двух с помощью логической процедуры (для удобства мы оставляем прямоугольник пустым до шага 3).

Шаг 1. $\boxed{1}\ 110 \xrightarrow[\text{величину}]{\text{Инвертировать}} 001$

Шаг 2. $001 + 1 = 010$

Шаг 3. $\boxed{1}\ 010$

Дополнение до единицы

Дополнение до единицы числа получается простым инвертированием каждого бита, за исключением знакового. Поэтому логическая процедура получения дополнения до единицы состоит из двух шагов.

Шаг 1. Инвертировать значение числа, не меняя знаковый бит.

Шаг 2. Присоединить первоначальный знаковый бит к результату, полученному на шаге 1.

Пример. Преобразование числа –6 в дополнение до единицы.

Шаг 1. $\boxed{1}\ 110 \xrightarrow[\text{величину}]{\text{Инвертировать}} 001$

Шаг 2. Дополнение до единицы равно $\boxed{1}\ 001$

Обратите внимание, что дополнение числа до двух может быть получено прибавлением единицы к дополнению этого числа до единицы.

Другие представления с помощью дополнения

Процедуры получения других дополнений могут быть выведены аналогично.

Дополнение до семи для восьмеричной системы.

Шаг 1. Определить дополнение до семи для каждой цифры значения числа путем вычитания этой цифры из семи.

Шаг 2. Восстановить знаковый бит.

Пример. Преобразование числа, значение которого со знаком равна -634_8, в его дополнение до семи:

$\boxed{-}\ 634 \xrightarrow{\overline{7}} \boxed{-}\ 143$

Дополнение до восьми для восьмеричной системы.

Шаг 1. Определить дополнение числа до семи.

Шаг 2. Прибавить единицу к дополнению этого числа до семи.

Пример. Преобразование числа, значение которого со знаком равна -634_8, в его дополнение до восьми:

Шаг 1. $\boxed{-}\ 634 \xrightarrow{\overline{7}} \boxed{-}\ 143$

Шаг 2. $\boxed{-}\ 143 + 1 = \boxed{-}\ 144$

Поскольку в ЭВМ PDP-11 восьмеричная система счисления используется для ввода-вывода данных, дополнение до восьми оказывается очень полезным для проверки результатов, полученных от машины. Приводимые ниже примеры иллюстрируют параллелизм между дополнением до единицы и дополнением до семи, а также между дополнением до двух и дополнением до восьми.

Пример 1. Пусть содержащееся в одном байте двоичное число в виде значения со знаком $X = \boxed{1}\ 0\ 101\ \ 011_2$. Тогда дополнение до единицы X будет

$$X(\overline{1}) = \boxed{1}\ 1\ 010\ \ 100_2$$

Здесь $X(\overline{1})$ обозначает дополнение до единицы числа X. Оно имеет следующее представление с помощью дополнения до семи:

$$X(\overline{7}) = -124_8 = \boxed{1}\ 124_8 = 324_8 \text{ (с учетом знакового бита)}$$

Но $X(\overline{7})$ может быть выведено и из первоначального числа. Поскольку $X = -053_8$, то

$$X(\overline{7}) = -124_8 = \boxed{1}\ 124_8 = 324_8 \text{ (с учетом знакового бита)}$$

Результат получается тем же самым.

Обратите внимание, что, поскольку самая старшая цифра восьмеричного представления имеет в данном случае только один бит, мы для этой цифры вместо дополнения до семи берем дополнение до единицы. Аналогично можно вывести дополнение числа X до двух:

$$|X(\overline{2})| = |X(\overline{1})| + 1 = \square\ 1\ 010\ \ 101$$

Восстанавливая знаковый бит, получаем

$$X(\overline{2}) = \boxed{1}\ 1\ 010\ \ 101$$

где $|X(\overline{2})|$ обозначает значение $X(\overline{2})$.

$$X(\overline{8}) = -125_8 = \boxed{1}\ 125_8 = 325_8 \text{ (с учетом знакового бита)}$$

Представление с помощью дополнения до восьми этого числа, получаемое прямым преобразованием $X(\overline{2})$. Но

$$|X(\overline{8})| = |X(\overline{7})| + 1 = \square\ 124 + 1 = \square\ 125$$

Таким образом,

$$X(\overline{8}) = 1\ \ 125 = 325_8 \text{ (с учетом знакового бита)}$$

В качестве другого примера рассмотрим 16-битовое двоичное число.

Пример 2. Пусть $X = 1\ 000\ 000\ 101\ 100\ 001_2$. Тогда процесс может быть выполнен следующим образом:

$$\begin{array}{l} \boxed{1}\ 000\ 000\ 101\ 100\ 001 \\ \boxed{1}\ 111\ 111\ 010\ 011\ 110 \end{array} \Big\} \overline{1}$$

А значит, мы имеем

$$X(\bar{1}) = \boxed{1}\ 111\ 111\ 010\ 011\ 110$$
$$|X(\bar{2})| = |X(\bar{1})| + 1 = \square\ 111\ 111\ 010\ 011\ 111$$
$$X(\bar{2}) = \boxed{1}\ 111\ 111\ 010\ 011\ 111$$
$$X(\bar{7}) = \boxed{1}\ 77236$$
$$|X(\bar{8})| = |X(\bar{7})| + 1 = \square\ 7\ 72\ 36 + 1 = \square\ 7\ 72\ 37$$

и

$$X(\bar{8}) = \boxed{1}\ 7\ 72\ 37$$

Если вывести дополнение до восьми из первоначального числа, то

$$X = \boxed{1}\ 000\ 000\ 101\ 100\ 001_2$$
$$= \boxed{1}\ 0\ 05\ 41_8$$

Тогда

$$X(\bar{7}) = \boxed{1}\ 7\ 72\ 36$$

и

$$|X(\bar{8})| = |X(\bar{7})| + 1 = \square\ 7\ 72\ 36 + 1 = \square\ 7\ 72\ 37$$

или

$$X(\bar{8}) = \boxed{1}\ 7\ 72\ 37$$

Аналогично могут быть выведены представления с помощью **дополнения до девяти** и **десяти** для десятичной системы счисления и **до пятнадцати и шестнадцати** для шестнадцатеричной системы счисления. То есть

$$|X(\overline{10})| = |X(\overline{9})| + 1 \text{ и } |X(\overline{16})| = |X(\overline{15})| + 1 \text{ и т. п.}$$

Свойства представления с помощью дополнения

Свойство цикличности. Дополнение дополнения числа совпадают с первоначальным представлением числа. Например, пусть $X = \boxed{1}\ 0\ 101\ 011$. Тогда $X(\bar{\bar{2}}) = \boxed{1}\ 1\ 010\ 101$. Если мы снова возьмем дополнение до двух числа $X(\bar{\bar{2}})$, то это даст первоначальное число $X = \boxed{1}\ 0\ 101\ 011$.

Арифметический сдвиг вправо. При арифметическом сдвиге вправо знаковый бит сдвигается в освобождающиеся битовые позиции в левой части значения числа. Пусть, например, $X = \boxed{1}\ 0\ 101\ 011$. Тогда после выполнения двух операций арифметического сдвига вправо $X = \boxed{1}\ 1\ 101\ 010$.

Арифметический сдвиг влево. Арифметический сдвиг влево может выполняться только тогда, когда самый старший бит числа имеет то же самое значение, что и знаковый бит. Например, число $X = \boxed{1}\ 0\ 101\ 011$ не может быть сдвинуто влево. Но если $X = \boxed{1}\ 1\ 101\ 001$, то его можно сдвигать влево не более двух раз. И после двух арифметических сдвигов влево $X = \boxed{1}\ 0\ 100\ 100$.

Дополнение положительного числа есть само число. Например, $X(\bar{\bar{2}})$ числа 00 101 011 есть 00 101 011. В табл. 3.2 показаны представления в виде значения со знаком и их эквивалентные представления по различным основаниям системы счисления.

Таблица 3.2

Двоично-кодированное значение со знаком	Десятичный эквивалент	Дополнение до единицы	Дополнение до двух	Дополнение до семи	Дополнение до восьми
0 0 0 0	0	0 0 0 0	0 0 0 0	0 0	0 0
0 0 0 1	1	0 0 0 1	0 0 0 1	0 1	0 1
0 0 1 0	2	0 0 1 0	0 0 1 0	0 2	0 2
0 0 1 1	3	0 0 1 1	0 0 1 1	0 3	0 3
0 1 0 0	4	0 1 0 0	0 1 0 0	0 4	0 4
0 1 0 1	5	0 1 0 1	0 1 0 1	0 5	0 5
0 1 1 0	6	0 1 1 0	0 1 1 0	0 6	0 6
0 1 1 1	7	0 1 1 1	0 1 1 1	0 7	0 7
1 0 0 0	–0	1 1 1 1	0 0 0 0	1 7	1 0
1 0 0 1	–1	1 1 1 0	1 1 1 1	1 6	1 7
1 0 1 0	–2	1 1 0 1	1 1 1 0	1 5	1 6
1 0 1 1	–3	1 1 0 0	1 1 0 1	1 4	1 5
1 1 0 0	–4	1 0 1 1	1 1 0 0	1 3	1 4
1 1 0 1	–5	1 0 1 0	1 0 1 1	1 2	1 3
1 1 1 0	–6	1 0 0 1	1 0 1 0	1 1	1 2
1 1 1 1	–7	1 0 0 0	1 0 0 1	1 0	1 1

3.4. АРИФМЕТИКА ДОПОЛНЕНИЙ

Большинство цифровых ЭВМ в настоящее время располагают только одним аппаратным сумматором, а операция вычитания реализуется с помощью арифметики дополнений. Поскольку дополнение положительного числа совпадает с самим числом, то мы сосредоточим наше внимание на выполнении вычитания путем сложения чисел, представленных в виде дополнения. Проиллюстрируем основную концепцию на простом примере.

На рис. 3.1 показан диск десятичного счетчика, напоминающего домашний счетчик расхода электроэнергии или газа, или воды. Пусть для сложения стрелка продвигается по часовой стрелке, для вычитания — против часовой стрелки. Сейчас стрелка указывает на цифру 6. Вычтем из текущего числа 6 число 2, т. е. передвинем стрелку против часовой стрелки на два деления. В результате стрелка передвинется на цифру 4. Теперь воспользуемся арифметикой дополнений следующим образом. Поскольку мы имеем дело с десятичной системой счисления, воспользуемся представлением с помощью дополнения до десяти. Заметим, что $6 - 2 = 6 + (-2)$. Дополнение до десяти числа –2 равно 8. Если мы прибавим дополнение до десяти числа –2 к числу 6, то получим число 14. Игнорируя перенос (поскольку у нас только одна цифра), получаем тот же самый результат, что и при обычном вычитании, т. е. 4. Используя ту же логику и обращаясь к рис. 3.1, мы можем повернуть стрелку по часовой стрелке на 8 позиций, и она также остановится на цифре 4, что является правильным ответом. Обратите внимание, что мы можем определить дополнение до десяти любого числа, которое меньше или равно десяти. Для –2 мы имеем 8, для –3 — это 7 и т. д. Такой подход применим к арифметике любых дополнений.

ВЫЧИТАНИЕ ПУТЕМ СЛОЖЕНИЯ ДОПОЛНЕНИЙ ДО ДВУХ

На рис. 3.2 показан процесс выполнения вычитания путем сложения дополнений до двух. Символ В/З обозначает число, представленное значением со знаком, а символ $\bar{2}$

Рис. 3.1. Арифметика дополнений

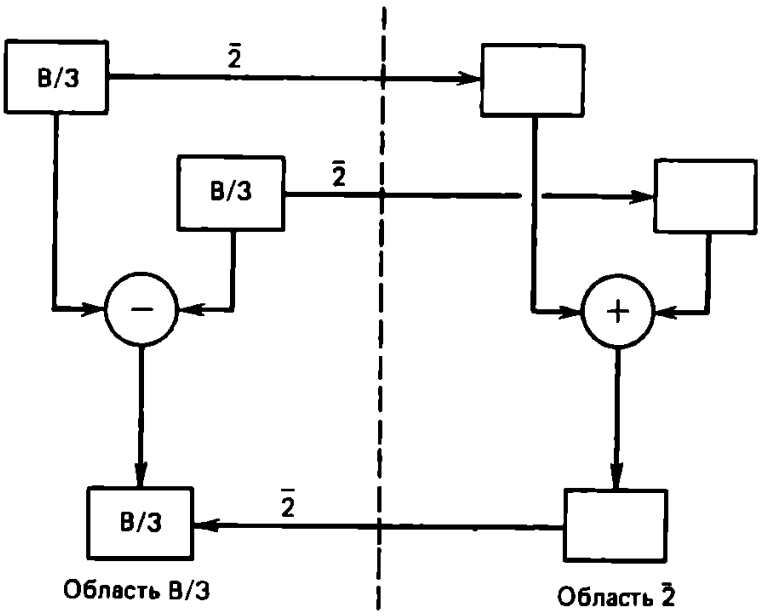


Рис. 3.2. Процесс выполнения вычитания путем сложения дополнений до двух

подразумевает операцию получения дополнения до двух. Процесс вычитания осуществляется с помощью следующих шагов.

Шаг 1. Преобразовать отрицательное число или числа, представленные значением со знаком, в их дополнения до двух, но оставив число неизменным, если оно положительное.

Шаг 2. Применить операцию двоичного сложения и интерпретировать знаковый бит наравне с другими битами, но игнорируя бит переноса.

Шаг 3. Если знаковый бит суммы равен единице, результат есть отрицательное число в виде дополнения до двух. Для обратного преобразования дополнения к значению со знаком следует вновь получить дополнение до двух.

Пример 1:

$$19_{10} - 14_{10} = ?$$

В/З:

$$19_{10} = \boxed{0}\ 10\ 011_2$$
$$-14_{10} = \boxed{1}\ 01\ 110_2$$

Шаг 1. Преобразовать число −14 в дополнение до двух

$$\begin{array}{l} \boxed{1}\ 01\ 110 \\ \quad\ 10\ 001 \\ \quad\ \ +\ 1 \\ \hline \boxed{1}\ 10\ 010 \end{array} \quad \begin{array}{l} \bar{1} \\ \\ \bar{2} \end{array}$$

Шаг 2.

$$\begin{array}{rl} & \boxed{0}\ 10\ 011 \\ +) & \boxed{1}\ 10\ 010 \\ \hline & (1)\ \boxed{0}\ 00\ 101 \end{array}$$

Игнорировать бит переноса

Шаг 3. Сумма равна 5.

Пример 2:

$$7_{10} - 14_{10} = ?$$

В/З:

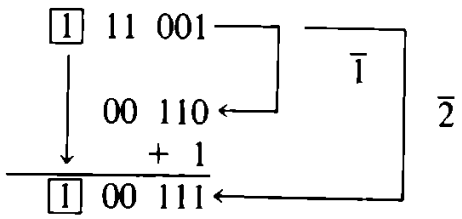
$$7_{10} = \boxed{0}\ 00\ 111_2$$
$$-14_{10} = \boxed{1}\ 01\ 110_2$$

Шаг 1. $-14_{10} \xrightarrow{\bar{2}} \boxed{1}\ 10\ 010$

Шаг 2.

$$\begin{array}{r} \boxed{0}\ 00\ 111 \\ +)\boxed{1}\ 10\ 010 \\ \hline \boxed{1}\ 11\ 001 \end{array}$$

Шаг 3.



Ответ в виде В/З: -7_{10}.

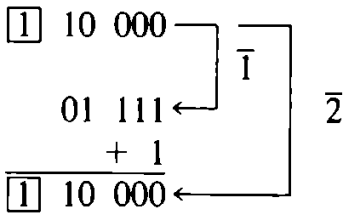
Пример 3:

$$-16_{10} - 14_{10} = ?$$

В/З:

$$-16_{10} = \boxed{1}\ 10\ 000$$
$$-14_{10} = \boxed{1}\ 01\ 110_2$$

Шаг 1. Преобразовать В/З в дополнение до двух



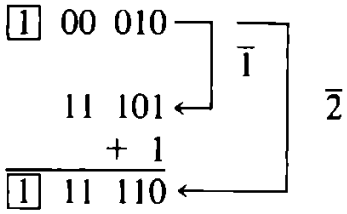
$$-14 \xrightarrow{\bar{2}} 1\ 10\ 010$$

Шаг 2.

$$\begin{array}{r} \boxed{1}\ 10\ 000 \\ +\quad \boxed{1}\ 10\ 010 \\ \hline (1)\ \boxed{1}\ 00\ 010 \end{array}$$

Игнорировать бит переноса

Шаг 3.



Ответ в виде В/З: -30_{10}.

ПЕРЕПОЛНЕНИЕ В АРИФМЕТИКЕ ДОПОЛНЕНИЙ

Переполнение сопряжено с получением ошибочного результата, о чем должны знать все программисты. Переполнение часто путают с результатом, приводящим к переносу. В большинстве вычислительных систем имеется стандартная системная программа, которая в случае появления переполнения выдает сообщение об ошибке и затем останавливает работу машины. Однако если в системном программном обеспечении нет программного модуля для реализации такого сервиса, то ответственность за конт-

роль бита переполнения в PSW ложится на программиста. Если происходит переполнение, то следует выдать сообщение об ошибке и остановить машину. Рассмотрим следующие примеры.

Пример 1:

$18_{10} + 16_{10} = ?$ для 6-битового АЛУ

Шаг 1. $18_{10} = \boxed{0}\ 1\ 00\ 10$
$16_{10} = \boxed{0}\ 1\ 00\ 00$

Шаг 2.

$$\begin{array}{r} \boxed{0}\ 1\ 00\ 10 \\ +)\ \boxed{0}\ 1\ 00\ 00 \\ \hline \boxed{1}\ 0\ 00\ 10 \end{array}$$

Шаг 3.

$$\begin{array}{l} \boxed{1}\ 0\ 00\ 10 \\ \quad\ 1\ 11\ 01 \\ \quad\quad +\ 1 \\ \hline \boxed{1}\ 1\ 11\ 10 \end{array}$$

$\overline{1}$, $\overline{2}$

Ответ равен –30, хотя мы ожидали, что он будет равен 34. Обратите внимание, что эти действия выполняются на 6-битовой машине, в которой могут быть представлены только числа в диапазоне от -31_{10} до $+31_{10}$. Однако в этом примере результат оказывается за пределами диапазона АЛУ, что приводит к формированию ошибочного результата с переполнением; машина должна быть остановлена.

Пример 2:

$-18_{10} - 17_{10} = ?$ для 6-битового АЛУ

В/З: $-18 = \boxed{1}\ 1\ 00\ 10$
$-17 = \boxed{1}\ 1\ 00\ 01$

Шаг 1.

$$\begin{array}{l} \boxed{1}\ 1\ 00\ 10 \\ \quad\ 0\ 11\ 01 \\ \quad\quad +\ 1 \\ \hline \boxed{1}\ 0\ 11\ 10 \end{array}$$

$\overline{1}$, $\overline{2}$

$$\begin{array}{l} \boxed{1}\ 1\ 00\ 01 \\ \quad\ 0\ 11\ 10 \\ \quad\quad +\ 1 \\ \hline \boxed{1}\ 0\ 11\ 11 \end{array}$$

$\overline{1}$, $\overline{2}$

Шаг 2.

$$\begin{array}{r} \boxed{1}\ 0\ 11\ 10 \\ +)\ \boxed{1}\ 0\ 11\ 11 \\ \hline (1)\ \boxed{0}\ 1\ 11\ 01 \end{array}$$

Шаг 3. Ответ равен +29, что, очевидно, неверно.

Внимательно изучив процесс переполнения, заключаем, что если значения знаковых бит двух чисел одинаковы, но знаковый бит суммы этих двух чисел имеет значение, отличное от первоначального, то произошло переполнение. Из приведенных выше примеров видно, что в примере 1 первоначальные знаки были положительны, но это

дало отрицательный знак; в примере 2 результат имеет положительный знак, хотя первоначальные знаки были отрицательные.

ВЫЧИТАНИЕ ПУТЕМ СЛОЖЕНИЯ ДОПОЛНЕНИЙ ДО ЕДИНИЦЫ

Хотя в семействе ЭВМ PDP-11 применяется арифметика дополнений до двух, в ряде цифровых ЭВМ используется арифметика дополнений до единицы. Процесс выполнения вычитания путем сложения дополнений до единицы осуществляется с помощью следующих шагов.

Шаг 1. Преобразовать отрицательное число или числа в дополнение до единицы.

Шаг 2. Применить операцию двоичного сложения с участием знакового бита. Если происходит переполнение, прибавить 1 к результирующей сумме.

Шаг 3. Если знаковый бит результата есть 1, опять применить операцию получения дополнения до единицы для получения представления в виде значения со знаком; в противном случае никакого преобразования не нужно.

Пример 1:

$34_{10} - 23_{10} = ?$ для 8-битового АЛУ

В/З: $34_{10} = \boxed{0}\ 0\ 10\ 00\ 10_2$

$-23_{10} = \boxed{1}\ 0\ 01\ 01\ 11_2$

Шаг 1.

$\boxed{1}\ 0\ 01\ 01\ 11$ → $\overline{1}$ → $\boxed{1}\ 1\ 10\ 10\ 00$

Шаг 2.

$\boxed{0}\ 0\ 10\ 00\ 10$

$+)\ \boxed{1}\ 1\ 10\ 10\ 00$

$(1)\ 0\ \ 0\ 00\ 10\ 10$

$+\ 1$

$0\ \ 0\ 00\ 10\ 11$

Произошел перенос

Шаг 3. Ответ равен +11.

Пример 2:

$17_{10} - 23_{10} = ?$ для 8-битового АЛУ

В/З: $17_{10} = \boxed{0}\ 0\ 01\ 00\ 01_2$

$-23_{10} = \boxed{1}\ 0\ 01\ 01\ 11$

Шаг 1.

$\boxed{1}\ 0\ 01\ 01\ 11$ → $\overline{1}$ → $\boxed{1}\ 1\ 10\ 10\ 00$

Шаг 2.

$\boxed{0}\ 0\ 01\ 00\ 01$

$+)\boxed{1}\ 1\ 10\ 10\ 00$

$\boxed{1}\ 1\ 11\ 10\ 01$

Шаг 3.

$\boxed{1}\ 1\ 11\ 10\ 01$ → $\overline{1}$ → $\boxed{1}\ 0\ 00\ 01\ 10$

Ответ равен –6.

Пример 3:

$-22_{10} - 8_{10} = ?$ для 8-битового АЛУ

В/З: $-22_{10} = [1]\ 0\ 01\ 01\ 10$

$-8_{10} = [1]\ 0\ 00\ 10\ 00$

Шаг 1. [1] 0 01 01 10 → $\overline{1}$ → [1] 1 10 10 01

[1] 0 00 10 00 → $\overline{1}$ → [1] 1 11 01 11

Шаг 2.

```
    [1] 1 10 10 01
+)  [1] 1 11 01 11
(1) [1] 1 10 00 00
                +1
    [1] 1 10 00 01
```

Шаг 3. [1] 1 10 00 01 → $\overline{1}$ → [1] 0 01 11 10

Ответ равен –30.

ВЫЧИТАНИЕ ПУТЕМ СЛОЖЕНИЯ ДОПОЛНЕНИЙ ДО ВОСЬМИ

Поскольку в семействе ЭВМ PDP-11 для ввода-вывода или взаимодействия пользователя с машиной широко используется восьмеричная система счисления, мы продемонстрируем, как следующая процедура может быть применена к 16-битовой машине.

Шаг 1. Преобразовать отрицательное число или числа в дополнение до восьми.

Шаг 2. Применить операцию восьмеричного сложения с участием знакового бита и игнорируя перенос, если он происходит.

Шаг 3. Если знаковый бит есть 1, опять применить дополнение до восьми для получения представления в виде значения со знаком.

Пример. Для 16-битового АЛУ $715_8 - 234_8 = ?$

Шаг 1.

```
[0] 0 07 15
[1] 0 02 34 → 7 → 7 75 43 → 8 →
        + 1
[1] 7 75 44
```

Шаг 2.

```
    [0] 0 07 15
+)  [1] 7 75 44
(1) [0] 0 04 61
```

Игнорировать перенос

Шаг 3. Ответ равен 461_8. Помните, что мы здесь используем восьмеричную арифметику!

Этот пример можно также выполнить, используя сложение дополнений до двух:

В/З. $715_8 =$ [0] 000 000 111 001 101
$-234_8 =$ [1] 000 000 010 011 100

```
Шаг 1.  [1] 000 000 010 011 100 ─┐  ─┐
                                  │ 1̄ │
            111 111 101 100 011 ←┘    │ 2̄
                              + 1     │
        ─────────────────────────     │
        [1] 111 111 101 100 100 ←─────┘

Шаг 2.          [0] 000 000 111 001 101
        +)      [1] 111 111 101 100 100
        ───────────────────────────────
           (1)  [0] 000 000 100 110 001
```

Игнорировать перенос

Шаг 3. Ответ равен 461_8, что идентично результату при использовании арифметики дополнений до восьми.

ВЫЧИТАНИЕ ПУТЕМ СЛОЖЕНИЯ ДОПОЛНЕНИЙ ДО ШЕСТНАДЦАТИ

Следуя той же логике, мы можем вывести процедуру выполнения вычитания путем сложения дополнений до шестнадцати. Для демонстрации этого процесса будет достаточно следующего числового примера: для 16-битового АЛУ $7C23_{16} - 1F96_{16} = ?$.

```
Шаг 1.  [0] 7C 34
        [1] 1F 96 ─┐  ─┐
                   │15̄ │
            60 69 ←┘   │ 16̄
               + 1     │
        ──────────     │
        [1] 60 6A ←────┘

Шаг 2.      [0] 7C 34
        +)  [1] 60 6A
        ─────────────
        (1) [0] 5C 9E
```

Игнорировать перенос

Шаг 3. Ответ равен 5C 9E.

Теперь проверим ответ этого примера, вновь воспользовавшись дополнением до двух.

```
Шаг 1.  [0] 111 110 000 110 100
        [1] 001 111 110 010 110 ─┐
                                  │ 1̄
            110 000 001 101 001 ←┘
                              + 1
        ─────────────────────────
        [1] 110 000 001 101 010
```

Шаг 2.

```
        [0] 111 110 000 110 100
+)      [1] 110 000 001 101 010
-----------------------------------
  (1)   [0] 101 110 010 011 110
```

Шаг 3. Результат идентичен результату, получаемому при использовании шестнадцатеричной операции.

3.5. ПРЕДСТАВЛЕНИЕ ЧИСЕЛ С ПЛАВАЮЩЕЙ ТОЧКОЙ

До сих пор изучение представления чисел ограничивалось целыми числами. Для представления дробных или вещественных чисел в большинстве цифровых ЭВМ применяется двоичная **система с плавающей точкой.** В этой системе представление числа состоит из трех полей: 1) знака; 2) экспоненты; 3) дробной части. В ЭВМ PDP-11 такая система обычно требует двух слов памяти со следующим форматом:

Знак	Экспонента	Дробная часть
1 бит	8 бит	23 бита

В качестве необязательного оборудования для пользователей ЭВМ PDP-11 доступен аппаратно-реализованный процессор, выполняющий набор инструкций для арифметики с плавающей точкой. Здесь мы имеем дело только с целыми числами; более подробное описание представления чисел с плавающей точкой будет изложено ниже.

3.6. ПЕРВОЕ ЗНАКОМСТВО С НАБОРОМ ИНСТРУКЦИЙ

В предыдущих разделах мы рассмотрели представление чисел, которое является одним из ключевых элементов при переносе информации в ЭВМ и из нее. Теперь мы познакомимся с другим ключевым элементом – набором инструкций ЭВМ PDP-11. Поскольку PDP-11 – мощная и гибкая мини-ЭВМ, ее набор инструкций для новичка представляется достаточно сложным. Для облегчения понимания сначала мы изучим простые и наиболее ясные инструкции и их типичные применения, а более сложные инструкции отложим до следующих разделов.

ФОРМАТ ИНСТРУКЦИЙ И МАШИННЫЕ КОДЫ

Одни инструкции ЭВМ PDP-11 имеют два операнда, другие – один операнд или вообще не имеют операндов. Как описано в гл 1, двухоперандная инструкция имеет три поля: одно для кода операции и два для операндов, где помещаются адреса данных. Мнемонический код операции и операнды разделяются одним или несколькими пробелами; для разделения операндов используется запятая. Например:

```
ADD R0, R1
```

Кроме того, для удобства, эффективности и гибкости программирования программистом могут добавляться два необязательных поля: **символьный адрес** и **поле комментария.** Что касается самой ЭВМ, то эти необязательные поля для нее бессмысленны, поскольку ЦП использует числовой адрес, по которому размещается в памяти оператор инструкции. Например, если эта инструкция помещена по адресу 1000, то ЦП извлечет и выполнит ее только тогда, когда регистр PC укажет на этот числовой адрес. Однако пользователи во многих случаях предпочитают символьные адреса, на которые легче ссылаться. Например, если на Солнечном бульваре, дом 1794 есть торговый центр, то мы, скорее всего, будем называть его каким-либо конкретным именем, скажем Солнечный дворец, нежели станем ссылаться на него по его адресу. Кроме того, при выполнении этой инструкции ЦП только складывает содержимое регистров R0 и R1 и помещает сумму в регистр R1, и не обращает внимания на содержимое этих регистров.

Но эту же инструкцию мы можем использовать в нескольких местах программы. Обычно удобно записывать для себя в поле комментариев, какие действия, порождаемые этой инструкцией, для каких целей предназначены. Например, в одном месте программы в регистре R0 может содержаться стоимость товара, а в регистре R1 – значение налога. Тогда после выполнения инструкции в регистре будет содержаться полная цена. В другом месте (где рассматривается прибыль) мы можем поместить в регистр R0 значение основного капитала, а в регистр R1 – значение процентов. Тогда после выполнения инструкции в регистре R1 будет содержаться полное значение основного капитала вместе с процентами и программа будет такой:

```
Символьный адрес    Инструкции      Комментарии
-----------------------------------------------
MER1:               ADD R0, R1      ; ПОЛНАЯ СТОИ-
                                    МОСТЬ
INCOME:             ADD R0, R1      ; ОСНОВНОЙ КА-
                                    ; ПИТАЛ И ПРО-
                                    ; ЦЕНТЫ
```

В данном случае мы воспользовались соглашениями, принятыми для ЭВМ PDP-11, т. е. мы используем двоеточие для отделения символьного адреса от мнемонического кода операции и точку с запятой для отделения поля комментариев от операндов. Обратите внимание, что мы использовали метку MER1 (для товаров) и метку INCOME (для вычисления полного капитала и процентов) в качестве соответствующих символьных адресов. Определение символьных адресов и заполнение полей комментариев не такой легкий процесс; чтобы символьные адреса и комментарии оказались полезными для программиста и пользователя программы, при их создании приходится приложить немалые усилия.

До сих пор мы рассматривали инструкции, представленные мнемонически, однако вычислительные машины понимают только двоичный машинный язык. Поэтому для каждого оператора инструкции имеется соответствующий уникальный машинный код, что дает возможность центральному процессору интерпретировать и выполнять инструкцию. Каждый производитель ЭВМ для каждого выпускаемого им типа цифровых машин предлагает уникальный набор кодов инструкций. Для ЭВМ PDP-11 существует около 67 инструкций плюс псевдоинструкции. Полное описание набора инструкций PDP-11 можно найти в приложении Б или в карте программиста, выпускаемой фирмой Digital Equipment Corporation (DEC), разработавшей ЭВМ PDP-11. Для приведенного выше примера соответствующий мнемонической инструкции машинный код в двоичном и восьмеричном виде соответственно может быть показан следующим образом:

Мнемоника	ADD	R0,	R1
	Одно слово памяти		
Двоичный код	0 1 1 0	0 0 0 0 0 0	0 0 0 0 0 1
Восьмеричный код	0 6	0 0	0 1

Программу, написанную в мнемоническом коде, называют **исходной** программой, а программы, написанные в двоичном и восьмеричном кодах, – **объектными** программами. Читателю следует помнить, что машина может читать только двоичный код; восьмеричный код является промежуточным, он нужен лишь для нашего удобства.

ФУНКЦИОНАЛЬНАЯ КЛАССИФИКАЦИЯ ИНСТРУКЦИЙ

Возможность использования большого числа инструкций может привести новичка в замешательство, и он не будет знать, с чего же ему начинать. Поэтому будет полезно провести классификацию инструкций по функциональному признаку, чтобы стало ясно, сколько различных функций может предложить ЭВМ программисту.

Транспортировка данных. Инструкции этой группы используются для пересылки или транспортировки данных из одного места в другое.

Пример 1. Инструкция

```
MOV  R1, R3    ; (R1) → R3
```

пересылает содержимое регистра R1 в регистр R3. Содержимое регистра R1 не изменяется, но первоначальное содержимое регистра R3 заменяется первоначальным содержимым регистра R1. Для большей точности вместо MOV нам следовало бы воспользоваться мнемоникой COPY, однако разработчики ЭВМ определили мнемонику MOV, поэтому мы используем именно ее.

Пример 2. Инструкция

```
MOV  # 2, R5    ; # 2 → R5
```

пересылает в регистр R5 число 2.

Манипуляция данными. Инструкции этой группы с помощью АЛУ манипулируют данными, извлеченными из одного или нескольких мест, и помещают результат по адресу назначения, соответствующему выполняемой инструкции.

Пример 1. Инструкция

```
SUB  A, R2    ; (R2) – (A) → R2
```

вычитает содержимое адреса памяти A из содержимого регистра R2 и помещает разность в регистр R2. Содержимое ячейки A не изменяется, а флаги N, Z, V и C в PSW устанавливаются в соответствии с результатом, находящимся в регистре R2.

Пример 2. Инструкция

```
CMP     A,R2            ;(A)-(R2) И PSW УСТАНАВЛИВАЕТСЯ
                        ;В СООТВЕТСТВИИ С РЕЗУЛЬТАТОМ
```

вычитает содержимое регистра R2 из содержимого ячейки A и соответственно устанавливает флаги PSW. Обратите внимание, что на этот раз и содержимое ячейки A, и содержимое регистра R2 остаются без изменения, а вычитание выполняется в обратном порядке: из содержимого ячейки A, т. е. первого операнда, вычитается содержимое регистра R2, т. е. второй операнд.

Логические операции. Инструкции этой группы осуществляют с помощью АЛУ логические манипуляции данными, извлеченными из одного или нескольких мест.

Пример 1. Инструкция

```
BIS  R4, A    ; (R4) ∨ (A) → A
```

выполняет логическую операцию ИЛИ над содержимым регистра R4 и ячейки памяти A. Чтобы операция была более понятной, положим

(R4) = 1 000 000 000 000 101
(A) = 0 000 111 000 111 001

Тогда после выполнения

(R4) – останется без изменения
(A) = 1 000 111 000 111 101

Обратите внимание, что в тех позициях, где в регистре R4 биты являются логическими единицами, соответствующие биты в A устанавливаются в логическую единицу, а остальные

ные биты остаются без изменения. Мнемоника этой инструкции BIS означает операция установки бит.

Пример 2. Инструкция

```
BIT     A,B             ;(A)&(B) И УСТАНОВКА ФЛАГОВ В PSW
```

выполняет логическую операцию **И** над содержимым ячеек памяти A и B и соответственно устанавливает флаги в PSW. Пусть

(A) = 0 000 000 000 000 111
(B) = 1 101 111 110 001 000

Тогда после выполнения

(A) – останется без изменения
(B) – останется без изменения

Но флаги в PSW станут такими:

перенос:	C – без изменения
переполнение:	V = 0
нуль:	Z = 1
знак:	N = 0

Обратите внимание, что в этом примере мы хотим проверить, являются ли все три самых младших бита (B) нулевыми или нет. Именно поэтому эта инструкция имеет мнемонический код BIT, что означает проверку бит.

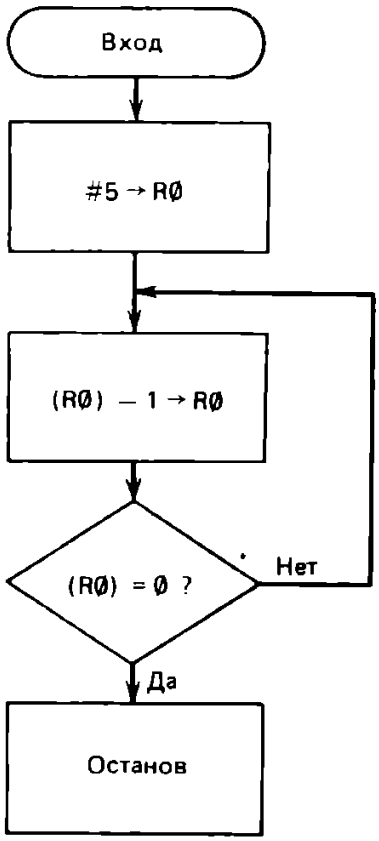
Управление последовательностью выполнения. Инструкции этой группы позволяют осуществить в программе различные пути выполнения в зависимости от текущих условий или статуса в PSW. Эта группа инструкций предназначается для принятия решений. Например:

```
START:  MOV     #5,R0           ;ПЕРВОНАЧАЛЬНО (R0)=5
LOOP:   DEC     R0              ;(R0)-1 -> R0
        BNE     LOOP            ;ЕСЛИ ФЛАГ Z В PSW НЕ 1,
                                ;ВЕТВЛЕНИЕ К LOOP, ИНАЧЕ
                                ;ПРОДОЛЖАТЬ
QUIT:   HALT                    ;ОСТАНОВИТЬ РАБОТУ МАШИНЫ
```

При записи этой программы использован формат, уже описанный выше. Здесь мы применяем три символьных адреса: START, LOOP и QUIT. Выполняемые программой действия могут быть описаны блок-схемой, изображенной на рис. 3.3. Овальные символы означают точки входа и выхода программы; ромбами обозначается ветвление или процесс принятия решений. Программа работает следующим образом:

1. Инициализировать содержимое регистра R0 значением 5.
2. Уменьшить (R0) на единицу. Результат может быть нулевым или ненулевым, в соответствии с чем устанавливается флаг Z в PSW.
3. Инструкция управления последовательностью выполнения BNE (ветвление, если не равно нулю) заставляет ЦП проверить флаг Z в PSW и действовать соответственно его значению. Вспоминаем, что Z = 1 (1 означает "истина") подразумевает, что результатом выполнения предыдущей инструкции был нуль, а Z = 0 (0 означает "ложь") подразумевает ненулевой результат предыдущей инструкции. Таким образом, если Z = 0, то результат был ненулевым. Тогда в программе осуществляется ветвление назад к ячейке LOOP и выполняется вновь инструкция DEC (уменьшить на 1). Этот цикл будет повторяться до тех пор, пока не станет (R0) = 0 и Z = 1. Тогда будет выполнена следующая инструкция, HALT, и работа машины прекратится.

Рис. 3.3. Блок-схема простой программы с ветвлением



Псевдоинструкции или директивы. Инструкции этой группы весьма отличаются от тех, что мы описывали до сих пор. Эти инструкции выполняются не процессором во время выполнения программы, а системной программой **ассемблером** еще до того, как программа доводится до вида, готового к выполнению. Это может показаться запутанным, но прояснить суть дела помогут проводимые ниже примеры инструкций такого типа.

Пример 1. Рассмотрим фрагмент программы:

```
        .
        .
        .
DATA:   .WORD   7,16,63         ;ОБРАТИТЕ ВНИМАНИЕ НА ТОЧКУ
                                ;ПЕРЕД СЛОВОМ WORD. ЕЕ НЕЛЬЗЯ ОПУСКАТЬ.
        .
        .
        .
```

Здесь используется псевдоинструкция .WORD для размещения данных 7, 16 и 63 в последовательных ячейках памяти, начиная от символьного адреса DATA. Если метка DATA = 1020, то карта памяти для этой части программы будет такой:

Числовой адрес	Содержимое
.	
.	
.	
10 20	00 00 07
10 22	00 00 16
10 24	00 00 63
.	.
.	.
.	.

Заметим, что во многих случаях программист стремится включить в программу некоторые данные, которые она могла бы использовать во время выполнения. Операнды этой псевдоинструкции являются не чем иным, как числами, причем недопустимо, чтобы ЦП рассматривал их как последовательность кодов мнемонических инструкций и пытался их выполнить. Поэтому нам необходим какой-то способ, позволяющий сообщить ЦП, что эти "числа" не должны рассматриваться как инструкции, а будут

использоваться только как данные при вычислениях. Инструкция .WORD предписывает системной программе ассемблеру поместить эти операнды или данные в память, начиная с предписанной ячейки, например DATA, как в данном примере. Еще один пример послужит иллюстрацией операции другого типа, но из того же класса псевдоинструкций.

Пример 2. Рассмотрим фрагмент программы:

```
        .
        .
        .
RESULT: .BLKW   N               ;N - НЕКОТОРОЕ ПОЛОЖИТЕЛЬНОЕ
        .                       ;ЦЕЛОЕ ЧИСЛО
        .
        .
        .
```

Этот пример показывает, что программисту желательно, чтобы ассемблер зарезервировал *n* слов памяти, начиная от ячейки с меткой RESALT, куда программа при выполнении поместит результаты своей работы. Перед выполнением системная программа зарезервирует блок из *n* слов и в качестве начальных значений поместит туда нули.

Пример 3:

```
        .
        .
        .
.END
```

Опять эта не выполняемая процессором инструкция, но псевдоинструкция, позволяющая программисту взаимодействовать с системной программой ассемблером и сообщить ему, что в этом месте—конец программы, где ему следует остановить свою работу.

ВЫПОЛНЕНИЕ ПРОГРАММЫ

Чтобы ближе познакомиться с тем, как выполняется программа, давайте представим себя на месте ЦП и выполним вручную следующую простую программу из примера, рассмотренного ранее в пункте об управлении последовательностью выполнения:

```
START:  MOV     #5,R0
LOOP:   DEC     R0
        BNE     LOOP
QUIT:   HALT
```

При выполнении программы выявляются два существенных цикла: 1) цикл извлечения; 2) цикл выполнения. Прежде всего мы должны гарантировать, что программа, которую мы хотим выполнять, находится в оперативной памяти и ожидает своего запуска. Положим, что некоторая системная программа уже загрузила нашу программу с диска в оперативную память, а разработчик системы принял решение, что первая инструкция программы всегда помещается в некоторую фиксированную ячейку, например в ячейку с адресом 1000. Тогда для данного примера START = 1000, т. е. инструкция MOV ≠ 5, R0 в оперативной памяти будет размещена с ячейки 1000. Из дальнейшего материала читателю будет ясно, что эта инструкция займет два слова памяти. Таким образом, вся программа потребует пяти слов памяти или десяти байт. Обращаясь к таблице кодирования инструкций, с которой читатель познакомится позднее в разделе, где рассматривается ассемблирование, вручную мы можем построить машинные коды или карту памяти программы. Для удобства ниже показана программа в мнемоническом виде вместе с соответствующими машинными кодами в оперативной памяти; такая совокупность называется картой памяти.

Исходная программа			Объектная программа в оперативной памяти (карта памяти)	
Символьный адрес	Инструкция (мнемоника)		Числовой адрес	Содержимое
START	MOV	# 5, R0	001000	012700
			001002	000005
LOOP	DEC	R0	001004	005300
	BNE	LOOP	001006	001376
QUIT	HALT		001010	000000

Обратите внимание, что значения, указанные в колонке числового адреса, увеличиваются с шагом в два байта, так что они всегда являются четными числами. Если только какая-либо инструкция не задает специально байтовую операцию, адресация в машине выполняется на пословной основе.

Чтобы выполнить программу, мы должны выдать команду одной из системных программ, которая запустит нашу программу, начиная с адреса 1000. Формат такой команды зависит от конкретной системы; обычно команда имеет формат 1000G, где буква G обозначает слово GO, означающее в данном случае "запустить". Эта команда приведет к тому, что машина поместит в регистр PC значение 1000, а затем выполнит циклы извлечения и выполнения.

Цикл извлечения

Шаг 1. Содержимое регистра PC (1000) переносится в регистр РАП, а (PC) автоматически увеличивается на 2.

Шаг 2. Содержимое ячейки, определенной регистром РАП, переносится в регистр РДП, а затем в регистр РИ.

Шаг 3. Устройство управления декодирует инструкцию, находящуюся в регистре РИ, и определяет, занимает ли инструкция одно или два слова. Если справедливо последнее, то шаги 1 и 2 повторяются.

В этом примере инструкция занимает два слова, поэтому текущее содержимое регистра PC (1002) переносится в регистр РАП, (PC) становится равным 1004, а содержимое ячейки 1002, т. е. 000005, переносится в регистр РДП, а затем в АЛУ.

Цикл выполнения. Поскольку в ЦП теперь находится полная инструкция, устройство управления начинает ее выполнение и помещает число 5 в регистр R0. После завершения цикла выполнения следует цикл извлечения-выполнения. Обратите внимание, что при выполнении инструкции BNE LOOP ЦП проверяет флаг Z. Если он нулевой, то в регистр PC помещается значение символьного адреса LOOP, равное 1004, несмотря на то, что содержимое регистра PC было автоматически увеличено до 1010 во время извлечения и выполнения инструкции BNE LOOP. Однако если Z = 1, то будет извлечена и выполнена следующая инструкция HALT из ячейки 1010. После этого работа машины остановится, но значение содержимого регистра PC окажется скорректированным: 1010 + 2 = = 1012.

ПРОСТАЯ ПРИКЛАДНАЯ ПРОГРАММА

В этом разделе мы продемонстрируем, как рассмотренные выше инструкции могут быть собраны вместе для выполнения определенной задачи. Показанная ниже программа может выбрать наибольшее нечетное число из двух чисел, находящихся в ячейках 1 и 2, и запомнить его в ячейке с меткой ODDBIG. Если оба числа нечетные, то будет запомнено наибольшее из них; если только одно из чисел нечетное, то запоминается это нечетное число; если же ни одно из чисел нечетным не является, то ничего

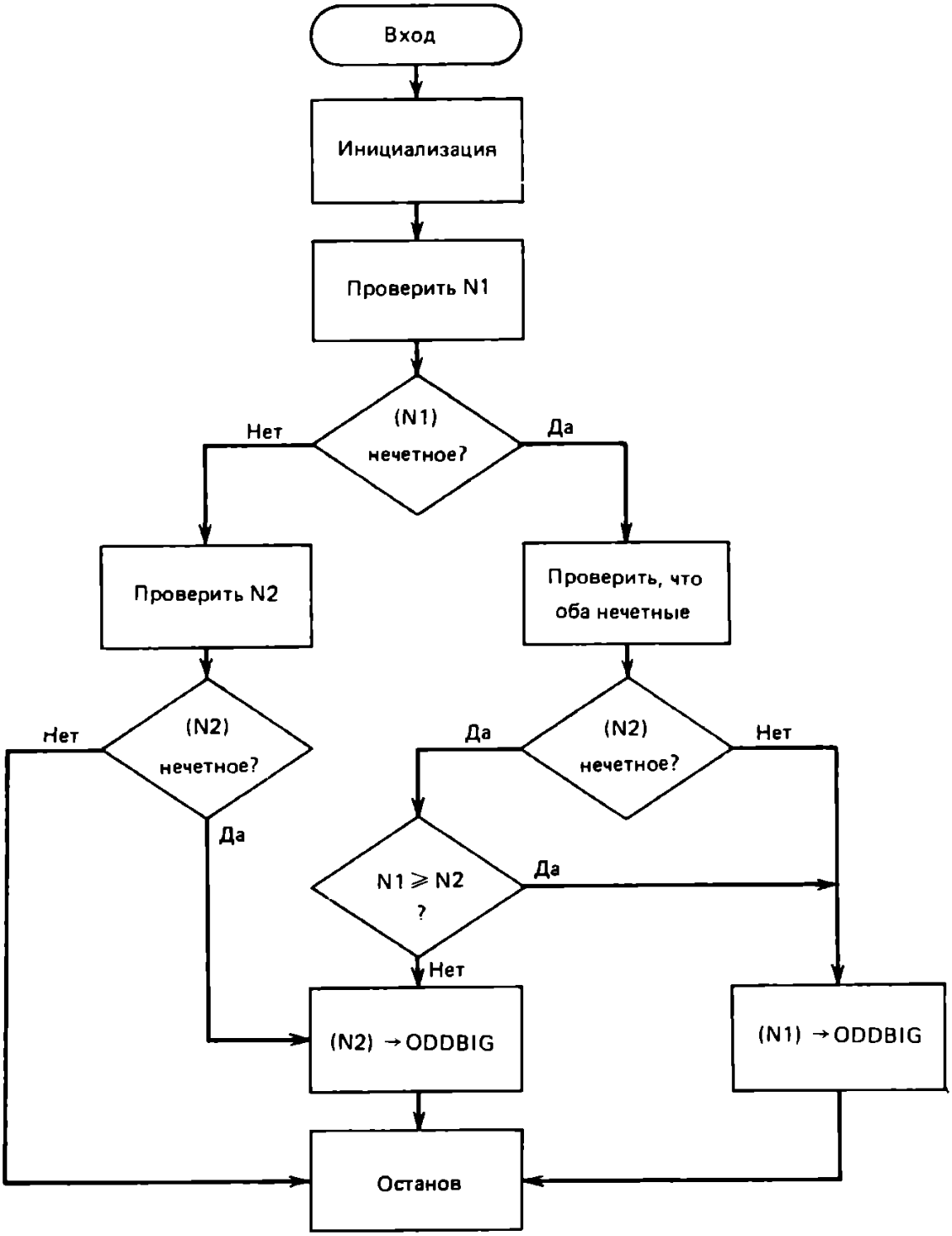


Рис. 3.4. Блок-схема программы поиска наибольшего нечетного числа

делать не нужно. Предложение по тому, как можно разработать такую программу, выражено блок-схемой, показанной на рис. 3.4.

Проанализируем теперь спецификацию программы. Поскольку обычно нам не нужно, чтобы необработанные данные подвергались случайному изменению, то, как правило, первым делом мы перемещаем данные в регистры общего назначения внутри ЦП. Есть и другое преимущество использования регистров общего назначения для манипуляции данными — скорость выполнения операций существенно больше, если они осуществляются полностью внутри ЦП. После переноса данных в регистр мы используем инструкцию проверки бит BIT для проверки самого младшего бита рассматриваемого числа. Если этот бит нулевой, то число четное. Исходная программа для реализации поставленной задачи имеет следующий вид:

```
        ;ЭТО ПРОСТАЯ ПРИКЛАДНАЯ ПРОГРАММА, КОТОРАЯ ДЕМОНСТРИРУЕТ,
        ;КАК МОЖЕТ БЫТЬ СОБРАНА ВМЕСТЕ ПОСЛЕДОВАТЕЛЬНОСТЬ ИНСТРУКЦИЙ
        ;С РАЗЛИЧНЫМИ ФУНКЦИЯМИ С ЦЕЛЬЮ ПОМЕСТИТЬ НАИБОЛЬШЕЕ НЕЧЕТНОЕ
        ;ЧИСЛО В ОПРЕДЕЛЕННУЮ ЯЧЕЙКУ ПАМЯТИ.
D-1)  ODDBIG:   .BLKW   1           ;ОТВЕСТИ СЛОВО ПАМЯТИ ДЛЯ РЕЗУЛЬТАТА
D-2)  N1:       .WORD   X           ;МЕСТО ДЛЯ ДАННЫХ, X - ЦЕЛОЕ
D-3)  N2:       .WORD   Y           ;МЕСТО ДЛЯ ДАННЫХ, Y - ЦЕЛОЕ
  1)  START:    MOV     N1,R1       ;(N1) -> R1
  2)            MOV     N2,R2       ;(N2) -> R2
  3)            MOV     #1,R0       ;МАСКА МЛАДШЕГО БИТА
  4)  LOOK1:    BIT     R0,R1       ;ПРОВЕРИТЬ МЛ. БИТ 1-ГО ЧИСЛА
  5)            BNE     ODD2        ;(N1) НЕЧЕТНОЕ, ПРОВЕРИТЬ N2
  6)  LOOK2:    BIT     R0,R2       ;ПРОВЕРИТЬ МЛ. БИТ 2-ГО ЧИСЛА
  7)            BNE     STOREN2     ;ЗАПОМНИТЬ N2
  8)            BR      QUIT        ;ПЕРЕХОД К МЕТКЕ QUIT
  9)  ODD2:     BIT     R0,R2       ;ОБА ЧИСЛА НЕЧЕТНЫЕ?
 10)            BNE     COMPR       ;ДА, ПЕРЕЙТИ К СРАВНЕНИЮ
 11)  STOREN1:  MOV     R1,ODDBIG   ;ЗАПОМНИТЬ (N1)
 12)            BR      QUIT        ;ПЕРЕХОД К МЕТКЕ QUIT
 13)  COMPR:    CMP     R1,R2       ;СРАВНИТЬ ДВА ЧИСЛА
 14)            BGE     STOREN1     ;(N1) >= (N2), ЗАПОМНИТЬ (N1)
 15)  STOREN2:  MOV     R2,ODDBIG   ;(N2) > (N1), ЗАПОМНИТЬ (N2)
 16)  QUIT:     HALT                ;ОСТАНОВ
 17)            .END    START       ;КОНЕЦ ПРОГРАММЫ
```

Обратите внимание, что в строках 1, 2, 3, 11 и 15 мы воспользовались инструкцией пересылки MOV для инициализации регистров R1, R2, R0 соответственно и для сохранения результата в ячейке ODDBIG. В строках 4, 6, 9 и 13 мы использовали инструкцию логической операции BIT для проверки младшего бита и инструкцию манипуляции данными CMP для сравнения чисел N1 и N2. В строках 5, 7, 10 и 14 использована инструкция управления последовательностью выполнения BNE для условного изменения последовательности выполнения программы. В строках 8 и 12 для завершения выполнения программы используется инструкция безусловного ветвления BR. В строке 16 для того чтобы остановить работу машины, использована безоперандная инструкция HALT. В строках D–1, D–2, D–3 и 17 применены псевдоинструкции, предписывающие ассемблеру зарезервировать область памяти под названием ODDBIG для сохранения результата, поместить данные x и y в ячейки N1 и N2 и остановить процесс ассемблирования в строке 17.

В данном случае используется метод символьных адресов, поскольку мы не знаем точно, в какое место памяти системная программа загрузит (поместит) нашу программу. Если мы не предполагаем осуществлять построчную отладку программы (поиск ошибок), то фактически нам безразлично, где будет помещена программа. Позднее читатель узнает, что данные x и y могут быть помещены в программу несколькими способами. Но сейчас мы полагаем, что данные помещаются в ячейки N1 и N2 во время подготовки программы. Инструкция .END START в последней строке на самом деле является псевдоинструкцией; она отмечает конец текста программы и указывает, что стартовым адресом этой программы является метка START. Кроме того, поскольку нам известно, что ЦП игнорирует все операторы или комментарии, следующие за точкой с запятой, мы широко воспользовались этим и поместили сообщения, комментарии и замечания, чтобы они служили нам памяткой о созданной программе. Такая информация бывает очень полезной и для создателя, и для пользователя программы.

На рис. 3.5 показан исходный файл для этого примера с числовыми данными в ячейках N1 и N2. На рис. 3.6 представлен соответствующий файл листинга и даны результаты выполнения программы.

```
;ПОИСК НАИБОЛЬШЕГО НЕЧЕТНОГО ЧИСЛА
;
;
START:  MOV     N1,R1           ;(N1) -> R1
        MOV     N2,R2           ;(N2) -> R2
        MOV     #1,R0           ;МАСКА МЛАДШЕГО БИТА
EXAM1:  BIT     R0,R1           ;ПРОВЕРИТЬ МЛ. БИТ 1-ГО ЧИСЛА
        BNE     ODD2            ;ЕСЛИ N1 НЕЧЕТНОЕ, ПРОВЕРИТЬ N2
EXAM2:  BIT     R0,R2           ;ПРОВЕРИТЬ МЛ. БИТ 2-ГО ЧИСЛА
        BNE     STON2
        BR      QUIT
ODD2:   BIT     R0,R2
        BNE     COMPR           ;ОБА ЧИСЛА НЕЧЕТНЫЕ, СРАВНИТЬ ИХ
STON1:  MOV     R1,ODDBIG
        BR      QUIT
COMPR:  CMP     R1,R2
        BGE     STON1
STON2:  MOV     R2,ODDBIG
QUIT:   HALT
ODDBIG: .BLKW   1
N1:     .WORD   145
N2:     .WORD   111
        .END    START
```

Рис. 3.5. Исходный файл программы ODDBIG с числовым примером

```
1                       ;ПОИСК НАИБОЛЬШЕГО НЕЧЕТНОГО ЧИСЛА
2                       ;
3                       ;
4  000000 016701 000050 START: MOV    N1,R1           ;(N1) -> R1
5  000004 016702 000046         MOV    N2,R2           ;(N2) -> R2
6  000010 012700 000001         MOV    #1,R0           ;МАСКА МЛАДШЕГО БИТА
7  000014 030001        EXAM1: BIT    R0,R1           ;ПРОВЕРИТЬ МЛ. БИТ 1-ГО ЧИСЛА
8  000016 001003               BNE    ODD2            ;ЕСЛИ N1 НЕЧЕТНОЕ, ПРОВЕРИТЬ N2
9  000020 030002        EXAM2: BIT    R0,R2           ;ПРОВЕРИТЬ МЛ. БИТ 2-ГО ЧИСЛА
10 000022 001010               BNE    STON2
11 000024 000411               BR     QUIT
12 000026 030002        ODD2:  BIT    R0,R2
13 000030 001003               BNE    COMPR           ;ОБА ЧИСЛА НЕЧЕТНЫЕ, СРАВНИТЬ ИХ
14 000032 010167 000014 STON1: MOV    R1,ODDBIG
15 000036 000404               BR     QUIT
16 000040 020102        COMPR: CMP    R1,R2
17 000042 002373               BGE    STON1
18 000044 010267 000002 STON2: MOV    R2,ODDBIG
19 000050 000000        QUIT:  HALT
20 000052               ODDBIG: .BLKW  1
21 000054 000145        N1:    .WORD   145
22 000056 000111        N2:    .WORD   111
23    000000'                  .END    START

        РЕЗУЛЬТАТ: ЯЧЕЙКА ODDBIG СОДЕРЖИТ ЧИСЛО 145
```

Рис. 3.6. Файл листинга программы ODDBIG с результатом ее работы

3.7 УПРАЖНЕНИЯ

1. Преобразуйте приведенные ниже двоичные числа без знака в восьмеричное, шестнадцатеричное, десятичное, двоично-кодированное десятичное (в коде BCD) представления

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

2. Повторите упр. 1, но рассматривайте данные как двоичные числа в виде значения со знаком.

3. Используя приложение А, преобразуйте данные, приведенные в упражнении 1, в символы. Вы можете положить, что данные записаны в коде ASCII, а самый старший бит является битом нечетности.

4. Преобразуйте следующие десятичные числа соответственно в восьмеричное, двоичное и шестнадцатеричное представления:

64000	32000	16000	−128	−256
121	−12	8	20	−10

5. Преобразуйте приведенные в упр. 4 числа в представления в виде дополнения до единицы, дополнения до двух и дополнения до восьми соответственно.

6. Используйте дополнение до двух и дополнение до восьми для выполнения следующих арифметических операций:

а) $64000_{10} - 63210_{10} = ?$
б) $20_{10} - 128_{10} = ?$
в) $-12_{10} + 8_{10} = ?$
г) $-256_{10} - 128_{10} = ?$
д) $16000_{10} + 32121_{10} = ?$

7. Выразите результаты упр. 6 в формате 16-битового слова памяти ЭВМ PDP-11. Ваши ответы должны быть упакованы в 16-битовом двоичном формате; покажите эти ответы также в эквивалентном восьмеричном представлении. Вам нет нужды изменять их в представление в виде значения со знаком.

8. Напишите на языке ассемблера ЭВМ PDP-11 программу для поиска наименьшего числа в массиве из двух элементов, которые могут быть любыми целыми числами, и помещения этого числа в ячейку с меткой SMALL. Представьте свою работу в виде блок-схемы и в виде исходного файла.

9. Напишите на языке ассемблера ЭВМ PDP-11 программу, которая будет переупорядочивать массив из четырех символов в коде ASCII в алфавитном порядке и помещать его в память, начиная с ячейки с меткой DIRECT в качестве "каталога". Оформите свою работу в виде блок-схемы, а также в виде соответствующей исходной программы.

ГЛАВА 4

НАБОР ИНСТРУКЦИЙ ЭВМ PDP-11

4.1. ВВЕДЕНИЕ

В гл. 3 мы отмечали некоторые функциональные характеристики набора инструкций. Мы познакомились с двухоперандными инструкциями, например ADD A, B. Давайте теперь изучим эту инструкцию более внимательно. Вспомним, что мы безоговорочно согласились с тем, что эта инструкция занимает два слова памяти, и пока у нас нет никакой возможности даже задаться вопросом, почему это так. Мы закодировали эту инструкцию в 16-битовом слове следующим образом:

$b_{15} \ldots b_{12}$ $b_{11} \ldots b_6$ $b_5 \ldots b_0$
Код операции 1-й операнд 2-й операнд

Обратите внимание, что формат инструкции содержит три поля: 1) код операции (четыре бита); 2) первый или исходный операнд (шесть бит); 3) второй операнд или операнд назначения (шесть бит). В таком случае мы можем располагать только 2^4 = = 16 различными кодами операций, 2^6 = 64 ячейками для исходных операндов и 2^6 = = 64 ячейками для операндов назначения. Совершенно ясно, что ЭВМ с такими ограниченными возможностями вряд ли приемлема даже в качестве игрушки. Нарушить эти ограничения можно, если помещать в поля операндов не действительные адресные значения, а некоторую информацию, позволяющую определить адреса операндов, и потребовать от ЦП вычислять действительные или **эффективные адреса** с помощью определенного набора правил. Цена, которую придется за это заплатить, будет состоять в расширении цикла извлечения до двух шагов: 1) извлечение инструкции; 2) вычисление эффективных адресов операндов. Таким образом, полный цикл извлечения и выполнения инструкции будет следующим:

1. (PC) → РАП.
   (PC) + 2 → PC.
2. (РДП) → РИ.
3. Вычисление эффективного адреса исходного операнда.
4. Вычисление эффективного адреса операнда назначения.
5. Выполнение инструкции.

Правила, используемые для вычисления эффективных адресов, на разных машинах несколько различаются. Обычно чем более мощной является ЭВМ, тем сложнее эти правила. Как и следует ожидать, ЭВМ PDP-11 обладает достаточно сложными правилами адресации. Однако усилия, затраченные на изучение правил адресации для набора инструкций ЭВМ PDP-11, окажутся с лихвой вознагражденными при программировании на языке ассемблера этой машины. Совокупность таких правил называют режимами адресации ЭВМ PDP-11.

4.2. РЕЖИМЫ АДРЕСАЦИИ

Известны три общих класса методов адресации: 1) ссылка на регистр; 2) ссылка на адрес памяти; 3) смешанная ссылка на регистр и на адрес памяти. Для каждого класса существует ряд режимов адресации. В общем случае **ссылка на регистр** при выполнении программы действует быстрее, тогда как **ссылка на память** обеспечивает намного большее адресное пространство. В ЭВМ PDP-11 есть три различных формата инструкций, определения которых даются ниже.

Двухоперандная инструкция

Любая двухоперандная инструкция имеет следующий формат:

коп	**исх**	**назн**
$b_{15}b_{14}b_{13}b_{12}$	$b_{11}b_{10}b_9$ $b_8b_7b_6$	$b_5b_4b_3$ $b_2b_1b_0$

где

коп $\triangleq$ код операции;
исх $\triangleq$ исходный операнд;
$b_8b_7b_6$ $\triangleq$ адрес вовлеченного регистра Rn при n = 0, 1, . . . , 7;
$b_{11}b_{10}b_9$ $\triangleq$ код, определяющий тип режима адресации;
назн $\triangleq$ операнд назначения;
$b_2b_1b_0$ $\triangleq$ адрес вовлеченного регистра Rn;
$b_5b_4b_3$ $\triangleq$ код, определяющий тип режима адресации (символ $\triangleq$ означает "определено как").

Пример.

Мнемонический код: ADD R1, R2
Восьмеричный машинный код: 06 01 02

где

06 ≙ код операции инструкций сложения ADD;
01 ≙ исходный операнд; режим 0, вовлечен регистр R1;
02 ≙ операнд назначения; режим 0, вовлечен регистр R2.

Одноопе рандная инструкция

Каждая одноопе рандная инструкция имеет следующий формат:

коп	**назн**
$b_{15} \ldots b_6$	$b_5 b_4 b_3 \; b_2 b_1 b_0$

где

коп ≙ код операции;
назн ≙ операнд назначения;
$b_2 b_1 b_0$ ≙ адрес вовлеченного регистра Rn;
$b_5 b_4 b_3$ ≙ код, определяющий тип режима адресации.

Пример.

Мнемонический код: CLR R5
Восьмеричный машинный код: 050 05

где

050 ≙ операция очистки CLEAR;
05 ≙ режим 0, вовлечен регистр R5.

Безоперандная инструкция

Безоперандная инструкция имеет формат

коп
$b_{15} \ldots b_0$

Пример.

Мнемонический код: NOP; холостой цикл инструкции
Восьмеричный машинный код: 000240

РЕГИСТРОВЫЕ РЕЖИМЫ АДРЕСАЦИИ (ИСКЛЮЧАЯ РЕГИСТР R7)

Режим 0: прямая адресация. Под прямой адресацией мы подразумеваем, что поле операнда показывает действительный или эффективный адрес и никаких дальнейших вычислений не нужно.

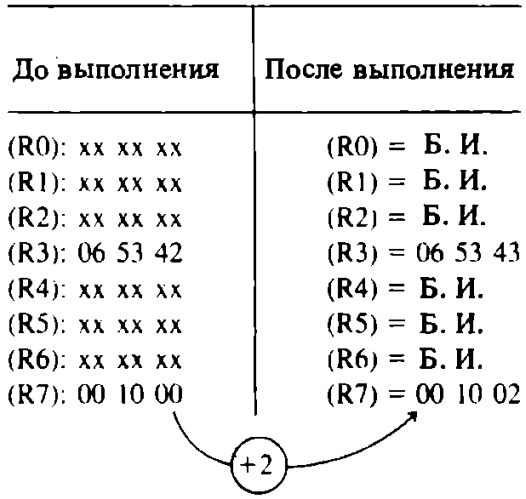
Пример 1.

```
Мнемонический код:        INC R3  ; (R3) + 1 → R3
Восьмеричный машинный код: 005203 ; ПОЛАГАЕМ, ЧТО ЭТА ИНСТРУКЦИЯ
                                  ; РАСПОЛАГАЕТСЯ ГДЕ-ТО В ОПЕРА-
                                  ;ТИВНОЙ ПАМЯТИ
```

где

```
INC = 0052
РЕЖИМ = 0
R3 = 3          ; ЗАМЕЧАНИЕ: "; " – ЭТО РАЗДЕЛИТЕЛЬ
                ; КОММЕНТАРИЯ ОТ ИНСТРУКЦИИ
```

Чтобы пояснить, что такое режим прямой адресации, давайте определим карту выполнения, показывающую состояние машины до и после выполнения инструкции:

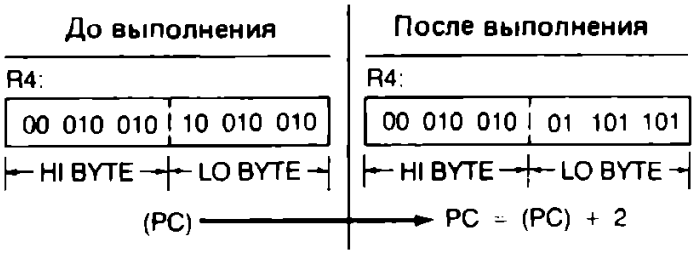


Примечание. XX XX XX ≙ любое восьмеричное значение; Б. И. ≙ без изменения; : ≙ подразумеваемое значение есть . . .

Для облегчения чтения мы здесь сгруппировали восьмеричные цифры в три пары и для удобства выбрали произвольные значения для регистров R3 и R7 (программного счетчика). Обратите внимание, что после выполнения инструкции содержимое регистра PC автоматически увеличивается на 2, а содержимое регистра R3 — на 1.

Пример 2.

Мнемонический код: COMB R4 ; ($\overline{\text{млад. байт R4}}$) → R4
Восьмеричный машинный код: 1051 ; РЕЖИМ 0, R*n* = R4



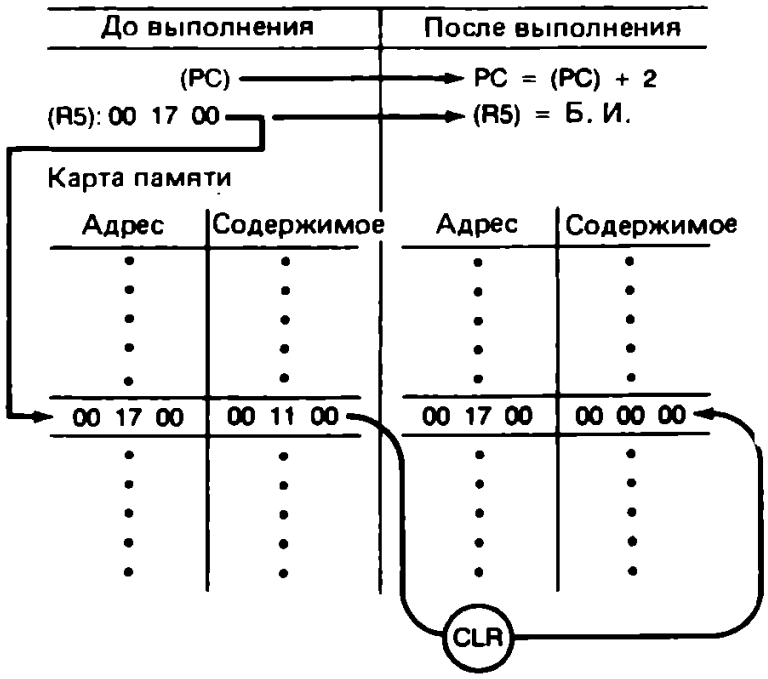
Это операция дополнения байта, которая требует специального внимания. Во-первых, обратите внимание, что при байтовой операции самый старший бит машинного кода всегда равен 1. Кроме того, показанная здесь инструкция подразумевает младший байт регистра R4, поэтому операция взятия дополнения выполняется только с младшим байтом регистра R4.

Режим 1: косвенная адресация. Косвенная адресация похожа на операцию "для передачи (кому-то)", выполняемую почтовой службой США, когда указанный адрес не является реальным адресом получателя, а является адресом его друга или родственника.

Пример.

Мнемонический код: CLR @R5 ; @R5 = (R5)
Восьмеричный машинный код: 0050 15 ; РЕЖИМ 1, R*n* = R5

Обратите внимание, что в качестве адреса операнда указан не R5, а @ R5, что эквивалентно записи (R5), т. е. запись @R5 равносильна указанию "для передачи R5", соответствующему нотации, принятой в почтовой службе США. Таким образом, реальным адресом операнда является содержимое регистра R5, а не сам адрес регистра R5. Выполняемое действие иллюстрирует карта выполнения:



Для этой иллюстрации ЦП определяет сначала, что применяется косвенный режим адресации через регистр R5. Поэтому он копирует содержимое регистра R5 в регистр РАП и очищает содержимое полученной ячейки.

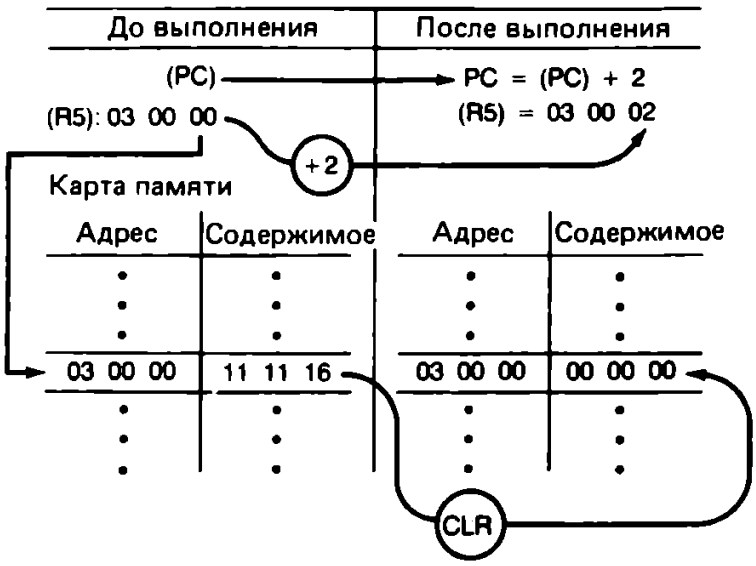
Режим 2: косвенная адресация с последующим автоувеличением. Это процесс косвенной адресации плюс последующее автоувеличение вовлеченного регистра. Содержимое вовлеченного регистра автоматически увеличивается на 2 (или на 1 при байтовой операции) после осуществления адресации. Символ "( ) +" обозначает косвенную адресацию и последующее увеличение. Следует отметить, что режимом "косвенной адресации с предварительным автоувеличением" ЭВМ PDP-11 не обладает.

Пример 1.

Машинный код: CLR (R5) +

Восьмеричный машинный код: 00 50 25 ; РЕЖИМ 2, R*n* = R5

Как показано в приведенной ниже карте выполнения, ЦП обнаруживает, что задан режим 2. Поэтому он копирует содержимое регистра R5 в регистр РАП, увеличивает содержимое слова памяти с адресом (R5) и, наконец, увеличивает содержимое регистра R5 на 2:

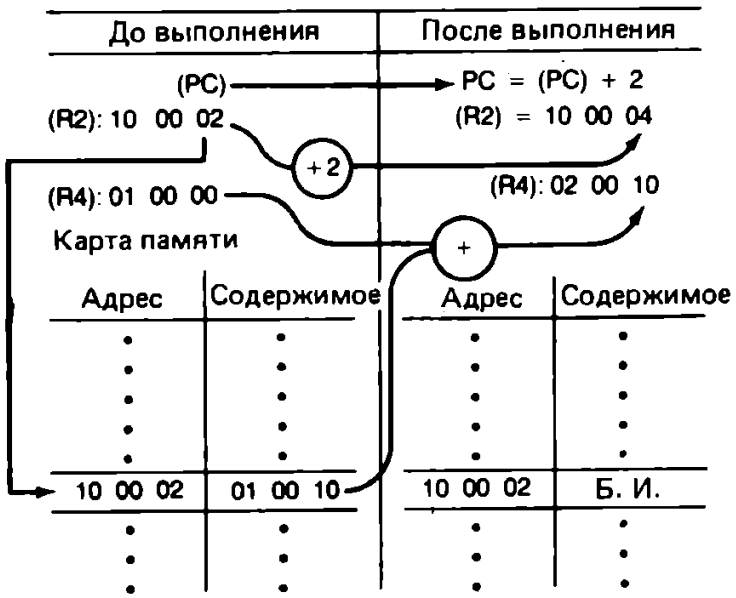


Пример 2.

Мнемонический код: ADD (R2) +, R4

Восьмеричный машинный код: 06 22 04

Обратите внимание, что в этой инструкции применяются режимы адресации 2 и 0. Поле исходного операнда имеет режим 2, а поле операнда назначения – режим 0. Карта выполнения будет иметь следующий вид:

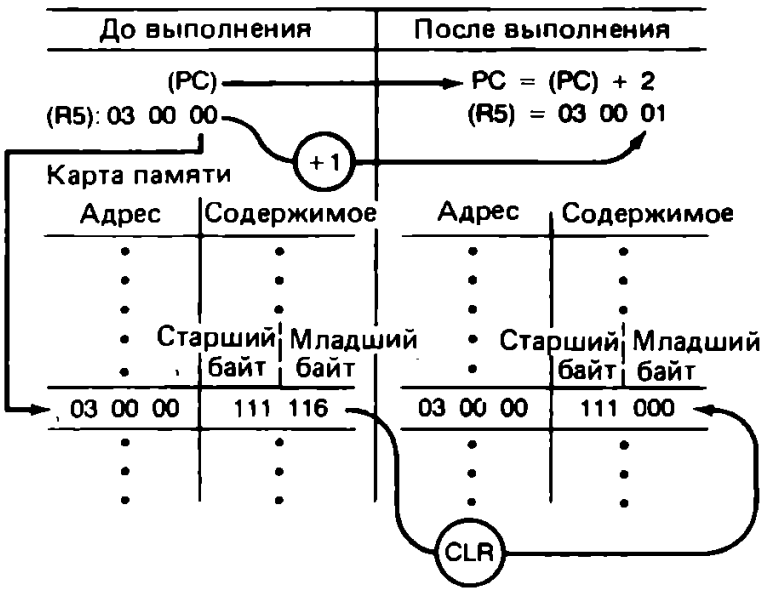


Пример 3 (операция над байтом). Этот пример показывает байтовую операцию с автоувеличением. Здесь после выполнения заданной операции содержимое вовлеченного регистра автоматически увеличивается на 1, а не на 2.

Мнемонический код: CLRB (R5) +

Восьмеричный машинный код: 1050 25

Карта выполнения:



Обратите внимание, что младший байт имеет адрес 03 00 00, а старший байт имеет адрес 03 00 01.

Режим 3: двойная косвенная адресация с последующим автоувеличением. В этом режиме присутствуют два уровня косвенной адресации. После выполнения адресации содержимое вовлеченного регистра автоматически увеличивается на 2 (или на 1 при байтовой операции). Если вновь воспользоваться аналогией с почтовой службой США, то

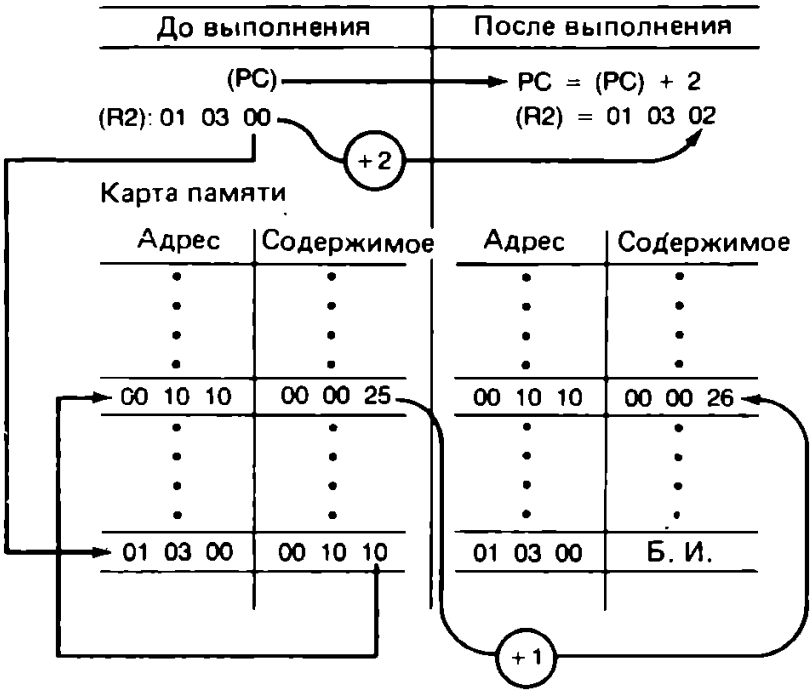
пришлось бы написать "для передачи . . . " дважды. Обратите внимание, что используется символ " @( ) +", поскольку имеет место двойная косвенная адресация плюс последующее автоувеличение.

Пример.

Мнемонический код: INC @(R2) +

Восьмеричный машинный код: 0052 32

Карта выполнения:



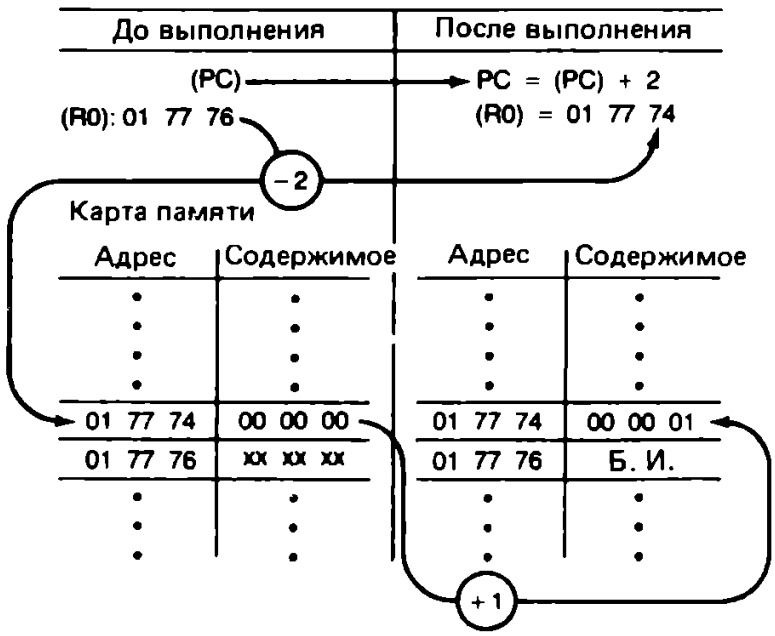
Режим 4: косвенная адресация с предварительным автоуменьшением. В этом режиме, прежде чем выполнять какую-либо операцию, осуществляется автоуменьшение. Содержимое вовлеченного регистра уменьшается на 2 (на 1 при байтовой операции) перед любым другим действием. Здесь используется символ "–( )". Следует отметить, что в ЭВМ PDP-11 режим адресации с последующим автоуменьшением не обеспечивается.

Пример 1.

Мнемонический код: INC –(R0)

Восьмеричный машинный код: 0052 40

Карта выполнения:

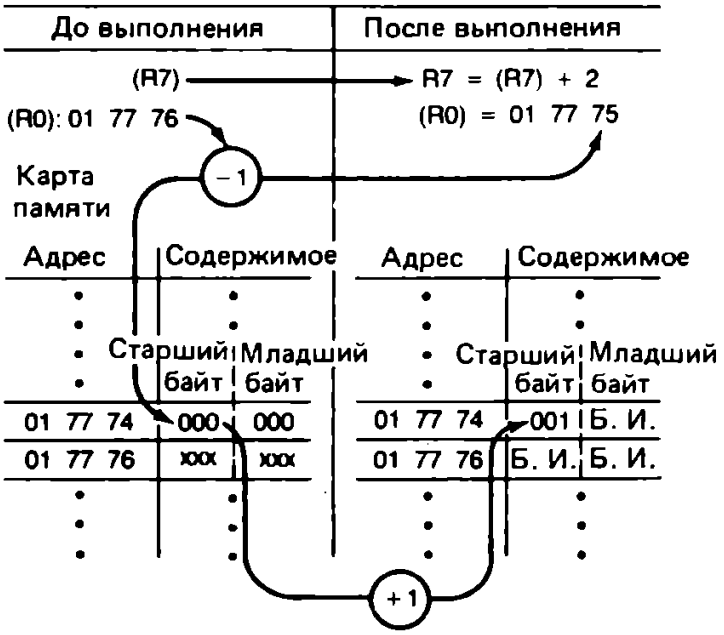


Пример 2 (операция над байтом).

Мнемонический код: INCB –(R0) ; БАЙТОВАЯ ОПЕРАЦИЯ

Восьмеричный машинный код: 1052 40

Карта выполнения:



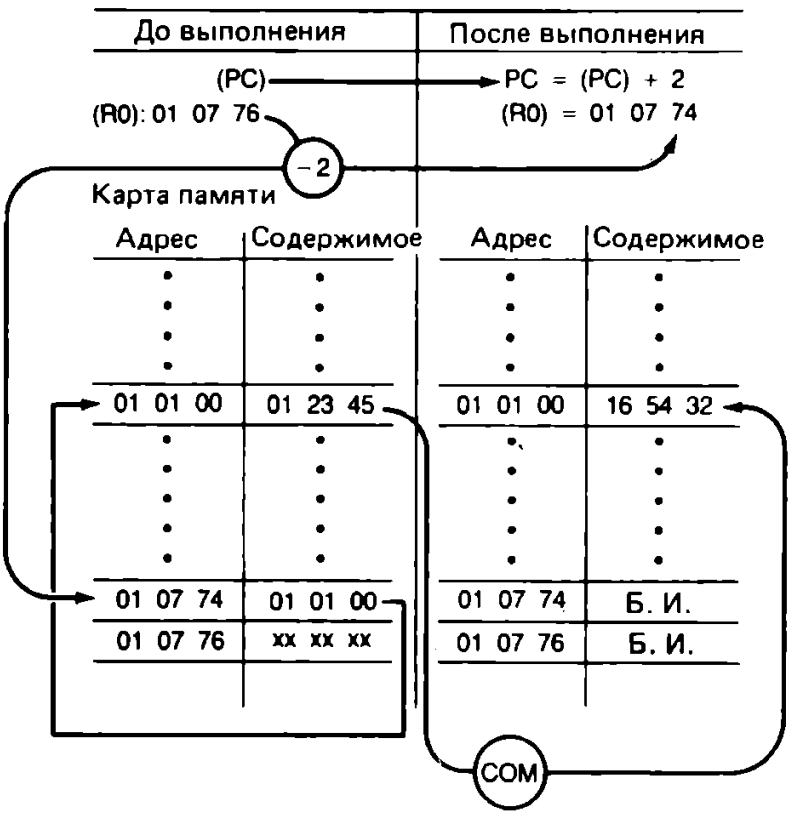
Режим 5: двойная косвенная адресация с предварительным автоуменьшением. В этом режиме осуществляется двухуровневая косвенная адресация с предварительным автоуменьшением. Сначала содержимое вовлеченного регистра уменьшается на 2 (на 1 при байтовой операции), затем следует двойная косвенная адресация. Поэтому используется нотация "@ –( )".

Пример.

Мнемонический код: COM @ –(R0)

Восьмеричный машинный код: 0051 50

Карта выполнения:



Режим 6: косвенная адресация с индексацией. В этом режиме инструкция занимает два слова памяти: в первом слове размещается код инструкции, во втором – значение индекса.

Пример 1.

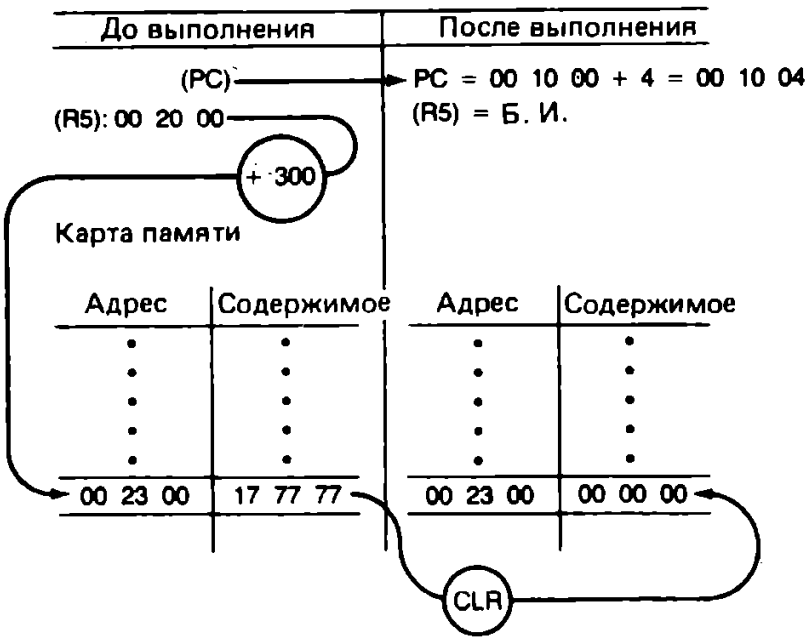
```
Мнемонический код:              CLR 300(R5)
Восьмеричный машинный код: 0050 65   ; В ПЕРВОМ СЛОВЕ ПАМЯТИ
                           0003 00   ; ВО ВТОРОМ СЛОВЕ ПАМЯТИ
```

В этом примере ЦП будет вычислять эффективный адрес операнда следующим образом:

эффективный адрес = ИНДЕКС + (R*n*) = 300 + (R5)
где ИНДЕКС – любое целое значение; R*n* – регистр с номером, *n* = 0, 1, . . . , 5.

И поскольку эта инструкция занимает два слова памяти, то программный счетчик (PC или R7) всегда будет увеличиваться на 4, т. е. скорректированное содержимое регистра (PC) равно (PC) + 4, где (PC) – это первоначальное содержимое программного счетчика. Положим для удобства, что рассматриваемая нами инструкция находится по адресу 1000. Тогда первоначальное содержимое PC есть 1000.

Карта выполнения будет такой:



Пример 2.

```
Мнемонический код:              ADD 300 (R2), 200 (R5)
Восьмеричный машинный код: 06 62 65      ; ЗАНИМАЕТ ТРИ СЛОВА
                           00 03 00
                           00 02 00
```

Поскольку эта инструкция имеет два операнда и в обоих операндах используется индексный режим адресации, она должна занимать три слова памяти: первое – для кода

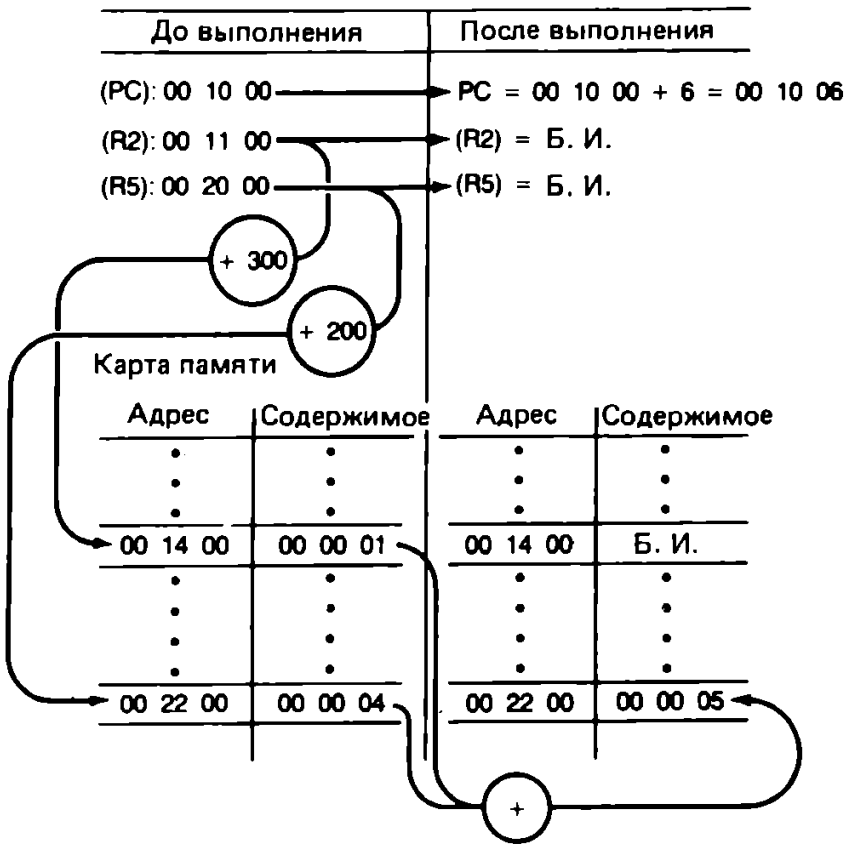
инструкции; второе – для индекса исходного операнда; третье – для индекса операнда назначения. Следовательно, содержимое регистра PC, всегда будет корректироваться следующим образом:

(PC) = (PC) + 6,

где (PC) – первоначальное содержимое. Эффективные адреса операндов будут следующими:

адрес исходного операнда = (R2) + 300;
адрес операнда назначения = (R5) + 200

Карта выполнения будет иметь вид:



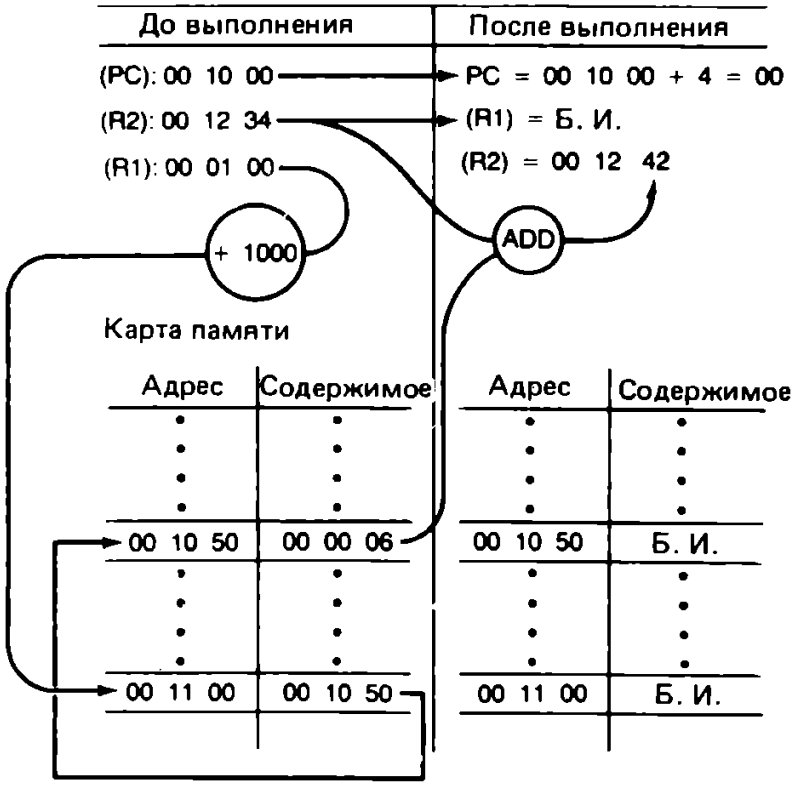
Режим 7: двойная косвенная адресация с индексацией. Подобно режимам с автоувеличением и автоуменьшением для индексного режима обеспечивается двойная косвенная адресация.

Пример.

```
Мнемонический код:          ADD @1000(R1), R2
Восьмеричный машинный код:  06  71  02   ;ЗАНИМАЕТ ДВА СЛОВА
                            00  10  00
```

Обратите внимание, что у исходного операнда режим адресации 7 и вовлечен регистр R1, поэтому соответствующий восьмеричный код – 71. У операнда назначения – режим 0 и вовлечен регистр R2, поэтому соответствующий восьмеричный код – 02.

Карта выполнения:



РЕЖИМЫ СО ССЫЛКОЙ НА ПАМЯТЬ (ИЛИ С ПРОГРАММНЫМ СЧЕТЧИКОМ)

В вычисления эффективного адреса рассмотренных до сих пор режимов адресации никогда прямо не вовлеклось содержимое регистра R7 (или PC). Мы представляем другую группу режимов адресации, в которой для определения эффективного адреса операнда (или операндов) используется только содержимое регистра PC. Поскольку значение программного счетчика постоянно корректируется, проследить за ним не так-то легко. Но на изучение методов адресации с вовлечением программного счетчика имеет смысл затратить некоторые усилия. Вспоминаем, что под информацию для адресных вычислений отводится шесть бит для каждого поля операнда: три левых бита – для задания режима адресации и три правых бита – для указания вовлеченного регистра. В данном случае мы используем только регистр PC (или R7), поэтому очевидно, что второй восьмеричной цифрой всегда будет 7.

Режим 2: непосредственный режим. Непосредственный режим обычно используется в поле исходного операнда и служит для оценки не эффективного адреса, но данных. Это бывает полезно, когда программисту хочется расположить некоторое постоянное число или значение вслед за рассматриваемой инструкцией. Например, если мы хотим инициализировать какой-то регистр или ячейку памяти определенной константой, то удобно воспользоваться этим режимом.

Пример 1.

```
Мнемонический код:              MOV  #1000, R1
Восьмеричный машинный код:  01   27   01     ;ЗАНИМАЕТ ДВА СЛОВА
                            00   10   00
```

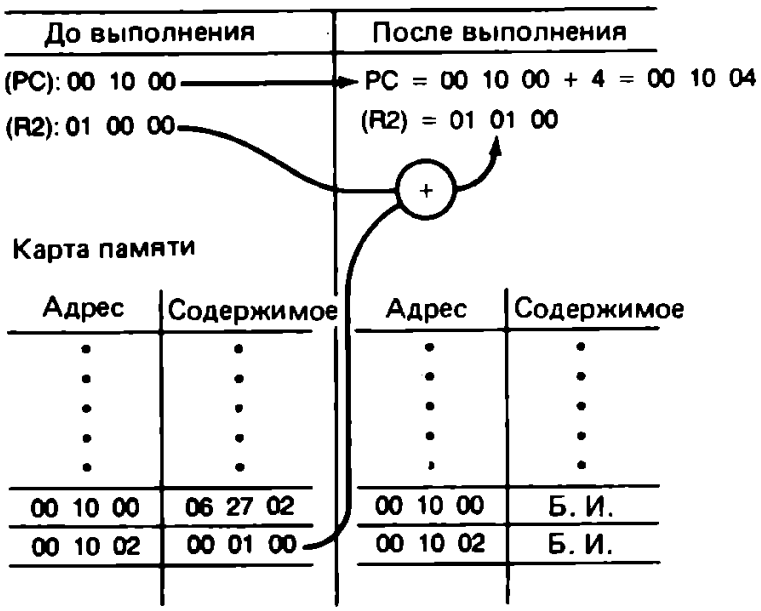
Обратите внимание на то, что эта инструкция занимает два слова, в первом из них содержится код инструкции, во втором – число или константа 1000. Для исходного операнда используется непосредственный режим (режим 2), а вовлеченным регистром является регистр R7. Поэтому появляется код 27. Для операнда назначения используется режим 0 с вовлеченным регистром R1, поэтому появляется код 01. Рассмотрим исходный операнд подробнее. Из записи инструкции видно, что регистр R7 в ней не присутствует. Почему же тогда мы говорим, что вовлечен регистр R7 (или регистр PC)? Вспоминаем, что при извлечении слова памяти машина автоматически увеличивает (PC)

на 2. В нашем примере, когда инструкция MOV окажется в ЦП, (PC) уже будет указывать на следующее слово, которое в данном случае содержит число 1000. При выполнении, распознав в поле исходного операнда код 27, ЦП просто "слепо" извлекает содержимое той ячейки, на которую указывает (PC), и помещает его в регистр R1. Программный счетчик в данном случае является "подпольным героем", обладающим "интеллектом", который позволяет ему знать, где нужно брать данные для выполняемого действия. И, конечно, когда ЦП получает число 1000, содержимое регистра PC опять автоматически увеличивается на 2. Карта выполнения имеет вид:

Пример 2.

Мнемонический код: ADD #100, R2
Восьмеричный машинный код: 06 27 02
00 01 00

Карта выполнения:

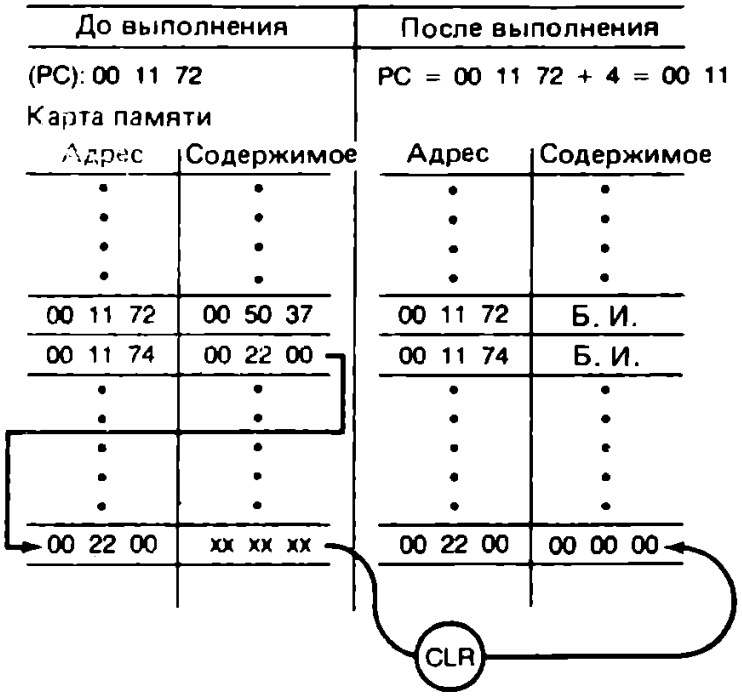


Режим 3: абсолютная адресация. Режим абсолютной адресации также требует два слова памяти: в первом слове располагается код инструкции, а во втором – данные или число. Однако данные во втором слове являются абсолютным адресом операнда, поэтому в вычислении исполнительного адреса нет необходимости.

Пример.

Мнемонический код: CLR @ #2200
Восьмеричный машинный код: 0050 37
0022 00

Карта выполнения:



Режим 6: относительная адресация. Относительная адресация обычно используется совместно с символьным адресом. Этот режим – один из наиболее важных в любой ЭВМ. Машина, в которой не обеспечен режим относительной адресации, считается примитивной. Однако что такое относительная адресация, поначалу понимается с трудом. Рассмотрим простой пример. Человек, заблудившийся на улице, может обратиться с вопросом к прохожему: "Где находится дом № 760?". Возможны такие ответы: "После четвертого магазина по ходу вашего движения" или "Пройдите назад семь зданий и окажетесь возле нужного" и т. д. Обратите внимание, что адрес № 760 является базовым адресом, тогда как "четыре магазина вперед" (+4) или "семь зданий назад от того места, где вы находитесь" (–7) – это относительный адрес, т. е. расстояние от того места, где вы находитесь, до места назначения. Если рассматривать скорректированное содержимое регистра PC как "место, где вы находитесь" и знать место назначения, то легко можно вычислить относительное расстояние, особенно с помощью АЛУ.

Инструкция, использующая относительную адресацию, также занимает два слова. Однако данные или значения во втором слове – это "относительное расстояние". Таким образом, эффективный адрес может быть вычислен следующим образом:

Эффективный адрес = $(PC)_{скор} + \pm$(относительное расстояние)

Пример 1.

Мнемонический код: INC A
Восьмеричный машинный код: 0052 67
OP

где A – символьный адрес операнда назначения, OP Δ относительное расстояние в виде дополнения до двух.

Для вычисления эффективного адреса нам, чтобы мы смогли определить скорректированное содержимое регистра PC, прежде всего надо знать адрес рассматриваемой инструкции. Если во втором слове задано относительное расстояние, то относительный адрес легко определить. Проблема в том, кто будет задавать значение относительного

расстояния во втором слове инструкции? Теоретически любой программист должен быть в состоянии вычислять значение OP, но на практике никому не хочется делать эту "черную работу". Логично, чтобы эту работу делал ассемблер, входящий в операционную систему. Во всяком случае, ассемблеру приходится транслировать мнемонический код в машинный код, поэтому ему можно поручить и определение значения OP. Ниже приведены примеры, показывающие, как ассемблер вычисляет значение OP при различных условиях.

Случай 1. Значение адресной метки A больше, чем (PC), т. е. адрес назначения находится после рассматриваемой инструкцией. Пусть символ ":" означает "полагаемое значение есть". Тогда мы можем записать.

```
(PC):  00    10    00        ;АДРЕС ИНСТРУКЦИИ
   A:  00    11    00        ;ПОЛАГАЕМ, ЧТО А ВПЕРЕДИ
 (A):  00    10    07        ;ПОЛАГАЕМОЕ ЗНАЧЕНИЕ ДЛЯ СОДЕРЖИМОГО А
```

Отсюда

$$(PC)_{скор} = (PC) + 4 = 1000 + 4 = 1004,$$

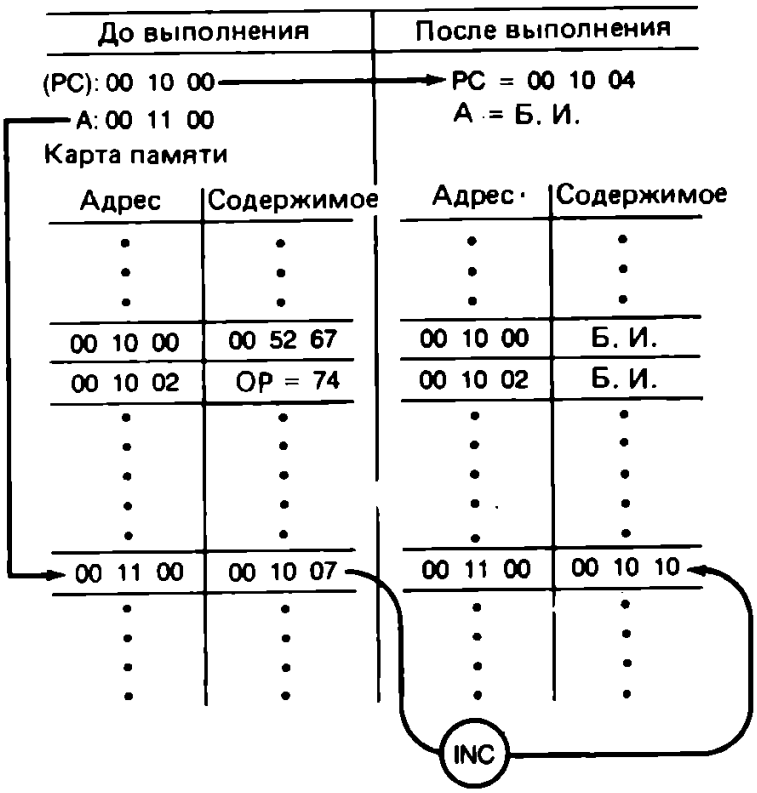
где (PC) $\triangleq$ первоначальное содержимое регистра PC или адрес инструкции

$(PC)_{скор}$ $\triangleq$ скорректированное содержимое регистра PC

и

$$OP = A - (PC)_{скор} = 1100 - 1004 = 74.$$

Карта выполнения при этом имеет вид:



Эффективный адрес операнда в этом случае вычисляется в процессе цикла извлечения следующим образом:

эффективный адрес = $(PC)_{скор}$ + OP = 1004 + 74 = 1100.

Случай 2. Значение A меньше, чем (PC), т. е. адрес назначения расположен до адреса инструкции:

(PC) : 1000

A : 0120
(A) : 1007

Тогда ОР будет отрицательным числом, эквивалентное значение дополнения до двух которого может быть вычислено следующим образом:

$(PC)_{скор}$ = 1000 + 4 = 1004;
OP = 120 – 1004.

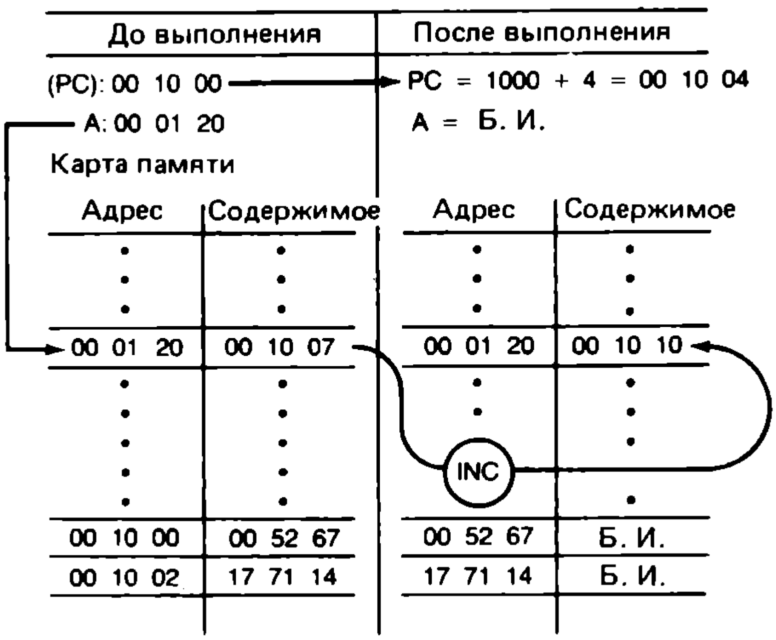
Преобразуем величину со знаком $(PC)_{скор}$ в дополнение до восьми:

```
B/З:   [1]0 10 04 ─┐  7̄  ┐
 7̄:       7 67 73 <┘     │   8̄
              +1         │
 8̄:    [1]7 67 74 <──────┘
```

Тогда ОР вычисляется так:

```
     00 01 20
+ [1]7 67 74
  [1]7 71 14
```

Карта выполнения:



Эффективный адрес операнда вычисляется во время цикла извлечения следующим образом:

эффективный адрес = $(PC)_{скор}$ + OP = 1004 + 17 71 14 = 00 01 20,

что согласуется с адресом назначения А.

Пример 2. Этот пример показывает, как можно ассемблировать вручную программу, начинающуюся с ячейки 000000.

```
START:  MOV     A,B
        HALT
    A:  .WORD   6
    B:  .BLKW   1
        .END    START
```

Определим сначала, сколько слов памяти необходимо для этой программы:

1. Первая инструкция имеет два операнда, и оба операнда используют режим относительное адресации; поэтому для этой инструкции нужно три слова.

2. Остальные инструкции – одноадресные.

3. Поскольку START = 000000, то A = 10, B = 12. Теперь определим OP для A и B соответственно. OP для A равно $A - (PC)_{скор}$, где A = 10, а коррекция равна (PC) + + 4 = 0 + 4 = 4. Следовательно, поскольку при извлечении центральным процессором (A) содержимое регистра PC увеличивается и указывает уже на следующее слово, адрес которого 4, то OP для A равно $A - (PC)_{скор} = 10 - 4 = 4$. Аналогично OP для B равно $B - (PC)_{скор} = B - [(PC) + 6] = 12 - [0 + 6] = 4$. Найдя необходимые машинные коды в наборе инструкций и обратившись к файлу листинга этой программы (названному "пример2.LST"), мы можем вручную ассемблировать программу следующим образом:

Карта памяти		Исходный код	
Адрес	Содержимое	Адрес	Мнемоническая инструкция
00 00 00	01 67 67	START	MOV A, B
00 00 02	00 00 04		
00 00 04	00 00 04		
00 00 06	00 00 00		HALT
00 00 10	00 00 06	A	.WORD 6
00 00 12	00 00 00	B	.BLKW 1

Пример 3. Этот пример показывает влияние различных адресов операндов назначения на вычисление значений относительного расстояния. Давайте поместим данные перед первой инструкцией следующим образом:

```
A:      .WORD   6
B:      .BLKW   1
START:  MOV     A,B
        HALT
        .END    START
```

где A = 0, B = 2, (A) = 6 (B) = 0 START = 4. Тогда OP для A и B можно вычислить так:

$A - (PC)_{скор} = 0 - [4 + 4] = 0 - 10 = -10(B/3) = 17\ 77\ 70\ (\bar{8})$

Аналогично OP для B равно 177770 ($\bar{8}$). Соответственно файл листинга будет следующим:

Карта памяти		Исходный код	
Адрес	Содержимое	Адрес	Содержимое
00 00 00	00 00 06	A	.WORD 6
00 00 02	00 00 00	B	.BLKW 1
00 00 04	01 67 67	START	MOV A, B
00 00 06	17 77 70		
00 00 10	17 77 70		
00 00 12	00 00 00		HALT

Интересно отметить, что и для примера 2, и для примера 3, если положить начальный адрес программы равным 1172 вместо 0, то вычисление значений OP производилось бы так же, как показано выше. Для примеров 2 и 3 мы имели бы следующее: OP для A равно 1202 – (1172 + 4) = 4, OP для B равно 1204 – (1172 + 6) = 4 (для примера 2); OP для A равно 1172 – (1176 + 4) = 4 – –10(B/3) = 17 77 70 ($\bar{8}$), OP для B= 1174 – – (1176 + 6) = –10(B/3) = 17 77 70 ($\bar{8}$) (для примера 3).

Обратите внимание, что значение OP не зависит от того, в каком месте памяти начинается программа. Особенно это удобно для процессов построения задания или компоновки (с помощью утилиты ТКВ) при разработке программы. Режим относительной адресации удобен также тем, что программисту не надо беспокоиться об изменении выражения адреса операнда, если между занимающей этот адрес инструкцией и операндом назначения требуется удалить или вставить несколько инструкций. Программист по-прежнему может использовать тот же символьный адрес, оставив ассемблеру заботу о подстройке значений OP. Следующие два примера иллюстрируют это свойство.

Пример 4. Используем режим абсолютной адресации для реализации программы, показанной в примере 3.

Числовой адрес	Исходный код
00 00 00	.WORD 6
00 00 02	BLKW 1
00 00 04	MOV @#0, @#2
00 00 06	
00 00 10	
00 00 12	HALT

Эта программа будет выполнять свою функцию до тех пор, пока начальный адрес остается неизменным. Если же мы загрузим программу, начиная, например, с адреса 1172, то должны будем изменить операнды инструкции MOV : @≠ 0, @≠ 2 заменим на @#1172, @#1174, ибо в противном случае программа работать не будет.

Пример 5. Воспользуемся примером 2 и предположим, что после операции MOV мы хотим разделить на 2 содержимое ячейки B.

```
START:  MOV     A,B
        ASR     B               ;АРИФМЕТИЧЕСКИЙ СДВИГ ВПРАВО
                                ;ДАЕТ ЭФФЕКТ ДЕЛЕНИЯ НА 2
        HALT
A:      .WORD   6
B:      .BLKW   1
        .END    START
```

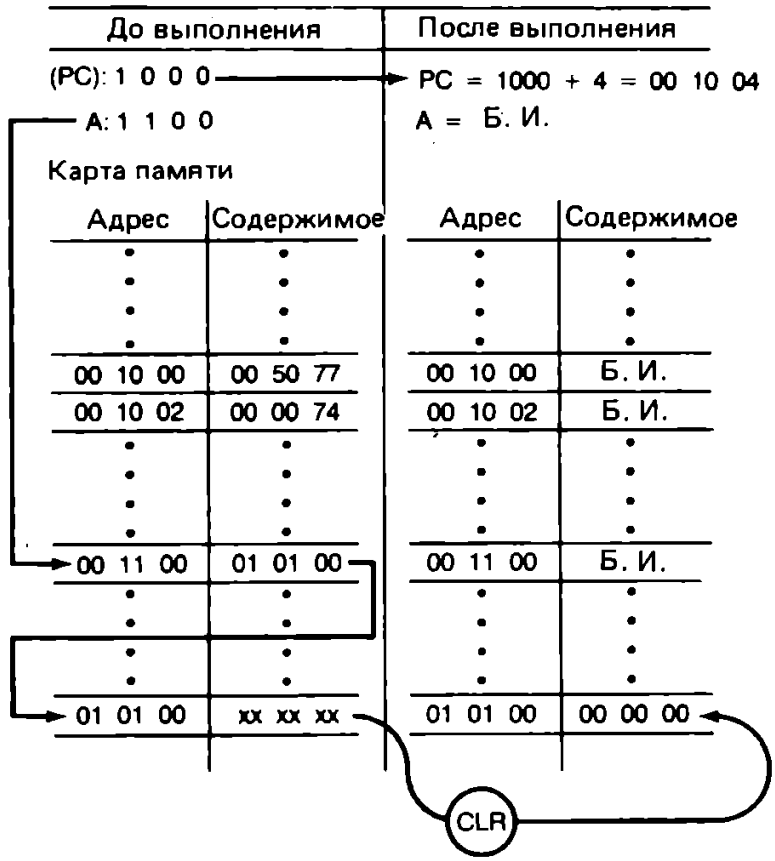
Инструкция ASR B займет два слова, и поэтому адресные значения A и B продвинутся дальше на 4 байта. Значение адресной метки A изменится с 00 00 10 на 00 00 14, а значение адресной метки B – с 00 00 12 на 00 00 16. Поскольку здесь мы используем режим относительной адресации, то беспокоиться не о чем. Мы просто реассемблируем программу с помощью ассемблера, который поместит правильные значения OP в нужные места. При использовании режима абсолютной адресации мы вынуждены были бы изменить выражение операндов инструкции MOV.

Режим 7: относительная косвенная адресация. Этот режим есть не что иное, как расширение режима 6 путем добавления одного уровня косвенной адресации операнда.

Пример.

Мнемонический код: CLR @A
Восьмеричный машинный код: 0050 77
OP

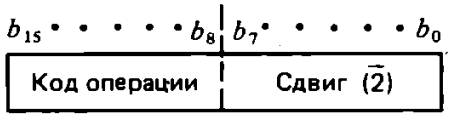
Карта выполнения:



Примечание: OP = A – СКОР (PC) = 1100 – (1000 + 4) = 74

АДРЕСАЦИЯ ИНСТРУКЦИЙ ВЕТВЛЕНИЯ

Инструкция ветвления – это одноадресная инструкция, обычно использующая режим относительной адресации, но занимающая одно слово памяти. Старший байт этого слова представляет поле кода операции, а младший – поле сдвига:



Поле сдвига – это половина относительного расстояния между скорректированным (PC) и адресатом. Значение сдвига задается дополнением до двух. Положительное значение сдвига подразумевает, что скорректированное содержимое регистра PC находится "дальше", отрицательное значение подразумевает, что это значение располагается "прежде": адрес адресата = $(PC)_{скор}$ + OP = $(PC)_{скор}$ + 2 (СДВИГ). Поскольку самый старший бит младшего байта, b_7, является знаковым битом, значение сдвига имеет ограниченный диапазон:

$$-128_{10} = 200_8 \leqslant \text{СДВИГ} < 177_8 = 127_{10},$$

а OP имеет диапазон от -256_{10} до 254_{10}. Очень важно, чтобы программисту было известно об этом ограничении. В противном случае ассемблер может выдавать туманные сообщения об ошибках, не способствующие ясному пониманию программистом происходящего.

Есть два класса инструкций ветвления: условного ветвления и безусловного ветвления. Однако вычисление их значений сдвига выполняется идентично. Хотя ассемблер

ответствен за определение значений сдвига, читателю необходимо изучить, каким образом определяются эти значения.

Пример 1.

Мнемонический код: BR A ; БЕЗУСЛОВНЫЙ ПЕРЕХОД К A
Восьмеричный машинный код: 001 СДВИГ

Положим, что текущее содержимое регистра PC, указывающее на интересующую нас инструкцию, равно 1000, а значение A (но не содержимое A!) равно 1100. Тогда

$$OP = A - (PC)_{скор},$$

что идентично вычислениям OP для режима относительной адресации. Однако, поскольку инструкция ветвления занимает только одно слово, $(PC)_{скор} = (PC) + 2 = 1000 + 2 = 1002$ и $OP = 1100 - 1002 = 0076$, но СДВИГ = 1/2 (OP) = 1/2 (0076) = 0037. Вспоминаем, что деление на 2 равносильно арифметическому сдвигу вправо (ASR) на один бит. Поэтому значение интересующего нас сдвига легко получить выполнением операции ASR над OP. Поскольку $0076_8 = 000\ 000\ 111\ 110_2$, то в результате инструкции ASR $000\ 000\ 111\ 110_2$ получим 037_8.

Вставив результат в поле сдвига инструкции, мы получим полный код:

	Старший байт ◄——► Младший байт	
Двоичный код:	00 000 001	00 011 111
В байтах:	0 0 1	0 3 7
В слове:	0 0 0	4 3 7

Очень важно, чтобы студент ясно осознал неожиданное различие представлений в байтах и словах, возникающее из-за способов группировки двоичных бит.

Поскольку чаще всего нас будет интересовать восьмеричное представление полного слова, мнемонический операционный код инструкций ветвления задается не в виде старшего байта, а в форме полного слова с полем сдвига, заполненным нулями. Таким образом полный машинный код может быть получен арифметическим прибавлением к этому коду значения сдвига. В руководствах для пользователей ЭВМ PDP-11 такой код инструкции ветвления называют базовым кодом. Например, в нашем случае базовый код инструкции BRANCH – 00 04 00, хотя код операции в старшем байте инструкции равен 0 0 1. Чтобы получить полный код инструкции, мы просто прибавляем значение сдвига к базовому коду. Полное слово, содержащее код инструкции BR A в нашем примере, равно 00 04 00 + 00 00 37 = 00 04 37.

Пример 2. В этом примере мы рассмотрим случай, когда A располагается прежде скорректированного содержимого регистра PC:

(PC): 1000
A : 700

Полное слово, содержащее код инструкции BR A, мы можем определить следующим образом:

$OP = A - (PC)_{скор} = 00\ 07\ 00 - 00\ 1002 = 00\ 07\ 00 + 17\ 67\ 76 = 17\ 76\ 76$
СДВИГ = 1/2 (OP) = 17 77 37 (для 16-битового представления) = 337 (для 8-битового представления)
Полное кодовое слово = (базовый код) + (сдвиг) = 00 04 00 + 337 = 00 07 37.

Пример 3. Этот пример показывает, как можно написать программу для копирования некоторого массива данных в определенную область памяти. В программе применяются различные режимы адресации.

```
1) A:      .WORD     0,1,2,3,4,5,6,7
2) B:      .BLKW     10
3) START:  MOV       #10,R0          ;R0 - СЧЕТЧИК
4)         MOV       #A,R1           ;R1 - УКАЗАТЕЛЬ ИСХ. МАССИВА ДАННЫХ
5)         MOV       #B,R2           ;R2 - МАССИВ ДАННЫХ НАЗНАЧЕНИЯ
6) LOOP:   MOV       (R1)+,(R2)+     ;ПЕРЕСЛАТЬ ЭЛЕМЕНТ ДАННЫХ
7)         DEC       R0              ;УМЕНЬШИТЬ СЧЕТЧИК
8)         BNE       LOOP
9)         HALT
           .END      START
```

Здесь инструкции 1 и 2 – это директивы данных или псевдоинструкции. Каждая имеет по 8 элементов с собственными начальными адресами A и B соответственно. В инструкциях 3 – 5 используется режим непосредственной адресации; эти инструкции инициализируют соответственно регистры R0, R1 и R2. Регистр R0 служит счетчиком, который показывает, завершена ли транспортировка данных. Регистры R1 и R2 используются в качестве указателей данных, т. е. значение A – начальный адрес исходного массива данных, пересылается в регистр R1, а значение B – начальный адрес массива данных назначения, помещается в регистр R2. Инструкция 6, в которой используется режим адресации с автоувеличением, копирует данные из массива A в массив B по одному элементу данных за один раз. Каждый раз, когда пересылается элемент данных, указатели регистров R1 и R2 увеличиваются на 2 и указывают на следующий элемент. Вслед за каждой пересылкой данных инструкция 7 уменьшает содержимое регистра R0 на единицу. В инструкции 8, осуществляющей ветвление к началу цикла, если содержимое регистра (R0) не равно нулю, используется инструкция условного ветвления и режим относительной адресации. Эта инструкция проверяет результат выполнения инструкции уменьшения DEC, просматривая флаг Z в PSW. Если результат выполнения инструкции уменьшения ненулевой (т. е. Z = 0), то транспортировка данных еще не завершена. Следовательно, происходит ветвление и выполнение продолжается с символьного адреса LOOP. Пересылка данных повторяется до тех пор, пока содержимое регистра R0 не станет равным нулю. При этом условии программа переходит к инстуркции 9, и выполнение программы останавливается. На рис. 4.1 показан файл листинга этой программы. Студенту следует внимательно просмотреть восьмеричные коды для всех использованных режимов адресации и проверить значения относительного расстояния для режима относительной адресации и значение сдвига для инструкции ветвления.

Адресация инструкции SOB. Эта инструкция особенно полезна для написания циклов в программах, она – двухадресная:

SOB Rn, назн

где Rn – регистр номер n, n = 0, 1, . . . , 5; "назн" – адрес назначения, который должен быть прежде самой инструкции или значение которого должно быть меньше чем $(PC)_{скор}$

Инструкция SOB действует следующим образом: из регистра Rn вычитается единица; если содержимое не равно нулю, то осуществляется ветвление к адресу назначения; в противном случае выполняется следующая инструкция. Она имеет формат

СДВИГ – поле = $b_0 \ldots b_5$, Rn – поле = $b_6 b_7 b_8$
код операции – поле = $b_0 \ldots b_{15}$ = 077.

Обратите внимание, что в этой инструкции значение сдвига без знака; поскольку ветвление всегда осуществляется назад, то максимальный диапазон равен $2^6 - 1 = 63$. Таким образом, максимальное значение OP будет 126 байт.

Пример. Инструкция SOB может использоваться для замены инструкций 7 и 8 в примере 3 предыдущего подраздела. Получаемый при этом файл листинга показан на рис. 4.2.

Ограничения инструкций ветвления. Поскольку инструкция ветвления требует только одного слова памяти, то диапазон (или дистанция) ветвления ограничивается 8-битовым значением сдвига. Поэтому перед тем, как использовать инструкцию вет-

```
1                       ;В ЭТОМ ПРИМЕРЕ ДЛЯ УПРАВЛЕНИЯ ЦИКЛОМ
2                       ;ПРИМЕНЯЕТСЯ ИНСТРУКЦИЯ BNE
3                       ;
4   000000 000000  A:     .WORD   0,1,2,3,4,5,6,7
    000002 000001
    000004 000002
    000006 000003
    000010 000004
    000012 000005
    000014 000006
    000016 000007
5   000020         B:     .BLKW   10
6   000040 012700  START: MOV     #10,R0          ;R0 - СЧЕТЧИК
           000010
7   000044 012701         MOV     #A,R1           ;R1 - УКАЗАТЕЛЬ МАССИВА A
           000000'
8   000050 012702         MOV     #B,R2           ;R2 - УКАЗАТЕЛЬ МАССИВА B
           000020'
9   000054 012122  LOOP:  MOV     (R1)+,(R2)+     ;СКОПИРОВАТЬ МАССИВ A
10  000056 005300         DEC     R0              ;В МАССИВ B
11  000060 001375         BNE     LOOP
12  000062 000000         HALT
13         000040'        .END    START
```

Рис. 4.1. Копирование массива данных с помощью цикла с ветвлением

```
 1                      ;В ЭТОМ ПРИМЕРЕ ДЛЯ УПРАВЛЕНИЯ ЦИКЛОМ
 2                      ;ПРИМЕНЯЕТСЯ ИНСТРУКЦИЯ SOB
 3                      ;
 4  000000 000000  A:     .WORD   0,1,2,3,4,5,6,7
    000002 000001
    000004 000002
    000006 000003
    000010 000004
    000012 000005
    000014 000006
    000016 000007
 5  000020         B:     .BLKW   10
 6  000040 012700  START: MOV     #10,R0          ;R0 - СЧЕТЧИК
           000010
 7  000044 012701         MOV     #A,R1           ;R1 - УКАЗАТЕЛЬ МАССИВА A
           000000'
 8  000050 012702         MOV     #B,R2           ;R2 - УКАЗАТЕЛЬ МАССИВА B
           000020'
 9  000054 012122  LOOP:  MOV     (R1)+,(R2)+     ;СКОПИРОВАТЬ МАССИВ A
10  000056 077002         SOB     R0,LOOP         ;В МАССИВ B
11  000060 000000         HALT
12  000040'               .END    START
```

Рис. 4.2. Копирование массива данных при использовании инструкции SOB

```
        1                       ;ЭТОТ ПРИМЕР ПОКАЗЫВАЕТ
        2                       ;ОГРАНИЧЕНИЕ ИНСТРУКЦИИ ВЕТВЛЕНИЯ
        3                       ;
        4  000000               BLOCK1: .BLKW   200
        5  000400  000000       A:      HALT            ;A СЛИШКОМ ДАЛЕКО ОТ B
        6  000402               BLOCK2: .BLKW   177
A       7  001000  000777       B:      BR      A
        8  001002               BLOCK3: .BLKW   37
        9  001100  000737       START:  BR      B
       10          001100'             .END    START
```

Рис. 4.3. Ограничение инструкции ветвления

вления, мы должны быть уверены, что расстояние, на которое осуществляется ветвление, не выходит за пределы ограничений. На рис. 4.3 показан файл листинга программы, иллюстрирующий эту проблему. Обратите внимание, что расстояние от ячейки A до скорректированного (PC) для инструкции в ячейке B равно 1002 – 400 = 402, а это больше, чем предел ветвления назад, равный 400. Поэтому ассемблер обнаружил ошибку и классифицировал ее как "ошибка A", которая в руководствах для пользователей ЭВМ PDP-11 определяется так: "неверный адрес в инструкции или ошибка перемещения". Однако же расстояние между ячейками START: и B: находится в пределах ограничений, поэтому в строке 9 листинга никаких проблем не обнаружено. Чтобы избежать подобного рода проблем, следует вместо инструкции BPANCH использовать инструкцию JUMP. Использование инструкции JUMP описано в следующем подразделе.

Адресация инструкции JMP. Эта инструкция аналогична знакомой нам инструкции безусловного ветвления, за исключением того, что она занимает два слова памяти, причем во втором слове сохраняется значение относительного расстояния. Поэтому она имеет диапазон от –32 тысяч байт до +32 тысяч байт (или ±16 тысяч слов). Если инструкция безусловного ветвления работать не может, используйте инструкцию JMP.

Пример.

Мнемонический код: JMP A
Восьмеричный машинный код: 0001 назн
OP

где поле "назн" определяет режим адресации и вовлеченный регистр, а OP по определению есть значение относительного расстояния.

Пример.

```
      .
      .
      .
    JMP  A
      .
      .
      .
A:  HALT
      .
      .
      .
    .END
```

При выполнении инструкции перехода в регистр PC загружается эффективный адрес ячейки A, после чего программа начинает выполнять инструкцию, размещенную по адресу A.

Адресация стековой памяти с помощью регистра R6. По мере накопления опыта программирования мы все более и более постигаем доступность регистров общего назначения и у нас появляется потребность иметь большее число регистров. Проводя вычисления на бумаге, мы нередко пользуемся черновиками для выполнения каких-то побочных операций, чтобы основные вычисления оставались опрятными. Стековая память в ЭВМ напоминает такой "черновик". Она может использоваться для запоминания временных данных. Стековую память можно представлять себе как расширение числа регистров общего назначения с той лишь разницей, что регистры находятся в ЦП,

тогда как стек располагается в оперативной памяти. Посмотрим, как работает стековая память.

В оперативной памяти мы резервируем некоторую область, скажем, в 256 слов, которую используем исключительно для целей кратковременного характера. Для этой специальной области в качестве указателя назначается регистр R6. Такая область называется стековой памятью (или кэш-памятью). Регистр R6 называется **указателем стека** (SP). Функции указателя стека сходны с функциями программного счетчика PC (R7), однако первый указывает на стековую память, а последний является указателем общей области памяти. Типичная система PDP-11 имеет объем памяти 64 Кбайта, и эта память адресуется с помощью 16-битового регистра. Область памяти с адреса 00 00 00 до некоторого адресного значения, например 00 11 72, резервируется для системного программного обеспечения. Остальная оперативная память отводится для программ.

Под стековую память обычно назначается область, расположенная вниз от некоторого адреса, в нашем случае от 00 11 72, и имеющая объем 256 байт. Соответственно система присваивает указателю стека начальное значение 1172 и затем ведет указатель стека в обратном направлении, тогда как содержимое программного счетчика PC начинается с адреса 1172 и корректируется по направлению вперед. Когда оперативная память имеет такую конфигурацию, инициализацию указателя стека обычно осуществляет система, присваивая ему значение 00 11 72; в противном случае инициализацию указателя SP должен осуществить программист.

Давайте попытаемся теперь занести (записать) информацию в стековую память и извлечь (прочитать) ее оттуда с помощью регистра R6 (или указателя стека SP). Нам нужны только две инструкции – одна для записи, другая – для чтения. Поскольку для выполнения **записи** мы фактически проталкиваем информацию в стек, а для **чтения** выталкиваем ее из стека, эти операции известны также как **проталкивание** и **выталкивание.**

Для выполнения **проталкивания** мы используем инструкцию

MOV ист, – (SP)

Содержимое исходного операнда копируется в ячейку памяти, расположенную по адресу, равному текущему значению указателя стека минус 2. Предположим, что указатель стека был инициализирован значением 1172. Тогда исходный операнд при выполнении нашей инструкции будет протолкнут в ячейку 1170.

Пример 1. Приведенная ниже программа будет сохранять в стеке содержимое регистров R0, R1 и R2.

```
START:  MOV     R0,-(SP)        ;ПРОТОЛКНУТЬ (R0)
        MOV     R1,-(SP)        ;ПРОТОЛКНУТЬ (R1)
        MOV     R2,-(SP)        ;ПРОТОЛКНУТЬ (R2)
        HALT
        .END    START
```

Карта выполнения этой программы:

До выполнения программы	После выполнения программы
(PC): 00 11 72	(PC) = 00 12 02
(SP): 00 11 72	(SP) = 00 11 64
(R0): 00 10 10	(R0) = Б. И.
(R1): 00 11 00	(R1) = Б. И.
(R2): 00 12 34	(R2) = Б. И.

	Адрес	Содержимое	Адрес	Содержимое	
		.		.	
		.		.	
		.		.	
	00 11 60	xx xx xx	00 11 60	xx xx xx	
	00 11 62	xx xx xx	00 11 62	xx xx xx	
	00 11 64	xx xx xx	00 11 64	00 12 34	← (SP)
	00 11 66	xx xx xx	00 11 66	00 11 00	
	00 11 70	xx xx xx	00 11 70	00 10 10	
(SP) (PC)	00 11 72	01 00 46	00 11 72	Б. И.	
	00 11 74	01 01 46	00 11 74	Б. И.	
	00 11 76	01 02 46	00 11 76	Б. И.	
	00 12 00	00 00 00	00 11 00	Б. И.	
	00 12 02	xx xx xx	00 11 02	Б. И.	← (PC)

Обратите внимание, что после выполнения (SP) указывает на адрес 00 11 64, а (PC) указывает на адрес 00 12 02. В стековую память занесено три "слоя" информации, причем последняя порция информации, (R2), расположена в верхнем слое.

Для выполнения **выталкивания** мы используем инструкцию

MOV (SP) +, назн

Здесь "назн" обозначает операнд назначения. Таким образом содержимое верхнего слоя стека копируется по адресу операнда назначения, а (SP) автоматически увеличивается на 2.

Пример 2. Используем условие предыдущего примера в качестве начального условия, но положим, что по каким-то причинам содержимое регистров R0, R1 и R2 было изменено. Мы хотим восстановить первоначальное содержимое этих регистров. Напишем для этого следующую программу:

```
START:  MOV     (SP)+,R2        ;ВЫТОЛКНУТЬ
        MOV     (SP)+,R1        ;ВЫТОЛКНУТЬ
        MOV     (SP)+,R0        ;ВЫТОЛКНУТЬ
        HALT
        .END    START
```

Карта выполнения:

До выполнения программы	После выполнения программы
(PC): 00 11 72	(PC) = 00 12 02
(SP): 00 11 64	(SP) = 00 11 72
(R0): xx xx xx	(R0) = 00 10 10
(R1): xx xx xx	(R1) = 00 11 00
(R2): xx xx xx	(R2) = 00 12 34

	Адрес	Содержимое		Адрес	Содержимое	
	.			.	.	
	.				.	
	.			.	.	
	.	.		.	.	
	00 11 60	xx xx xx		00 11 60	xx xx xx	
	00 11 62	xx xx xx		00 11 62	xx xx xx	
(SP)→	00 11 64	00 12 34		00 11 64	xx xx xx	
	00 11 66	00 11 00		00 11 66	xx xx xx	
	00 11 70	00 10 10		00 11 70	xx xx xx	
(PC)→	00 11 72	01 26 02		00 11 72	Б. И.	←—(SP)
	00 11 74	01 26 01		00 11 74	Б. И.	
	00 11 76	01 26 00		00 11 76	Б. И.	
	00 12 00	00 00 00		00 12 00	Б. И.	
	00 12 02	xx xx xx		00 12 02	Б. И.	←—(PC)
	.	.		.	.	
		.			.	
		.			.	
		.			.	

Обратите внимание, что первоначально (SP) = 00 11 64, (PC) = 00 11 72, а после выполнения программы значение указателя стека снова стало равным 1172. Вся сохраненная информация оказывается вытолкнутой назад в регистры. При такой конфигурации области действия, определяемые (PC) и (SP), не перекрываются.

При рассмотрении процесса проталкивания-выталкивания видно, что содержимое регистра R2 было протолкнуто в стек последним, а вытолкнуто из стека первым. Память с конфигурацией такого типа известна как память, работающая по принципу "последним пришел – первым вышел". LIFO

КРАТКОЕ ИЗЛОЖЕНИЕ МЕТОДОВ АДРЕСАЦИИ

Режимы адресации. В этой главе мы подробно изучили режимы адресации, доступные в системе PDP-11. В табл. 4.1 приведены режимы адресации со ссылками на регистры, когда исключается использование регистра R7 (или PC). Третья колонка таблицы показывает изменение содержимого регистров до, в процессе и после извлечения операнда. Например, в режиме 0 регистр R*n* представляет собой адрес, а (R*n*) – операнд, поэтому в процессе извлечения операнда содержимое регистров меняться

Таблица 4.1. **Режимы адресации без использования регистра PC ЭВМ PDP-11 и LSI-11**

Обозначение поля адреса операнда	Режим	Операция адресации: Регистр R_n, $n = 0, 1, 2, \ldots, 5$			Память		Память		Операнд
		До выполнения	Во время выполнения	После выполнения	Адрес	Содержимое	Адрес	Содержимое	
R_n	0	(R_n)	(R_n)	(R_n)	R_n	(R_n)			(R_n)
@ R_n	1	(R_n)	(R_n)	(R_n)	(R_n)	A_1			A_1
				$(R_n)+2$					
$(R_n)^+$	2	(R_n)	(R_n)	$(R_n)+1$**	(R_n)	A_2			A_2
@$(R_n)^+$	3	(R_n)	(R_n)	$(R_n)+2$	(R_n)	C_1	C_1	D_1	D_1
			$(R_n)-2$	$(R_n)-2$	$(R_n)-2$				
$-(R_n)$	4	(R_n)	$(R_n)-1$**	$(R_n)-1$**	$(R_n)-1$**	A_3			A_3
@$-(R_n)$	5	(R_n)	$(R_n)-2$	$(R_n)-2$	$(R_n)-2$	C_2	C_2	D_2	D_2
$X(R_n)$	6	(R_n)	(R_n)	(R_n)	$(R_n)+X$	A_4			A_4
@$X(R_n)$	7	(R_n)	(R_n)	(R_n)	$(R_n)+X$	C_3	C_3	D_3	D_3

**Байтовая операция

A_1, A_2, A_3, A_4
C_1, C_2, C_3
D_1, D_2, D_3
} Δ некоторый двоичный код

не будет. А в режиме 4 (R*n*) увеличивается на 2 перед извлечением операнда. Колонки 4 и 5 представляют карту памяти для одноуровневой и двухуровневой косвенной адресации. В табл. 4.2 приведены режимы адресации с вовлечением только регистра PC.

Таблица 4.2. **Режимы адресации при использовании регистра PC**

Обозначение поля адреса операнда	Название	Режим	Адресация операнда			Память		Память		Операнд
			До выполнения	Во время выполнения	После выполнения	Адрес	Содержимое	Адрес	Содержимое	
#*n*	Непосредственный	2	(R_7)	$(R_7)+2$	$(R_7)+4$					n
@#A	Абсолютный (косвенный непосредственный)	3	(R_7)	$(R_7)+2$	$(R_7)+4$	A	C_1			C_1
A	Относительный	6	(R_7)	$(R_7)+2$	$(R_7)+4$	$(R_7)+4^*+B$	C_2			C_2
@A	Косвенный относительный	7	(R_7)	$(R_7)+2$	$(R_7)+4$	$(R_7)+4^*+B$	C_3	C_3	D_3	D_3

* Может быть 6, если в адресации обоих операндов используется относительный режим

n △ некоторое число

B △ содержимое ячейки, $(R_7)+2$

Процедура ассемблирования вручную. На нескольких примерах мы проиллюстрировали, как можно вручную ассемблировать исходную программу в соответствующую объектную программу в восьмеричном машинном коде.

Ручное ассемблирование кода осуществляется следующим образом:

Шаг 1. Решить, сколько слов необходимо для каждой инструкции.

Шаг 2. Выделить последовательные ячейки памяти, начиная с ячейки 0 для всех инструкций.

Шаг 3. Определить восьмеричный (двоичный) код для каждой инструкции.

Правило определения числа слов для каждой инструкции: инструкция занимает одно слово памяти, если в ней не используются следующие режимы адресации: индексный, индексный косвенный, непосредственный, абсолютный, относительный и относительный косвенный; при использовании этих режимов адресации добавляется по одному слову на каждый операнд инструкции (см. таблицу режимов адресации).

Значение сдвига (или относительного расстояния), представленного в виде дополнения до двух, имеет следующий порядок определения:

1. Для режима относительной адресации с символьным адресом операнда назначения A значение сдвига равно A – (скорректированное содержимое регистра PC).
2. Для инструкции ветвления A равно (скорректированное содержимое регистра PC) + 2 × (значение сдвига) или значение сдвига равно (1/2) × [A – (скорректированное содержимое регистра PC)].
3. Для инструкции SOB R, A, предписывающей вычесть 1 из (R) и осуществить ветвление к A, если результат не равен нулю, значение сдвига равно (1/2) × × [(скорректированное содержимое регистра PC) – A].

Ограничение: (скорректированное содержимое регистра PC) всегда больше чем A; т. е. значение сдвига всегда будет положительным 6-битовым двоичным числом.

ОБРАЗЦЫ ПРОГРАММ ПРИ ИСПОЛЬЗОВАНИИ РАЗЛИЧНЫХ РЕЖИМОВ АДРЕСАЦИИ

Примеры, поясняющие материал этой главы, приведены на рис. 4.4 – 4.8. Примеры 1–3 очевидны, а примерам 4 и 5 надо уделить внимание. В примере 4 для удобства

```
1                       ;ПРИМЕР1: ЭТОТ ПРИМЕР ПОКАЗЫВАЕТ ЧИСЛО СЛОВ ПАМЯТИ,
2                       ;ТРЕБУЕМЫХ ПОД ИНСТРУКЦИЮ ПРИ РАЗНЫХ РЕЖИМАХ АДРЕСАЦИИ
3                       ;
4 000000 010100  START: MOV     R1,R0           ;НУЖНО 1 СЛОВО
5 000002 012702         MOV     #20,R2          ;НУЖНО 2 СЛОВА
         000020
6 000006 010177         MOV     R1,@177370      ;НУЖНО 2 СЛОВА
         177370'
7 000012 012777         MOV     #10,@177372     ;НУЖНО 3 СЛОВА
         000010
         177372'
8 000020 000000         HALT                    ;НУЖНО 1 СЛОВО
9        000000'        .END    START
```

Рис. 4.4. Определение требуемого числа слов памяти

```
 1                      ;ПРИМЕР 2: РЕЖИМ ОТНОСИТЕЛЬНОЙ АДРЕСАЦИИ.
 2                      ;ПОИСК НАИБОЛЬШЕГО ИЗ ДВУХ ЧИСЕЛ, ХРАНЯЩИХСЯ
 3                      ;В ЯЧЕЙКАХ N1 И N2, И ПОМЕЩЕНИЕ ЕГО
 4                      ;НА МЕСТО НАИБОЛЬШЕГО ЧИСЛА.
 5                      ;
 6 000000 000017  N1:    .WORD   15.             ;ЗДЕСЬ ХРАНИТСЯ ДЕСЯТИЧНОЕ 15
 7 000002 177744  N2:    .WORD   -28.            ;ЗДЕСЬ ХРАНИТСЯ ДЕСЯТИЧНОЕ -28
 8 000004         BIG:   .BLKW   1
 9 000006 026767  START: CMP     N1,N2
          177766
          177766
10 000014 002004        BGE     POS             ;ИСТИНА, ЕСЛИ (N1) >= (N2)
11 000016 016767        MOV     N2,BIG
          177760
          177760
12 000024 000000        HALT
13 000026 016767  POS:  MOV     N1,BIG
          177746
          177750
14 000034 000000        HALT
15        000006'               .END    START
```

Рис. 4.5. Режим относительной адресации

```
 1                      ;ПРИМЕР 3: ДЕМОНСТРАЦИЯ НЕПОСРЕДСТВЕННОГО,
 2                      ;ОТНОСИТЕЛЬНОГО И ИНДЕКСНОГО РЕЖИМОВ АДРЕСАЦИИ.
 3                      ;
 4 000000  012701  START:  MOV     #2,R1
           000002
 5 000004  000411  L1:     BR      L2
 6 000006  062701          ADD     #2,R1
           000002
 7 000012  016103          MOV     L1+2(R1),R3     ;L1+2 = 4+2
           000006'
 8 000016  016100          MOV     177560(R1),R0
           177560
 9 000022  022701          CMP     #4,R1
           000004
10 000026  001402          BEQ     STOP
11 000030  000161  L2:     JMP     L1(R1)
           000004'
12 000034  000000  STOP:   HALT
13         000000'         .END    START
```

Рис. 4.6. Непосредственный, индексный и относительный режимы адресации

```
 1                      ;ПРИМЕР 4: ДЕМОНСТРАЦИЯ ИНСТРУКЦИИ SOB
 2                      ;
 3                              .MCALL  .WRITE,.EXIT
 4 000000 012700 START: MOV     #3,R0
          000003
 5 000004        L1:    .WRITE  "_ (R0)=,R0     ;НА ЭТОТ ПОЯВИТСЯ СОДЕРЖИМОЕ R0
 6 000016 077006        SOB     R0,L1   ;УМЕНЬШАТЬ НА 1, ПОКА НЕ СТАНЕТ (R0)=0
 7 000020               .WRITE  "\\ПОСЛЕ_ЗАВЕРШЕНИЯ_SOB_ (R0)=,R0
 8 000060               .EXIT
 9        000000'       .END    START

          РЕЗУЛЬТАТ:    (R0)=3  (R0)=2  (R0)=1
                        ПОСЛЕ ЗАВЕРШЕНИЯ SOB (R0)=0
```

Рис. 4.7. Инструкция SOB

```
 1                      ;ПРИМЕР 5А: ДЕМОНСТРАЦИЯ НЕКОТОРЫХ НЕТРИВИАЛЬНЫХ
 2                      ;           РЕЖИМОВ АДРЕСАЦИИ
 3                      ;
 4                      ;
 5                              .MCALL  .WRITE,.EXIT,.RAD
 6 000000  012737 START: MOV     #3000,@#2000    ;ИНИЦИАЛИЗАЦИЯ ЯЧЕЙКИ 2000
           003000
           002000
 7 000006  012737        MOV     #123,@#3000     ;ИНИЦИАЛИЗАЦИЯ ЯЧЕЙКИ 3000
           000123
           003000
 8 000014  013700        MOV     @#2000,R0
           002000
 9 000020  012701        MOV     #2000,R1
           002000
10 000024  017702        MOV     @2000,R2
           002000'
11 000030  016703        MOV     2000,R3
           002000'
12 000034                .WRITE  "\\И_РЕЗУЛЬТАТЕ_ИМЕЕМ:__
13 000040                .RAD    BL
14 000044                .WRITE  "\(R0)=,R0,"_(R1)=,R1,"_(R2)=,R2,"_(R3)=,R3
15 000050  013703        MOV     @#DATA,R3
           000144'
16 000054  013704        MOV     @#140,R4
           000140
17 000060  012702        MOV     #DATA,R2
           000144'
18 000064  017701        MOV     @DATA,R1
           000054
19 000070  016700        MOV     DATA,R0
           000050
20 000074  016767        MOV     A,C
           000046
           000050
21 000102  016767        MOV     B,C+2
           000042
           000044
22 000110  016767        MOV     A,B
           000032
           000032
23 000116                .WRITE  "\\ТЕПЕРЬ_МЫ_ИМЕЕМ:__
24 000122                .WRITE  "\(R0)=,R0,"_(R1)=,R1,"_(R2)=,R2,"_(R3)=,R3
25 000126                .WRITE  "_(R4)=,R4
26 000132                .WRITE  "\(A)=,A,"_(B)=,B,"_(C)=,C,"_(C+2)=,C+2
27 000136                .EXIT
```

```
28 000144  002000  DATA:   .WORD   2000
29 000146  000003  A:      .WORD   3
30 000150  000007  B:      .WORD   7
31 000152          C:      .BLKW   1000
32         000000'         .END    START

РЕЗУЛЬТАТ:      В РЕЗУЛЬТАТЕ ИМЕЕМ:
                (R0)=3000 (R1)=2000 (R2)=123 (R3)=3000
                ТЕПЕРЬ МЫ ИМЕЕМ:
                (R0)=2000 (R1)=3000 (R2)=1336 (R3)=2000 (R4)=-1
                (A)=3 (B)=3 (C)=3 (C+2)=7
* ЯЧЕЙКА 140 СОДЕРЖИТ -1
```

а)

```
      1                     ;ПРИМЕР 5Б: ДЕМОНСТРАЦИЯ НЕКОТОРЫХ НЕТРИВИАЛЬНЫХ
      2                     ;           РЕЖИМОВ АДРЕСАЦИИ
      3                     ;
      4                     ;
      5                             .MCALL  .WRITE,.EXIT,.RAD
M     6          000146             A = 146
      7 000000   012737     START:  MOV     #3000,@#2000    ;ИНИЦИАЛИЗАЦИЯ ЯЧЕЙКИ 2000
                 003000
                 002000
      8 000006   012737             MOV     #123,@#3000     ;ИНИЦИАЛИЗАЦИЯ ЯЧЕЙКИ 3000
                 000123
                 003000
      9 000014   013700             MOV     @#2000,R0
                 002000
     10 000020   012701             MOV     #2000,R1
                 002000
     11 000024   017702             MOV     @2000,R2
                 002000'
     12 000030   016703             MOV     2000,R3
                 002000'
     13 000034                      .WRITE  '\\В_РЕЗУЛЬТАТЕ_ИМЕЕМ:__
     14 000040                      .RAD    8L
     15 000044                      .WRITE  '\(R0)=,R0,'_(R1)=,R1,'_(R2)=,R2,'_(R3)=,R3
     16 000050   013703             MOV     @#DATA,R3
                 000146'
     17 000054   013704             MOV     @#140,R4
                 000140
     18 000060   012702             MOV     #DATA,R2
                 000146'
     19 000064   017701             MOV     @DATA,R1
                 000056
     20 000070   016700             MOV     DATA,R0
                 000052
D    21 000074   016767             MOV     A,C
                 000146'
                 000052
     22 000102   016767             MOV     B,C+2
                 000044
                 000046
D    23 000110   016767             MOV     A,B
                 000146'
                 000034
     24 000116                      .WRITE  '\\ТЕПЕРЬ_МЫ_ИМЕЕМ:__
     25 000122                      .WRITE  '\(R0)=,R0,'_(R1)=,R1,'_(R2)=,R2,'_(R3)=,R3
     26 000126                      .WRITE  '_(R4)=,R4
     27 000132                      .WRITE  '\(A)=,A,'_(B)=,B,'_(C)=,C,'_(C+2)=,C+2
D    28                             .NTYPE  $$$$,A
```

```
D     29 000136  000146                        .WORD   A
      30 000140                      .EXIT
      31 000146  002000  DATA:       .WORD   2000
M     32 000150  000003  A:          .WORD   3
      33 000152  000007  B:          .WORD   7
      34 000154          C:          .BLKW   1000
      35         000000'             .END    START
```

б)

```
 1                    ;ПРИМЕР 5Б: ДЕМОНСТРАЦИЯ НЕКОТОРЫХ НЕТРИВИАЛЬНЫХ
 2                    ;           РЕЖИМОВ АДРЕСАЦИИ
 3                    ;
 4                    ;
 5                            .MCALL  .WRITE,.EXIT,.RAD
 6          000146            A = 146
 7 000000   012737    START:  MOV     #3000,@#2000    ;ИНИЦИАЛИЗАЦИЯ ЯЧЕЙКИ 2000
            003000
            002000
 8 000006   012737            MOV     #123,@#3000     ;ИНИЦИАЛИЗАЦИЯ ЯЧЕЙКИ 3000
            000123
            003000
 9 000014   013700            MOV     @#2000,R0
            002000
10 000020   012701            MOV     #2000,R1
            002000
11 000024   017702            MOV     @2000,R2
            002000'
12 000030   016703            MOV     2000,R3
            002000'
13 000034                     .WRITE  '\\В_РЕЗУЛЬТАТЕ_ИМЕЕМ:__
14 000040                     .RAD    8L
15 000044                     .WRITE  '\(R0)=,R0,'_(R1)=,R1,'_(R2)=,R2,'_(R3)=,R3
16 000050   013703            MOV     @#DATA,R3
            000144'
17 000054   013704            MOV     @#140,R4
            000140
18 000060   012702            MOV     #DATA,R2
            000144'
19 000064   017701            MOV     @DATA,R1
            000054
20 000070   016700            MOV     DATA,R0
            000050
21 000074   016767            MOV.    A,C
            000146'
            000050
22 000102   016767            MOV     B,C+2
            000042
            000044
23 000110   016767            MOV     A,B
            000146'
            000032
24 000116                     .WRITE  '\\ТЕПЕРЬ_МЫ_ИМЕЕМ:__
25 000122                     .WRITE  '\(R0)=,R0,'_(R1)=,R1,'_(R2)=,R2,'_(R3)=,R3
26 000126                     .WRITE  '_(R4)=,R4
27 000132                     .WRITE  '\(A)=,A,'_(B)=,B,'_(C)=,C,'_(C+2)=,C+2
28 000136                     .EXIT
29 000144   002000  DATA:     .WORD   2000
30 000146   000003            .WORD   3
31 000150   000007  B:        .WORD   7
32 000152          C:        .BLKW   1000
33          000000'           .END    START
```

```
РЕЗУЛЬТАТ:      В РЕЗУЛЬТАТЕ ИМЕЕМ:
                (R0)=3000  (R1)=2000  (R2)=123  (R3)=3000
                ТЕПЕРЬ МЫ ИМЕЕМ:
                (R0)=2000  (R1)=3000  (R2)=1336  (R3)=2000  (R4)=-1
                (A)=-1  (B)=-1  (C)=-1  (C+2)=7
* ЯЧЕЙКА 146 СОДЕРЖИТ -1
* ЯЧЕЙКА 140 СОДЕРЖИТ -1
```

в)

Рис. 4.8. Иллюстрация нетривиального режима адресации

используются макроинструкции .WRITE и .EXIT. Поскольку макроинструкции будут описаны в гл. 6, здесь мы ограничимся лишь кратким описанием их функций. Инструкция .EXIT используется для замены инструкции HALT, а инструкция .WRITE применяется, чтобы отобразить содержимое определенной области памяти или регистра без использования отладчика ODT. Например, инструкция .WRITE в строке 6 при выполнении каждый раз будет отображать содержимое регистра R0 на экране ЭЛТ в следующем виде:

(R0) = (содержимому в восьмеричном представлении)

В примере 5 делается попытка прояснить некоторые режимы адресации, способствующие внесению неясностей. Для удобства мы применяем здесь еще одну определенную в системе макроинструкцию .RAD, задающую основание системы счисления, которым мы хотим пользоваться. Например, инструкция в строке 13 указывает, что последующие отображения мы хотим производить с восьмеричным основанием системы счисления и с выравниванием по левому краю. Ниже мы приводим результаты, полученные после выполнения программы, представленной на рис. 4.8, *а*.

1. После строки 13 строки 11, 12 и 13 дают следующий результат: (R0) = 3000, (R1) = 2000, (R2) = 123, (R3) = 3000.
2. После строки 26 строки 23, 24 и 25 дают следующий результат:

```
ТЕПЕРЬ МЫ ИМЕЕМ:          ;РЕЗУЛЬТАТ ВЫПОЛНЕНИЯ СТРОКИ 23
(R0)=2000, (R1)=3000
(R2)=1332                 ;РЕЗУЛЬТАТ ВЫПОЛНЕНИЯ СТРОКИ 24
(R3)=2000
(R4)=-1                   ;РЕЗУЛЬТАТ ВЫПОЛНЕНИЯ СТРОКИ 25
(A)=3
(B)=3
(C)=3                     ;РЕЗУЛЬТАТ ВЫПОЛНЕНИЯ СТРОКИ 26
(C+2)=7
```

Замечание 1. В строке 17 используется режим непосредственной адресации. В соответствии с объектным кодом (R2) = 144. Однако, поскольку объектный файл, полученный после обработки построителем задач ТКВ, был загружен для выполнения в область памяти, начинающуюся с адреса 1172, в нашей системе

(R2) = 144 + 1172 = 1336.

Такой результат получается из-за того, что метка DATA была перемещена на адрес 1336.

Замечание 2. В строке 16 используется режим абсолютной адресации; т. е. ЦП извлекает содержимое ячейки 140 и помещает его в регистр R4. Нам известно, что ячейка 140 расположена внутри области, которую использует система; в нашей машине в этой ячейке содержится −1.

В качестве другого примера рассмотрим рис. 4.8, *б*. Обратите внимание, что ассемблер "пожаловался" нам, что мы сделали шесть ошибок: две ошибки M (многократное определение метки) и четыре ошибки D (ссылка на многократно определенный сим-

вол). При внимательном изучении находим, что мы использовали оператор прямого присваивания A = 146 и в строке 30 мы опять используем символьный адрес A в качестве адресной метки для этой ячейки. После обработки построителем задач ТКВ (подробности см. в приложении, где приводятся сведения по операционной системе RSX =11M) эта ячейка будет иметь адрес 146 + 1172 = 1340. Поэтому ЭВМ "считает", что мы определим метку A дважды, и отказывается с нами взаимодействовать. В результате всякий раз при использовании метки A ассемблер будет отмечать ошибку D. В данном примере ошибка отмечается в строках 21, 23 и 27. Чтобы устранить эти ошибки, нужно либо просто удалить строку 6, как показано на рис. 4.8, *а*, либо поместить строку 30 каким-либо уникальным символом, либо просто стереть ее (если это приемлемо). На рис 4.8, *в* показан результат в случае стирания метки в строке 30. Обратите внимание, что из-за удаления метки в строке 30 адресное значение A теперь равно 146 и является абсолютным. Поскольку в нашей машине содержимое ячеек памяти с абсолютными адресами 140 и 146 равно −1, мы получаем результат, приведенный в нижней части рис. 4.8, *в*, т. е.

```
(A)=-1
(B)=-1                  ;ИЗ-ЗА ВЫПОЛНЕНИЯ СТРОКИ 23, ХОТЯ
                        ;ПЕРВОНАЧАЛЬНОЕ ЗНАЧЕНИЕ ПРИ ССЫЛКЕ
                        ;В СТРОКЕ 31 БЫЛО 7
(C)=-1                  ;ИЗ-ЗА ВЫПОЛНЕНИЯ СТРОКИ 21
(C+2)=7                 ;ИЗ-ЗА ВЫПОЛНЕНИЯ СТРОКИ 22
```

4.3. НЕОЧЕВИДНЫЕ ПРИМЕНЕНИЯ НЕКОТОРЫХ ИНСТРУКЦИЙ

Подробное описание набора инструкций дано в приложении Б, а также в справочных руководствах, опубликованных фирмой Digital Equipment Corporation. Здесь мы опишем некоторые неочевидные применения выбранной из набора инструкций подгруппы, чтобы показать, как иногда инструкции могут использоваться для целей, отличных от тех, для которых они предназначались.

Инструкция TST (R1)+

Первоначальное назначение инструкции TST – проверка операнда и установка бит условий в PSW. Однако здесь мы хотим лишь увеличить содержимое регистра R1 на 2. Конечно, такой же результат может быть достигнут с помощью инструкции ADD#2, R1, но для этой инструкции требуется два слова памяти, а для инструкции TST нужно только одно.

Инструкция ASR A

Эта инструкция делит содержимое A на 2. Аналогично инструкция ASL A умножит (A) на 2.

Инструкция BIT #10, A

Эта инструкция выполняет логическую операцию И для двух операндов, что может быть использовано для проверки состояния определенного бита или бит в операнде назначения. В данном случае содержимое ячейки A сравнивается по операции И с числом #10 следующим образом:

#10	= 0	000	000	000	001	000
(A)	$= b_{15}$	$b_{14}b_{13}b_{12}$	$b_{11}b_{10}b_9$	$b_8b_7b_6$	$b_5b_4b_3$	$b_2b_1b_0$
Результат	= 0	000	000	000	$00\,b_3$	000

Если $b_3 = 1$, то флаг Z в PSW будет равен нулю (ЛОЖЬ), в противном случае он будет равен единице (ИСТИНА). Как до, так и после выполнения оба операнда остаются неизменными. Это удобный способ определения значения бита b_3.

Инструкция BIC ист, назн

Эта инструкция выполняет операцию в два шага.

Шаг 1. Выполняет операцию дополнения до единицы исходного операнда (**исх**).

Шаг 2. Выполняет логическую операцию **И** над результатом шага 1 и операндом назна-

чения (**назн**) и помещает результат на место операнда назначения. Эта инструкция может быть использована для очистки одного или нескольких определенных бит операнда назначения. Например, если мы хотим очистить бит b_3 содержимого ячеек A, B и C, то можно использовать следующую программу:

```
MOV     #10,R1          ;ИНИЦИАЛИЗИРОВАТЬ (R1)
BIC      R1,A           ;ОЧИСТИТЬ БИТ 3 В (A)
BIC      R1,B           ;ОЧИСТИТЬ БИТ 3 В (B)
BIC      R1,C           ;ОЧИСТИТЬ БИТ 3 В (C)
```

Аналогично можно установить один или несколько бит в операнде назначения, если воспользоваться инструкцией установки бит (BIS), поскольку эта инструкция выполняет логическую операцию ИЛИ. Например, для установки бит $b_5b_4b_3$ в ячейках A, B и C в единицу может быть сделано следующее:

```
MOV     #70,R1
BIS      R1,A
BIS      R1,B
BIS      R1,C
```

СРАВНЕНИЕ ИНСТРУКЦИЙ COM И NEG

Инструкция дополнения COM выполняет операцию дополнения до единицы, а инструкция NEG – операцию дополнения до двух. Например:

```
1)  MOV     #31,R1          ;ИНИЦИАЛИЗАЦИЯ
    COM      R1             ;ДОПОЛНЕНИЕ ДО 1 (R1) -> R1
```

В результате (R1) = 177746

```
2)  MOV     #20,R1
    NEG      R1
```

В результате (R1) = 177760.

СРАВНЕНИЕ ИНСТРУКЦИЙ CMP И SUB

Инструкция сравнения CMP сравнивает исходный операнд (исх) с операндом назначения (**назн**) с помощью операции (**исх**)–(**назн**), после которой соответственно устанавливает коды условий в PSW. Но содержимое обоих операндов при этом остается неизменным.

В свою очередь, инструкция вычитания SUB выполняет операцию (**назн**)–(**исх**). Разность помещается на место операнда назначения, что приводит к потере его первоначального содержимого. Обратите также внимание на то, что в этом случае порядок вычитания противоположен случаю инструкции сравнения. Например:

```
1)  MOV     #31,A
    MOV     #20,B
    CMP      A,B
```

В результате (A) – без изменения; (B) – без изменения; в PSW N = 0, Z = 0, V = 0, C = 1

```
2)  MOV     #31,A
    MOV     #20,B
    SUB      A,B
```

В результате (A) – без изменения; (B) = 177767; в PSW N = 1, Z = 0, V = 0, C = 1.

СРАВНЕНИЕ ИНСТРУКЦИЙ ВЕТВЛЕНИЯ С УЧЕТОМ И БЕЗ УЧЕТА ЗНАКА

Основное различие этих двух групп инструкций состоит в том, что инструкции ветвления с учетом знака имеют дело с числами в виде дополнения до двух, а инструкции ветвления без учета знака оперируют числами, которые знакового бита не имеют. Для 16-битового слова 077777 – это наибольшее число, а 100000 – это наименьшее число при представлении с учетом знака; 177777 – это наибольшее, а 000000 – это наименьшее число при представлении без учета знака. Различие проиллюстрируем следующим примером:

```
N1:     .WORD   -2
N2:     .WORD   7
RESULT: .BLKW   1
START:  CMP     N1,N2
        BGE     POS             ;УСЛОВНОЕ ВЕТВЛЕНИЕ С УЧЕТОМ ЗНАКА
        MOV     N2,RESULT
        HALT
POS:    MOV     N1,RESULT
        HALT
        .END    START
```

Здесь мы использовали условное ветвление с учетом знака. Поэтому после выполнения эта программа поместит в ячейку RESULT число 7. Но если заменить инструкцию BGE POS инструкцией условного ветвления без учета знака BHI POS, то программа поместит в ячейку RESULT число –2. Это произойдет потому, что число –2 ассемблером будет преобразовано в его дополнение до двух, 177776, а это число больше 7, если оно не рассматривается как число со знаком. Одно из типичных применений условного ветвления без учета знака – сравнение адресных значений двух символьных адресов. Очевидно, что, например, адресное значение 177750 больше адресного значения 000010.

ИНСТРУКЦИЯ NOP

Эта инструкция приводит к тому, что (PC) увеличивается при ее извлечении, а на программу никакого воздействия не оказывается; тем не менее она занимает одно слово памяти и требует для выполнения примерно 3,5 мкс. Этой инструкцией можно пользоваться в режиме отладки (например, с помощью отладчиков ODT) для проверки правильности работы системы; ее можно вставлять в любое нужное место (или места) программы в качестве фиктивной инструкции для отладки с помощью отладчика ODT. Например, если мы знаем, что в некоторой секции программы возможна логическая ошибка, то можем заранее вставить в этом месте инструкцию NOP, а затем использовать отладчик для замены "внедренной" нами в программу инструкции инструкцией останова HALT, после чего программа остановится на этой ячейке и мы получим возможность исследовать состояние программы.

4.4. ПЕРЕМЕЩАЕМЫЕ АДРЕСА

Вспомним, что функция описанного ранее построителя задач (TKB) состоит в увязывании (компоновке) одного или нескольких объектных файлов в один файл, определенный как файл образа памяти (TSK), готовый к загрузке с диска в оперативную память для выполнения. Но стартовый адрес любого объектного файла всегда равен нулю. Без процесса компоновки все объектные файлы загружались бы в оперативную память, начиная с нулевой ячейки, что приводило бы к перезаписи одного файла другим и к разрушению выполняемого файла. Поэтому построитель задач TKB является тем системным программным обеспечением, которое ответственно за решение этой проблемы. Если: 1) три объектных файла A, B и C должны быть скомпонованы в один файл; 2) под файл A требуется 4 слова памяти, под файл B – 10 слов, а под файл C – 20 слов; система строится так, что все выполняемые программы должны начинаться с ячейки 1000, то файл A будет загружаться от адреса 1000 до адреса 1006,

файл B – от адреса 1010 до адреса 1026, а файл C – от адреса 1030 до адреса 1066. Обратите внимание, что ни одна из подпрограмм не начинается с нуля, поскольку стартовый адрес каждого файла изменился соответственно на 1000, 1010 и 1030, а первоначально он был равен 0000.

Другими словами, подпрограммы с целью выполнения перемещаются в различные неперекрывающиеся, но расположенные последовательно области памяти. Это может привести к проблемам, если в подпрограмме используется символьный адрес. Поскольку значение символьному адресу обычно присваивается в предположении, что программа начинается с нулевой ячейки, то к нему следует прибавить базовое значение, с которого начинается подпрограмма. Чтобы облегчить такую модификацию для построителя задач ТКВ, ассемблер обычно помечает апострофом адресные значения, относящиеся к символьным адресам и нуждающиеся в перемещении, поэтому ТКВ остается только модифицировать базовым значением все числа, помеченные апострофом.

В качестве примера рассмотрим пример 3 (рис 4.6). В этом примере используются четыре символьных адреса: START, L1, L2 и STOP, имеющие первоначальные адресные значения 000000, 000004, 000030 и 000034 соответственно. Из текста программы (или исходного кода) видно, что только инструкции в строках 6, 8, 11, 12 и 14 ссылаются на эти символьные адреса. Но в строках 6 и 11 используется относительное расстояние, поэтому здесь нет прямой функциональной зависимости от адресного значения символьных адресов L2 и STOP. В строке 8 индексное значение равно L1 + 2. При изменении значения L1 индексное значение будет меняться соответственно, что и учитывается инструкцией в строке 12. Аналогично в строке 14 псевдоинструкция .END ссылается на символьный адрес START, имеющий первоначальное адресное значение, равное нулю. Но после перемещения построителем задач стартового адреса его значение станет другим, поэтому его также следует модифицировать. Предположим, что эта программа должна загружаться, начиная с адреса 1000. Тогда критическими символьными адресами, влияющими на программу, будут L1 и START. Их адресные значения – 1004 для L1 и 1000 для START. Индексные значения в строках 8 и 12 должны быть изменены следующим образом: 000006' на 001006 и с 000004' на 001004 соответственно, а значение в строке 11: 000000' на 001000. Обратите внимание, что апострофом были помечены только эти критические числа. Адреса, модифицируемые таким образом, называют перемещаемыми адресами.

4.5. ПОЗИЦИОННО НЕЗАВИСИМЫЙ КОД (ПНК)

Программа, не требующая модификации адресов построителем задач независимо от ее стартового адреса после ассемблирования, известна как программа, написанная в **позиционно независимом** коде (ПНК). Очевидно, что программы, в которых используются только инструкции со ссылками на регистры, являются программами в ПНК; исключение составляет индексный режим, когда индексом служит символьный адрес. Преимущество программ в ПНК состоит в том, что их можно перемещать в любое место оперативной памяти без необходимости осуществлять модификацию адресов. Ниже приведены типичные примеры.

Пример 1. Следующие инструкции образуют ПНК:

```
1)  A:      .WORD   16
    B:      .BLKW   1
            MOV     A,B
2)          MOV     R0,-(SP)
            MOV     R1,-(SP)
            MOV     (SP)+,R1
            MOV     (SP)+,R0
3)  LOOP:   MOV     @#177756,B
            BR      LOOP
    B:      .BLKW   1
```

Пример 2. Однако следующие инструкции не образуют ПНК:

```
A:      .WORD   16
B:      .BLKW   1
        MOV     @#A,B
```

Сравните этот пример с пунктом 3) предыдущего примера. В обоих случаях используется абсолютный режим адресации. В первом примере содержимое второго слова инструкции MOV – 177756, а во втором примере – текущее значение символьного адреса A, которое будет соответственно меняться при перемещении программы.

Следовательно, построитель задач должен модифицировать значение второго слова инструкции перемещения MOV @ # A, B.

4.6. УПРАЖНЕНИЯ

1. Ассемблируйте вручную следующие наборы независимых инструкций.

Вы можете полагать, что каждый набор является независимым, а значения меток символьных адресов определяются следующим образом:

```
START = 2000
    A = 1770
    B = 2760
```

А)
```
START:  MOV     #61,R0
        INC     R0
        HALT
```

Б)
```
START:  MOV     #1200,R1
        MOV     #1717,R2
        SUB     R1,R2
        HALT
```

В)
```
START:  MOV     #1200,R1
        MOV     #1717,R2
        CMP     R1,R2
        HALT
```

Г)
```
START:  MOVB    #1200,R1
        MOVB    #1717,R2
        CMPB    R1,R2
        HALT
```

Д)
```
A:      .WORD   1200
START:  CMP     A,B
        HALT
B:      1717
```

Е)
```
A:      .WORD   10,12,7,3
START:  MOV     #A,R1
        CMP     (R1)+,(R1)
        CMP     (R1)+,(R1)+
        SUB     (R1),-(R1)
        HALT
```

Ж)
```
A:      .WORD   0,1,2,3,4,0,-1,-2
START:  MOV     #A,R0
        MOV     #B,R1
        MOV     #10,R2
LOOP:   MOV     (R0)+,(R1)+
        SOB     R2,LOOP
        HALT
B:      .BLKW   10
```

```
З) A:      HALT
   START:  MOV   #3,R0
           DEC   R0
           BNE   START
           JMP   A

И) A:      .BYTE 0,1,2,3,4,5
   START:  MOV   #A,R0
           MOV   #B,R1
           MOV   #6,R2
   LOOP:   MOV   (R0)+,(R1)+
           SOB   R2,LOOP
           HALT
   B:      .BLKB 6

К) A:      .WORD 2000
   START:  MOV   A,R0
           MOV   (R0)+,R1
           MOV   (R0)+,R2
           MOV   R0,R3
           HALT
Л) A:      .WORD 2000
   START:  MOV   A,R0
           MOV   (R0)+,B
           MOV   (R0)+,B
           TST   (R0)+
           MOV   (R0),B
           HALT
   B:      .BLKW 1

М) A:      .WORD 2000
   START:  MOV   A,R0
           MOV   (R0)+,-(SP)
           MOV   (R0)+,-(SP)
           MOV   R0,-(SP)
           HALT

Н) A:      .WORD 2000
   START:  MOV   @A,-(SP)
           MOV   @#A,-(SP)
           MOV   A,-(SP)
           HALT
```

2. Выполните вручную инструкции из упр. 1 и запишите содержимое каждого регистра (R0, . . . , R7, PSW) и каждой ячейки памяти, которая подвергалась изменению при выполнении программы.

ГЛАВА 5

ПОДПРОГРАММЫ

В предыдущих главах мы изучили работу набора инструкций ЭВМ PDP-11 и приобрели некоторый опыт разработки простой программы. Чтобы разработать любую программу, простую или сложную, необходимо выполнить следующую процедуру: РЕДАКТОР → АССЕМБЛЕР → ПОСТРОИТЕЛЬ ЗАДАЧ → ЗАГРУЗЧИК и ЗАПУСК → ОТЛАДКА. Эта процедура показана на рис. Д.5 приложение Д. Поскольку разработка программы поглощает немало времени, нет смысла повторно разрабатывать программу, если существует ее отработанный экземпляр. И, конечно, хорошую программу хотелось бы использовать снова и снова.

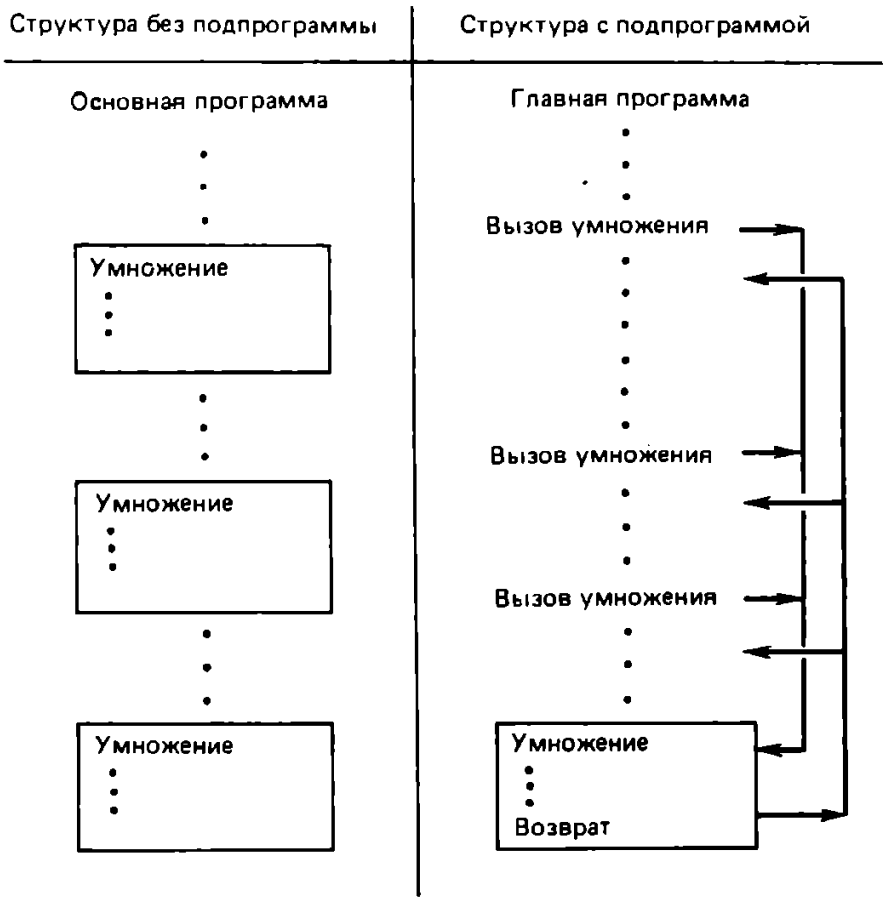


Рис. 5.1. Карта памяти для концепции подпрограммы

Предположим, например, что мы разработали хорошую программу, способную умножать два числа. Если в какой-либо прикладной программе нам потребуется несколько раз выполнить умножение двух чисел, то, без сомнения, мы постараемся вставить эту программу в нашу программу столько раз, сколько необходимо, а для этого мы перепечатаем эту программу (назовем ее подпрограммой) внутри основной программы столько раз, сколько требуется. При таком перепечатывании не исключена вероятность появления ошибок, не говоря уже о неизбежных дополнительных потерях времени. Кроме того, для каждого нового помещения подпрограммы требуется новый блок оперативной памяти. Рассмотрим подпрограмму, для которой требуется, например, 1000 слов памяти. Если в нашей программе эта подпрограмма нужна 10 раз, то дополнительно потребуется 10 000 слов памяти. Это может оказаться слишком дорогой ценой. К счастью, существует способ разрешения этой проблемы. Он заключается в концепции подпрограммы, приведенной на рис. 5.1.

Мы можем выделить для подпрограммы одну область памяти и использовать инструкцию перехода JUMP для входа в эту область всякий раз, когда в основной программе нужно выполнить операцию, реализованную подпрограммой, возвращаясь назад в основную программу после завершения работы подпрограммы. Основную программу часто называют **главной программой**, а процесс перехода — **вызовом подпрограммы**.

5.1. ОСНОВНЫЕ ПОНЯТИЯ

На рис. 5.2 показано выполнение главной программы, вызывающей подпрограмму. Обратите внимание, что при вызове подпрограммы есть два существенных шага: вход в подпрограмму из главной программы и возврат из подпрограммы в главную программу. Поясним некоторые детали. Вспомним, что в наборе инструкций имеется класс, называемый инструкциями управления последовательностью выполне-

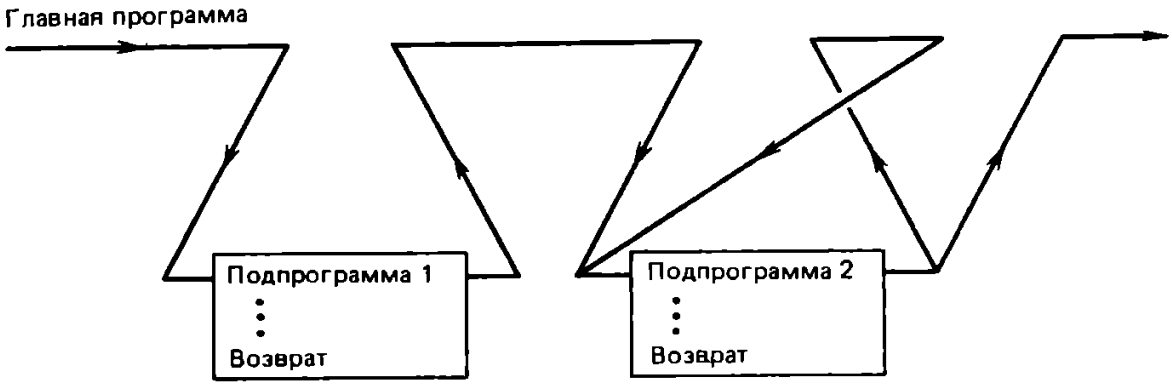


Рис. 5.2. Выполнение главной программы, содержащей подпрограммы

ния программы, куда входят такие инструкции, как BRANCH и JUMP. Однако первая из них обладает ограниченным диапазоном ветвления: от –256 до +254 слов, следовательно, для выхода из главной программы и входа в подпрограмму должна быть выбрана инструкция JUMP.

Кроме входа в подпрограмму нужно еще, чтобы ЭВМ могла узнать, в какое место главной программы следует осуществить возврат после завершения операции в подпрограмме. Вспоминаем, что в качестве программного счетчика или указателя, позволяющего выполнять программу шаг за шагом, мы использовали регистр PC. И главной программе, и подпрограмме для выполнения необходимо использовать один и тот же регистр PC. Чтобы разрешить эту проблему, ЭВМ должна располагать способом временного запоминания старого содержимого регистра PC вплоть до завершения работы подпрограммы. Таким образом, нам необходима пара работающих совместно связующих инструкций – одна для выхода из главной программы в подпрограмму, а другая – для возврата. Обычно инструкция для выхода включается в главную программу, а инструкция для возврата – в подпрограмму.

Последняя проблема, возникающая при вызове подпрограммы, сопряжена с передачей параметров, посредством которых подпрограмме могут поставляться необходимые данные. Как правило, данные для подпрограммы от вызова к вызову меняются. Например, если вызывается подпрограмма для выполнения умножения чисел, то может потребоваться каждый раз поставлять подпрограмме различные данные. Следовательно, способ, каким главная программа передает параметры подпрограмме, является существенным.

Итак, применяя вызовы подпрограмм, мы должны быть в курсе следующих проблем:

1. Определение **точки входа** или стартового адреса подпрограммы.
2. Выбор **пары связующих инструкций.**
3. Сохранение содержимого регистра PC для возврата в главную программу.
4. Средства **передачи параметров** для транспортировки данных от главной программы к подпрограмме и обратно.

КЛЮЧЕВЫЕ ПАРЫ ИНСТРУКЦИЙ ДЛЯ СВЯЗЫВАНИЯ ГЛАВНОЙ ПРОГРАММЫ С ПОДПРОГРАММОЙ

Использование R*n* ($n = 0, \ldots, 5$) в качестве связующего регистра

Инструкция выхода в подпрограмму

Мнемонический код: JSR R*n*, назн

Восьмеричный машинный код: 0 04 R*n*, назн

ОР

Здесь Rn $\triangleq$ n-ый регистр при $n = 0, 1, \ldots, 5$; для Rn требуется три бита: b_8, b_7, b_6.
назн $\triangleq$ символьный стартовый адрес подпрограммы или просто имя подпрограммы; требуется шесть бит: $b_5 \ldots b_0$.
ОР $\triangleq$ **относительное расстояние** между **скорректированным** (PC) и адресом назначения.

Обратите внимание, что эта инструкция – инструкция перехода специально для подпрограмм. Ее название состоит из первых букв английских слов, означающих "*переход к подпрограмме*". Этой инструкции требуется два слова памяти; для операнда назначения используется режим относительной адресации.

Хотя детали выполнения этой инструкции знать и не обязательно, процесс выполнения в целом понимать необходимо. Рассмотрим в качестве примера инструкцию JSR R5, назн; выполняются следующие действия:

1-й цикл извлечения: (PC) → РАП
(PC) + 2 → PC
Код инструкции JSR → РИ
2-й цикл извлечения: (PC) → РАП
(PC) + 2 → PC
ОР → ЦП

Заметим, что, поскольку эта инструкция занимает два слова, требуется два цикла извлечения. Далее при выполнении старое содержимое регистра R5 сохраняется путем проталкивания его в стековую память. Инструкция перехода к подпрограмме как бы "заимствует" указанный в ней регистр. Но этот регистр может содержать важную информацию, которая может понадобиться после выполнения подпрограммы. После сохранения старого содержимого регистра R5 ЦП может использовать его для временного сохранения адреса возврата, позволяющего возобновить выполнение главной программы. Затем в регистр PC загружается адресное значение для операнда назначения, являющееся суммой ОР и скорректированного (PC). После этого ЦП начинает выполнение подпрограммы, производя следующие действия:

(R5) → –(SP)
$(PC)_{скор}$ → R5
ОР + $(PC)_{скор}$ → PC

К счастью, все эти действия выполняются внутри ЦП, и программист за их выполнение не отвечает. Однако следует знать, какая информация где и когда находится. Проницательный читатель может здесь удивиться, почему мы беспокоимся о регистре R5. Почему бы нам просто не протолкнуть скорректированное содержимое регистра PC в стек? Такой метод мы рассмотрим ниже. Регистр R5 обеспечивает две функции: удержание адреса возврата и передачу параметров к подпрограмме и обратно. Как использовать регистр R5 в этом случае, будет описано ниже.

Инструкция возврата

Мнемонический код: RTS Rn
Восьмеричный машинный код: 00020 Rn; для Rn требуется три бита: $b_2 b_1 b_0$

Указываемый здесь регистр должен быть тем же самым, что и в инструкции JSR, описанной выше, а процесс выполнения следующий: скорректированное содержимое регистра (PC) или адрес возврата, который был сохранен в регистре R5, загружается в регистр PC, а верхний элемент стека выталкивается в регистр R5. То есть исходное содержимое регистра R5, ранее протолкнутое в стек, восстанавливается, и главная программа может продолжать работу. Эти действия производит ЦП, а не программист.
Пример. Предположим, что инструкция перехода к подпрограмме находится в ячейке 2000; подпрограмма с именем LOAFER начинается с адреса 3000; стековая память

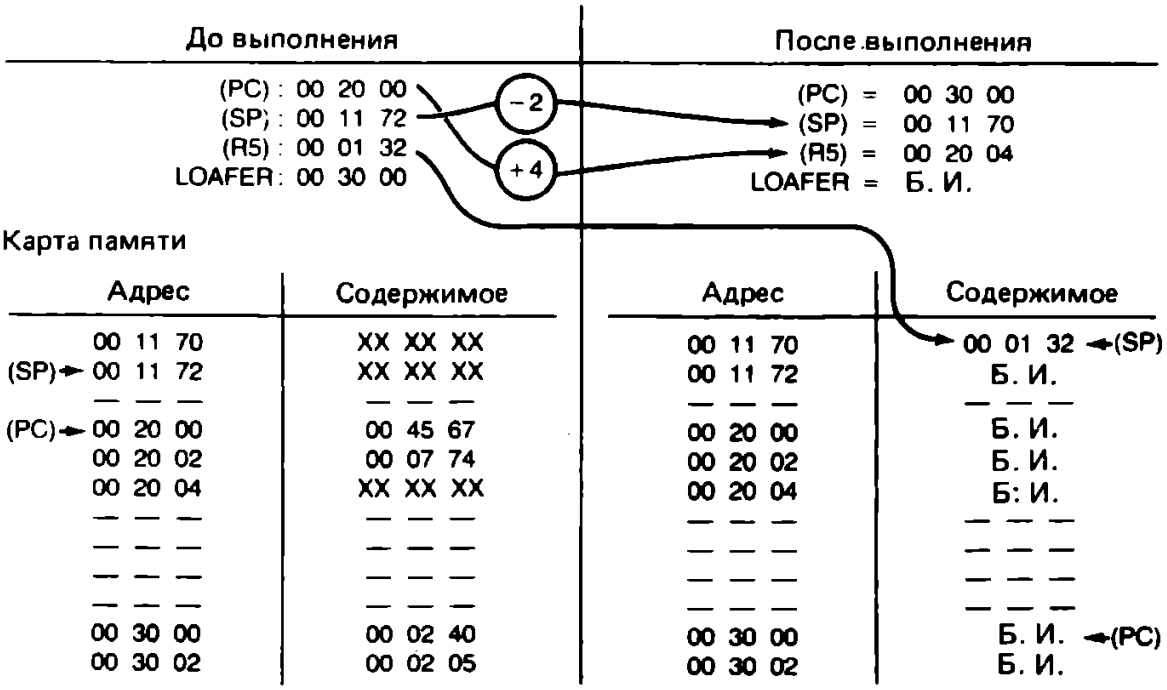
начинается с адреса 1172 и продолжается в обратном направлении, т. е. в сторону уменьшения адресов. Исходная программа может иметь вид:

Диапазон стека – 00 07 14 → 00 11 70
Начало стека – 00 11 72

Символьный адрес	Числовой адрес	Исходный код
START	— — — —	— — — —
	00 20 00	JSR R5, LOAFER
	— — — —	— — — —
	— — — —	— — — —
LOAFER	00 30 00	NOP
	00 30 02	RTS R5
	— — — —	— — — —

Обратите внимание, что для иллюстрации концепции подпрограммы мы использовали фиктивную подпрограмму. Карты выполнения для обеих инструкций:

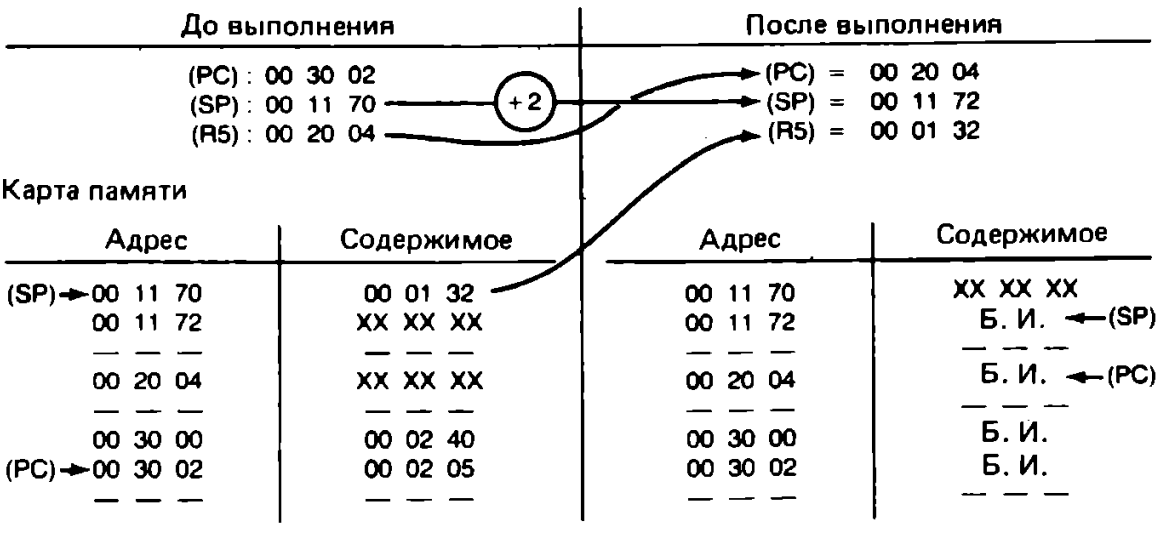
Карта выполнения инструкции [JSR R5, LOAFER]



Карта памяти

Адрес	Содержимое	Адрес	Содержимое
00 11 70	XX XX XX	00 11 70	00 01 32 ←(SP)
(SP)→ 00 11 72	XX XX XX	00 11 72	Б. И.
— — —	— — —	— — —	— — —
(PC)→ 00 20 00	00 45 67	00 20 00	Б. И.
00 20 02	00 07 74	00 20 02	Б. И.
00 20 04	XX XX XX	00 20 04	Б: И.
— — —	— — —	— — —	— — —
— — —	— — —	— — —	— — —
— — —	— — —	— — —	— — —
— — —	— — —	— — —	— — —
00 30 00	00 02 40	00 30 00	Б. И. ←(PC)
00 30 02	00 02 05	00 30 02	Б. И.

Карта выполнения инструкции [RTS R5]



Карта памяти

Адрес	Содержимое	Адрес	Содержимое
(SP)→ 00 11 70	00 01 32	00 11 70	XX XX XX
00 11 72	XX XX XX	00 11 72	Б. И. ←(SP)
— — —	— — —	— — —	— — —
00 20 04	XX XX XX	00 20 04	Б. И. ←(PC)
— — —	— — —	— — —	— — —
00 30 00	00 02 40	00 30 00	Б. И.
(PC)→ 00 30 02	00 02 05	00 30 02	Б. И.
— — —	— — —	— — —	— — —

Инструкция выхода к подпрограмме

Мнемонический код: JSR PC, назн
Восьмеричный машинный код: 0047 назн
OP

Выполнение:

1-й цикл извлечения: (PC) → РАП
(PC) + 2 → PC
Код инструкции JSR → РИ

2-й цикл извлечения: (PC) → РАП
(PC) + 2 → PC
OP → ЦП

Цикл выполнения: (PC) → –(SP)
OP + $(PC)_{скор}$ → PC

Обратите внимание, что эти действия очень похожи на описанные выше действия, за исключением того, что значение адреса возврата к главной программе в данном случае проталкивается в стек, а не помещается в регистр R5.

Инструкция возврата

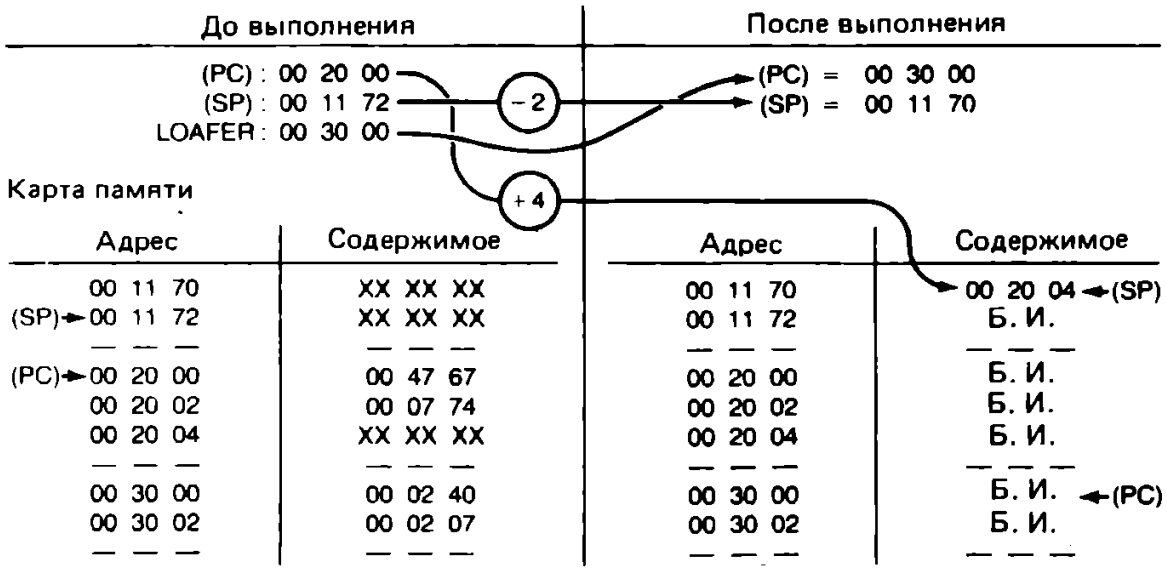
Мнемонический код: RTS PC
Восьмеричный машинный код: 00020 7

Выполнение этой инструкции осуществляется выталкиванием значения адреса возврата, сохраненного в стеке, из вершины стека назад в регистр PC. Например:

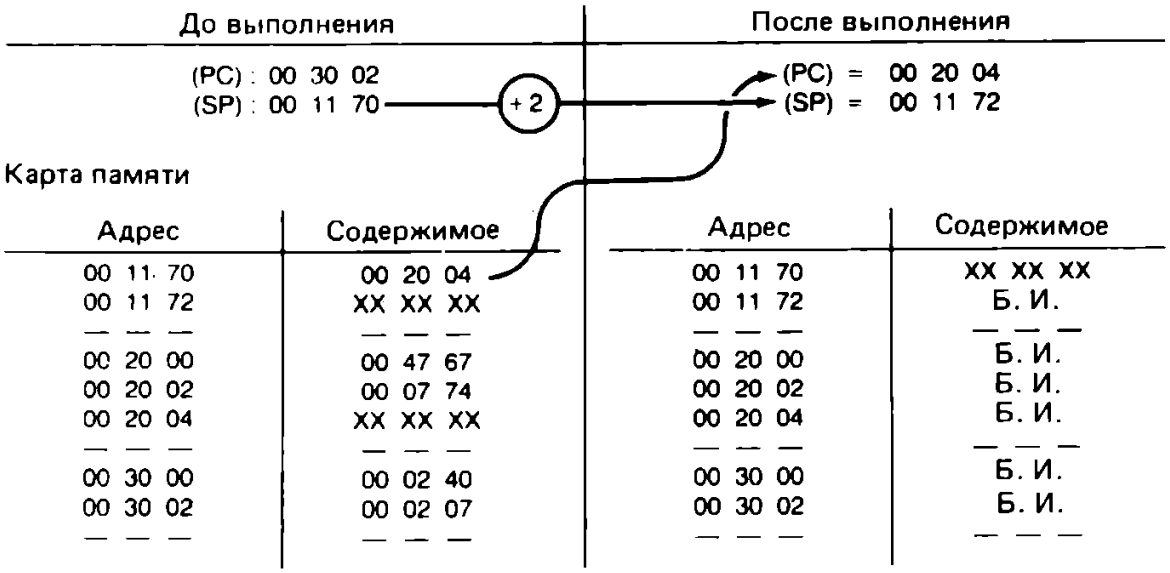
Исходная программа:

Символьный адрес	Числовой адрес	Исходный код
STACK	00 11 70	— — —
	00 11 72	— — —
	— — — —	— — —
	00 20 00	JSR PC, LOAFER
	— — — —	— — —
	— — — —	— — —
LOAFER	00 30 00	NOP
	00 30 02	RTS PC

Карта выполнения [JSR PC, LOAFER]



Карта выполнения [RTS PC]



Инструкция связывания сопрограмм

Это специальная инструкция для связывания сопрограмм, а неглавной программы и подпрограммы. Различие состоит в том, что в случае сопрограмм обе подпрограммы равноправны. Каждая из них может вызывать другую, тогда как при главной программе и подпрограмме процесс вызова – односторонний. Поскольку подпрограммы, используемые как сопрограммы, равноправны, инструкция возврата в них не нужна. Вместо возврата одна подпрограмма просто вызывает другую и наоборот. Другими словами, инструкции вызова и возврата идентичны:

Мнемонический код: JSR PC, @(SP)+
Восьмеричный машинный код: 004 7 36

Обратите внимание, что для этой инструкции требуется только одно слово памяти, причем в поле операнда назначения используется косвенная адресация с автоувеличением или режим адресации 3. Опыт показывает, что для новичков понимание этой инструкции сопряжено с известными трудностями. Проанализируем ее.

Прежде всего напомним, что, каким бы ни было содержимое поля операнда назначения, ЦП в конце концов поместит в регистр PC адресное значение. Давайте рассмотрим предыдущий пример, в котором был исходный код JSR PC, LOAFER. В какой-то момент ЦП помещает в регистр PC адресное значение LOAFER = 3000. Аналогично нам надо определить адресное значение для операнда @(SP)+ и поместить его в регистр PC. Теперь нам уже известно, что SP – это регистр R6, (SP) – содержимое регистра R6, @(SP) – содержимое слова стековой памяти, адрес которого равен содержимому указателя стека. Вспоминаем, что этот режим – режим адресации с двойным уровнем косвенности. При оценке @(SP)+ ЦП помещает в регистр PC верхний элемент стека и автоматически увеличивает (SP) на 2. В то же время ЦП помещает в стек скорректированное содержимое регистра (PC). Следовательно, эта инструкция осуществляет обмен скорректированного содержимого регистра PC с верхним элементом стековой памяти. Для иллюстрации работы воспользуемся двумя фиктивными подпрограммами: LOAFER1 и LOAFER 2:

Символьный адрес	Числовой адрес	Исходный код	Комментарий
INITIAL	00 11 72	MOV #LOAFER2, –(SP)	; ЗАМЕЧАНИЕ 1
LOAFER1	00 11 74	NOP	
	00 11 76	JSR PC, @(SP)+	; ЗАМЕЧАНИЕ 2
	00 12 00	NOP	
	00 12 04	JSR PC, @(SP) +	
	00 12 06	NOP	
	00 12 10	JSR PC, @(SP)+	
	00 12 12	HALT	
LOAFER2	00 12 14	NOP	
	00 12 16	JSR PC, @(SP)+	; ЗАМЕЧАНИЕ 3
	00 12 20	NOP	
	00 12 22	JSR PC, @(SP)+	
	00 12 24	NOP	
	00 12 26	JSR PC, @(SP)+	

Замечания:

1. Верхний элемент стека проинициализирован значением 00 12 14, т. е. (SP) = = 00 12 14 – адресному значению метки LOAFER2.
2. Эта инструкция получает адресное значение метки LOAFER2, 00 12 14, помещенное в стек ранее, и помещает его в регистр PC; тем временем в стек проталкивается скорректированное содержимое регистра (PC), равное 00 12 14.
3. В данном случае происходит обмен местами скорректированного содержимого регистра (PC) и содержимого вершины стека, и программа, таким образом, переходит назад к значению 00 12 00 и т.д.

Последовательность выполнения этих двух сопрограмм показана на рис. 5.3.

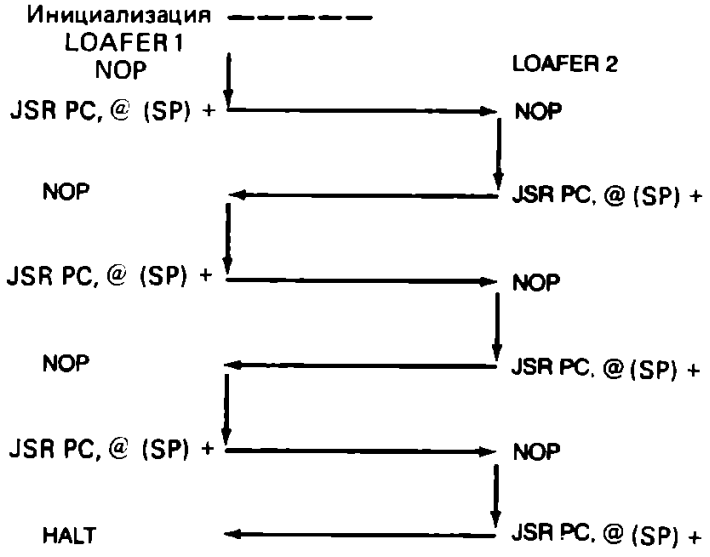


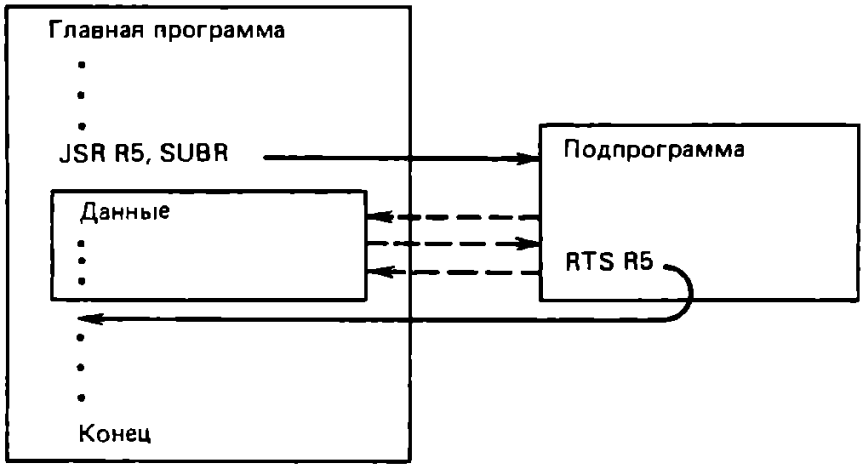
Рис. 5.3. Последовательность выполнения сопрограмм

5.2. ПЕРЕСЫЛКА ПАРАМЕТРОВ ИЛИ АРГУМЕНТОВ

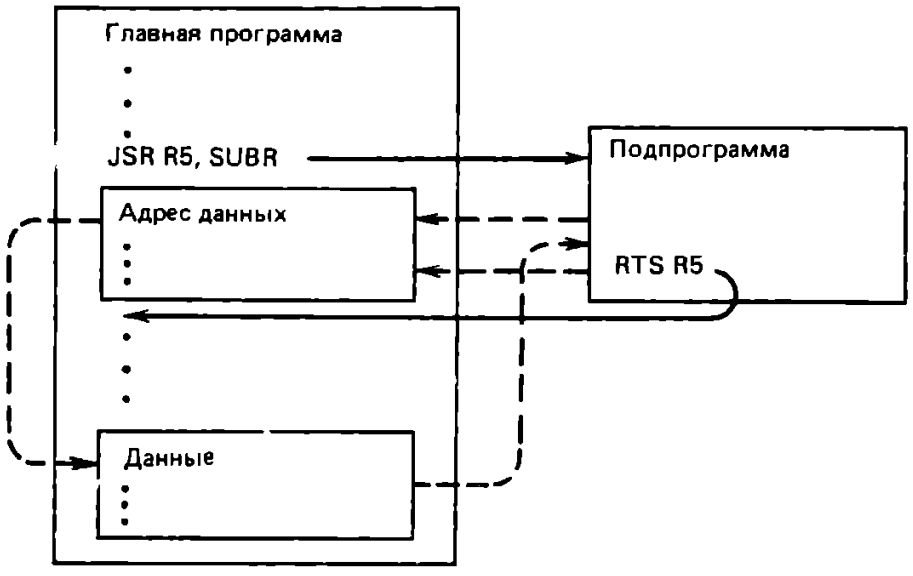
Итак, средства связывания главной программы и подпрограммы выбраны. Следующим шагом надо осуществить выбор подходящего способа передачи данных из главной программы в подпрограмму и получения назад результатов. Этой цели можно достичь множеством способов. В этой главе мы опишем три наиболее популярных из них.

РАЗМЕЩЕНИЕ ПАРАМЕТРОВ ПОСЛЕ ИНСТРУКЦИИ ПЕРЕХОДА К ПОДПРОГРАММЕ

При этом способе мы должны размещать данные или адреса данных непосредственно после инструкции JSR в главной программе. Тогда ответственность за использование связующего регистра для получения данных и помещения назад результатов лежит на подпрограмме. Такой способ передачи изображен на рис. 5.4, *а* и *б*. На первом из них показана ситуация, когда после инструкции помещаются данные; на втором, когда после инструкции размещаются не данные, а их адреса. На этих рисунках сплошными линиями со стрелками отмечено связывание двух программ с помощью инструкций, штриховыми линиями помечены пути передачи данных или параметров. Следующие примеры иллюстрируют такой способ передачи данных.



а) Прямая передача данных



б) Передача адресов в качестве параметров

Рис. 5.4. Размещение параметров после инструкции JSR

Пример 1:

```
START:
        .
        .
        .
        JSR     R5,SUM
DATA:   .WORD   3,4
TOTAL:  .BLKW   1
        .
        .
        .
QUIT:   HALT
SUM:    MOV     (R5)+,R0        ;3 -) R0
        ADD     (R5)+,R0        ;4 + (R0) -) R0
        MOV     R0,(R5)+        ;РЕЗУЛЬТАТ -) TOTAL
        RTS     R5
        .END    START
```

В этом примере главная программа вызывает подпрограмму SUM, поставляет ей данные, 3 и 4, и требует вычислить сумму указанных данных и поместить результат в слово памяти с символьным адресом TOTAL.

Давайте разместим рядом друг с другом нашу программу и рис. 5.4, *а* и выполним программу, полагая, что адресное значение инструкции JSR равно 2000.

1. Выполнение инструкции JSR R5, SUM заключается в следующем:

(R5) → стек
(PC) + 4 или #DATA → R5
#SUM → PC

2. Сплошная линия на рис. 5.4, *а* показывает, что происходит переход к выполнению первой инструкции подпрограммы. Здесь (R5) указывает на первый элемент данных – на 3. Таким образом осуществляется операция 3 → R0, что отражено штриховой линией; затем в регистре R5 оказывается значение (R5) + 2, в результате чего в конце выполнения регистр R5 указывает на второй элемент данных – число 4, поскольку теперь (R5) = = #DATA. Напоминаем, что в предыдущей главе мы определили:

#DATA ≙ адресное значение символьного адреса DATA..

3. После выполнения следующей инструкции имеем:

4 + (R0) = 4 + 3 = 7 → R0
(R5) + 2 = #DATA + 4 = #TOTAL → R5

То есть в конце выполнения этой инструкции (R5) указывает на символьный адрес TOTAL.

4. Следующая инструкция копирует содержимое регистра R0, равное сумме чисел 3 и 4, в ячейку TOTAL. Затем она автоматически увеличивает (R5), в результате чего (R5), равное #TOTAL + 2, указывает теперь на ячейку, в которой главная программа хочет получить результат операции.

5. Следующей выполняется инструкция RTS:

(R5) или #TOTAL + 2 → PC
верхний элемент стека → R5

Напоминаем, что старое содержимое регистра R5 было сохранено в стеке инструкцией JSR.

6. Главная программа возобновляет свою работу по адресу #TOTAL + 2.

Пример 2. В этом примере после инструкции JSR помещаются адреса данных:

```
START:  .
        .
        .
        .
        JSR     R5,SUM
PARAM:  .WORD   A,B
TOTAL:  .BLKW   1
        .
        .
        .
QUIT:   HALT
A:      .WORD   3
B:      .WORD   4
SUM:    MOV     @(R5)+,R0
        ADD     @(R5)+,R0
        MOV     R0,(R5)+
        RTS     R5
        .END    START
```

Разница между примерами 1 и 2 состоит в том, что в последнем используется режим косвенной адресации с автоувеличением, так что с помощью связующего регистра R5 могут быть получены данные, находящиеся соответственно в ячейках A и B. Поскольку предполагается, что результат должен оказаться в ячейке TOTAL, как и в примере 1, то в этом примере также используется режим адресации с автоувеличением.

Обратите внимание, что этот способ передачи параметров довольно прост. ЦП автоматически сохраняет (R5), что дает возможность подпрограмме "заимствовать" регистр R5 и использовать его в качестве связующего. У этого способа, однако, есть и недостаток – смешивание данных с инструкциями.

Следует обратить внимание еще на один важный момент, состоящий в том, что подпрограмма в обоих примерах "заимствует" под данные также и регистр R0. Если главная программа хранит в регистре R0 важную информацию, это может вызвать определенные трудности, поскольку старое содержимое этого регистра будет потеряно после входа в подпрограмму. Чтобы сделать подпрограмму пригодной для любой главной программы, имеет смысл сохранять (R0) в стеке перед его "заимствованием" подпрограммой и восстанавливать старое содержимое этого регистра ближе к концу подпрограммы. То есть подпрограмма суммирования может быть модифицирована следующим образом:

```
SUM:    MOV     R0,-(SP)        ; ПОМЕСТИТЬ (R0) В СТЕК
        MOV     (R5)+,R0
        ADD     (R5)+,R0
        MOV     R0,(R5)+
        MOV     (SP)+,R0
        RTS     R5
```

Если предположить, что до выполнения инструкции JSR (SP) равно 1172, то карта памяти стека до, во время и после выполнения подпрограммы будет иметь вид:

До выполнения	Во время выполнения	После выполнения
(R0) = старое (R0)	(R0) – переменное	(R0) = старое (R0)
(R5) = старое (R5)	(R5) – регистр связи	(R5) = старое (R5)

Адрес	Содержимое	Адрес	Содержимое	Адрес	Содержимое
00 10 62	XX XX XX	00 10 62		00 10 62	XX XX XX
00 10 64	XX XX XX	00 10 64		00 10 64	XX XX XX
00 10 66	XX XX XX	00 10 66	Старое (R0)	00 10 66	XX XX XX
00 11 70	XX XX XX	00 11 70	Старое (R5)	00 11 70	XX XX XX
00 11 72	XX XX XX	00 11 72	XX XX XX	00 11 72	XX XX XX

РАЗМЕЩЕНИЕ ПАРАМЕТРОВ В СПЕЦИАЛЬНО ВЫДЕЛЕННОЙ ОБЛАСТИ

При этом способе передачи данные размещаются в специально выделенной области памяти и с инструкциями не смешиваются. Пару, обеспечивающую связывание главной программы и подпрограммы, в данном случае составляют инструкции JSR PC, SUBR и RTS PC. Для связи с данными используется какой-либо регистр общего назначения, например R5. Выполняемые действия отражены на рис. 5.5. Штриховые линии мы снова используем для обозначения путей данных, а сплошные линии – для связывания программ. Ниже приводятся два примера: первый описывает общую работу, второй является числовым.

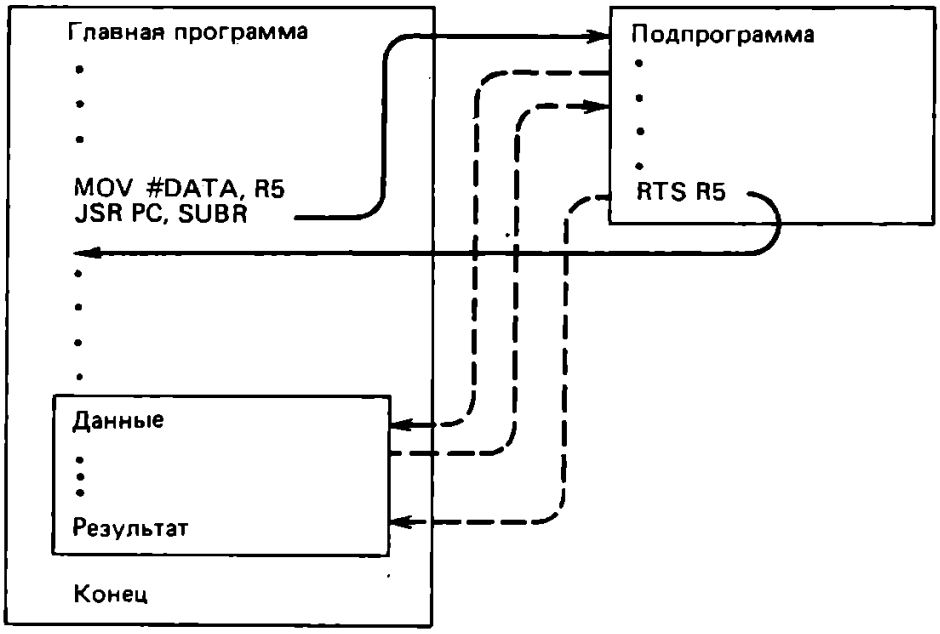


Рис. 5.5. Размещение параметров в специально выделенной области

Пример 1:

```
START:  .
        .
        .
        .
INIT:   MOV     #DATA,R5        ;(R5) = АДРЕС ДАННЫХ
        JSR     PC,SUBR         ;(PC) --> СТЕК, #SUBR --> PC
        .
        .
        .
QUIT:   HALT
DATA:   D1
        D2
        .
        .
        .
        DN
```

```
SUBR:   MOV     (R5)+,НАЗН      ;ОБЛАСТЬ НАЗНАЧЕНИЯ,
        .                       ;КУДА ДОЛЖЕН БЫТЬ СКОПИРОВАН
        .                       ;МАССИВ ДАННЫХ
        MOV     (R5)+,НАЗН
        .
        .
        RTS     PC
        .END    START
```

Пример 2. В этом примере мы воспользуемся подпрограммой SUM для определения полной суммы для массива данных, предполагая, что полная сумма не превышает 32 000; т. е. переполнения не произойдет. Предположим, кроме того, что первый элемент массива данных представляет число элементов или размер массива и равен, например 10.

```
START:  MOV     #DATA,R5        ;(R5) -> ДАННЫЕ
        JSR     PC,SUM          ;(PC)+4 -> PC, (PC) -> СТЕК, #SUM -> PC
QUIT:   HALT
DATA:   .WORD   10              ;РАЗМЕР МАССИВА ДАННЫХ
        .WORD   1,-2,3,-4,5,-6,7,10   ;ДАННЫЕ
TOTAL:  .BLKW   1               ;МЕСТО ДЛЯ РЕЗУЛЬТАТА
SUM:    CLR     R0              ;R0 БУДЕТ АККУМУЛЯТОРОМ
        MOV     (R5)+,R1        ;R1 БУДЕТ СЧЕТЧИКОМ ДАННЫХ
LOOP:   ADD     (R5)+,R0        ;СЛОЖЕНИЕ ДАННЫХ
        SOB     R1,LOOP         ;УМЕНЬШИТЬ R1 НА 1; ЕСЛИ НЕ 0,
                                ;ПЕРЕЙТИ К LOOP
        MOV     R0,TOTAL        ;РЕЗУЛЬТАТ -> TOTAL
        RTS     PC              ;ВЕРХНИЙ ЭЛЕМЕНТ СТЕКА -> PC
        .END    START
```

Файл листинга этой программы и результат ее выполнения показаны на рис. 5.6.

```
 1                      ;ПРИМЕР
 2                      ;
 3                      ;
 4                      ;
 5                      ;
 6                      ;ПРИМЕР 2:
 7                      ;
 8                      ;
 9 000000  012705  START:  MOV     #DATA,R5        ;(R5) -> ДАННЫЕ
           000012'
10 000004  004767         JSR     PC,SUM          ;(PC)+4->PC,(PC)->СТЕК,#SUM->PC
           000026
11 000010  000000  QUIT:   HALT
12 000012  000010  DATA:   .WORD   10              ;РАЗМЕР МАССИВА ДАННЫХ
13 000014  000001         .WORD   1,-2,3,-4,5,-6,7,10   ;ДАННЫЕ
   000016  177776
   000020  000003
   000022  177774
   000024  000005
   000026  177772
   000030  000007
   000032  000010
14 000034         TOTAL:  .BLKW   1               ;МЕСТО ДЛЯ РЕЗУЛЬТАТА
15 000036  005000 SUM:    CLR     R0              ;R0 БУДЕТ АККУМУЛЯТОРОМ
16 000040  012501         MOV     (R5)+,R1        ;R1 БУДЕТ СЧЕТЧИКОМ ДАННЫХ
17 000042  062500 LOOP:   ADD     (R5)+,R0        ;СЛОЖЕНИЕ ДАННЫХ
18 000044  077102         SOB     R1,LOOP         ;УМЕНЬШИТЬ R1 НА 1; ЕСЛИ НЕ 0,
19                                                ;ПЕРЕЙТИ К LOOP
```

```
20 000046  010067          MOV     R0,TOTAL        ;РЕЗУЛЬТАТ -> TOTAL
           177762
21 000052  000207          RTS     PC              ;ВЕРХНИЙ ЭЛЕМЕНТ СТЕКА -> PC
22         000000'         .END    START

РЕЗУЛЬТАТ ВЫПОЛНЕНИЯ: ЯЧЕЙКА TOTAL СОДЕРЖИТ 14 (ВОСЬМЕРИЧНОЕ ЧИСЛО)
```

Рис. 5.6. Пример передачи параметров при размещении данных в выделенной области

ПЕРЕСЫЛКА ПАРАМЕТРОВ ЧЕРЕЗ СТЕКОВУЮ ПАМЯТЬ

Этот способ наиболее подходит в том случае, когда нужны вложенные подпрограммы или требуется обслуживать прерывания. В основном пересылка параметров через стековую память осуществляется в несколько этапов: 1. Главная программа копирует данные в стековую память. 2. Подпрограмма извлекает данные из стека и помещает в стек результат. 3. Главная программа копирует результат из стека по адресу назначения. При таком подходе подпрограмме не надо заботиться о том, где искать данные, поскольку они всегда находятся в стеке. На рис. 5.7 показана пересылка параметров этим способом. Обратите внимание, что сплошными линиями показано связывание программ, а штриховыми – пути переноса данных.

В этом случае мы должны до выполнения инструкции JSR протолкнуть все данные в стек, чтобы подпрограмма смогла с ними работать, а после выполнения инструкции RTS нам нужно вытолкнуть из стека результат и поместить его по адресу назначения. Во время работы подпрограмма извлекает данные из стека и оставляет в нем результат. Этот способ кажется довольно простым, однако некоторые детали требуют рассмотрения. Вспоминаем, что инструкция JSR помещает в стек скорректированное содержимое регистра PC для обеспечения возврата из подпрограммы. Но мы здесь, кроме того, проталкиваем в стек данные и выталкиваем их из него. Очевидно, если мы не будем достаточно внимательны, то может возникнуть путаница. Для прояснения этого воспользуемся примером, аналогичным примеру 2 из предыдущего раздела.

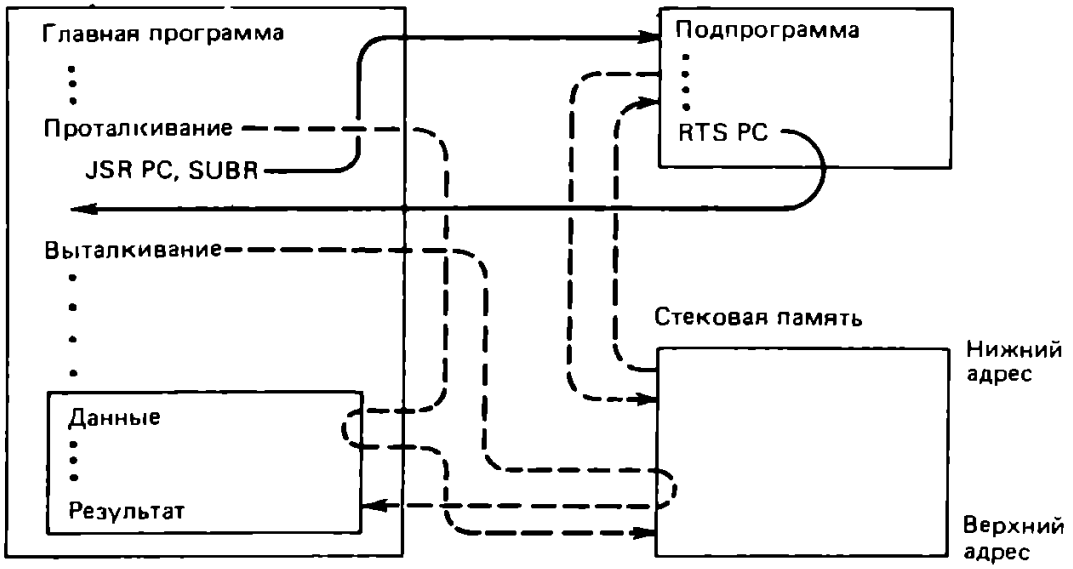


Рис. 5.7. Пересылка параметров через стек

Пример 1:

```
1) START: MOV   SP,R4          ;СКОПИРОВАТЬ (SP) В R4
2)        TST   -(R4)          ;(R4) - 2 -> R4
3)        MOV   #DATA,R5       ;R5 - УКАЗАТЕЛЬ ДАННЫХ
4)        MOV   (R5)+,R3       ;R3 - СЧЕТЧИК ДАННЫХ ДЛЯ МАССИВА
5)        MOV   R3,R2          ;R2 - СЧЕТЧИК ДАННЫХ ДЛЯ СТЕКА
```

```
6)PUSH: MOV     (R5)+,-(R4)     ;ПРОТОЛКНУТЬ ДАННЫЕ В СТЕК
7)      SOB     R3,PUSH         ;ЕСЛИ НЕ КОНЕЦ ПРОТАЛКИВАНИЯ ДАННЫХ,
                                ;ПЕРЕЙТИ К ПРОТАЛКИВАНИЮ. ЕСЛИ КОНЕЦ,
                                ;(R5) УКАЗЫВАЕТ НА ПОЛНУЮ СУММУ И
                                ;НАДО ВЫПОЛНИТЬ СЛЕДУЮЩУЮ ИНСТРУКЦИЮ
8)CALLSB: JSR   PC,SUM          ;(PC)+4 -> PC, (PC) -> СТЕК, #SUM -> PC
9)      MOV (R4),TOTAL          ;РЕЗУЛЬТАТ -> TOTAL
10) QUIT: HALT
11) DATA: .WORD N
12)     .WORD   D1,D2,...,DN
13) TOTAL: .BLKW 1
14) SUM: CLR    R0              ;R0 БУДЕТ АККУМУЛЯТОРОМ
15) LOOP: ADD   (R4)+,R0,       ;ВЫТОЛКНУТЬ ДАННЫЕ ИЗ СТЕКА И СЛОЖИТЬ ИХ
16)     SOB     R2,LOOP         ;КОНЕЦ СЛОЖЕНИЯ?
17)     MOV     R0,-(R4)        ;РЕЗУЛЬТАТ -> СТЕК
18)     RTS     PC
        .END    START
```

Чтобы избежать путаницы в стеке между данными и скорректированным содержимым регистра PC, обеспечивающим возврат в главную программу из подпрограммы, мы в первую очередь копируем значение содержимого регистра SP в регистр R4, который будет использоваться как указатель данных в стеке. Затем мы уменьшаем (R4) на 2, чтобы пропустить одно слово, которое вскоре будет использовано для запоминания скорректированного содержимого регистра PC. После этого мы назначаем регистр R5 в качестве указателя массива данных, а регистры R3 и R2 – в качестве счетчиков данных соответственно для первоначального массива данных и для стека. Завершив проталкивание данных в стек, мы готовы вызвать подпрограмму SUM. Подпрограмма суммирует данные, находящиеся в стеке, и помещает в стек результат. После возврата из подпрограммы главная программа извлекает из стека результат и помещает его по адресу назначения TOTAL.

На рис. 5.8 представлена карта памяти стека, показывающая его состояние до, во время и после выполнения программы в предположении, что указатель стека SP был инициализирован значением 1172. В первой колонке дано содержимое регистров после завершения инструкции 5. Вторая колонка показывает состояние стековой памяти после инструкции 7. Здесь программа использует указатели R5 и R4 для копирования данных в стек из первоначального массива. Третья колонка таблицы показывает, что инструкция 8 помещает скорректированное содержимое регистра PC в стек по адресу 00 11 70 с помощью указателя стека и программа переходит к подпрограмме, начинающейся с символьного адреса SUM. Четвертая колонка показывает, что в конце выполнения подпрограммы инструкция 17 с помощью регистра R4 переносит полную сумму из регистра R0 в ячейку 00 11 66. Из пятой колонки видно, что инструкция 18 помещает в регистр PC скорректированное содержимое регистра PC из ячейки 00 11 70, после

Состояние регистров после выполнения инструкции 5	Состояние стековой памяти									
	После выполнения инструкции 7		После выполнения инструкции 8		После выполнения инструкции 17		После выполнения инструкции 18		После выполнения инструкции 9	
(SP) = 1172	Адрес	Содержимое	Адрес	Содержимое	Адрес	Содержимое	Адрес	Содержимое	Адрес	Содержимое
(R4) = 1170		d*n*		d*n*		xx xx xx		xx xx xx		xx xx xx
(R5) = #DATA	•	•	•	•	•	•	•	•	•	•
(R3) = *n*	•	•	•	•	•	•	•	•	•	•
(R2) = *n*	•	•	•	•	•	•	•	•	•	•
	•	•	•	•	•	•	•	•	•	•
	•	•	•	•	00 11 64	xx xx xx	•	•	•	•
	00 11 66	d1	00 11 66	d1	00 11 66	Результат	00 11 66	Результат	00 11 66	xx xx xx
	00 11 70	xx xx xx	00 11 70	#CALLSB + 4	00 11 70	#CALLSB + 4	00 11 70	xx xx xx	00 11 70	xx xx xx
	00 11 72	xx xx xx	00 11 72	xx xx xx	00 11 72	xx xx xx	00 11 72	xx xx xx	00 11 72	xx xx xx

Рис. 5.8. Состояние стека во время выполнения программы

```
 1                      ;ПРИМЕР
 2                      ;
 3                      ;
 4                      ;
 5                      ;ПРИМЕР 2:
 6 000000  010604  START:  MOV     SP,R4           ;СКОПИРОВАТЬ (SP) В R4
 7 000002  005744          TST     -(R4)           ;(R4) - 2 -> R4
 8 000004  012705          MOV     #DATA,R5        ;R5 - УКАЗАТЕЛЬ ДАННЫХ
           000032'
 9 000010  012403          MOV     (R5)+,R3        ;R3 - СЧЕТЧИК ДАННЫХ ДЛЯ МАССИВА
10 000012  010302          MOV     R3,R2           ;R2 - СЧЕТЧИК ДАННЫХ ДЛЯ СТЕКА
11 000014  012544  PUSH:   MOV     (R5)+,-(R4)     ;ПРОТОЛКНУТЬ ДАННЫЕ В СТЕК
12 000016  077302          SOB     R3,PUSH         ;ЕСЛИ НЕ КОНЕЦ ПРОТАЛКИВАНИЯ ДАННЫХ,
13                                                 ;ПЕРЕЙТИ К ПРОТАЛКИВАНИЮ. ЕСЛИ КОНЕЦ,
14                                                 ;(R5) УКАЗЫВАЕТ НА ПОЛНУЮ СУММУ И
15                                                 ;НАДО ВЫПОЛНИТЬ СЛЕДУЮЩУЮ ИНСТРУКЦИЮ
16 00020   004767  CALLSB:JSR     PC,SUM          ;(PC)+4->PC,(PC)->СТЕК,#SUM->PC
           000022
17 000024  011467          MOV (R4),TOTAL          ;РЕЗУЛЬТАТ -> TOTAL
           000014
18 000030  000000  QUIT:   HALT
19 000032  0000006 DATA:   .WORD   10
20 000034  0000006         .WORD   1,-2,3,-4,5,-6,7,10
   000036  0000006
   000040  0000006
   000042  0000006
21 000044          TOTAL:  .BLKW 1
22 000046  005000  SUM:    CLR     R0              ;R0 БУДЕТ АККУМУЛЯТОРОМ
23 000050  062400  LOOP:   ADD     (R4)+,R0        ;ВЫТОЛКНУТЬ ДАННЫЕ ИЗ СТЕКА И СЛОЖИТЬ
24 000052  077202          SOB     R2,LOOP         ;КОНЕЦ СЛОЖЕНИЯ?
25 000054  010044          MOV     R0,-(R4)        ;РЕЗУЛЬТАТ -> СТЕК
26 000056  000207          RTS     PC
27         000000'         .END    START

   РЕЗУЛЬТАТ ВЫПОЛНЕНИЯ: ЯЧЕЙКА TOTAL СОДЕРЖИТ 14 (ВОСЬМЕРИЧНОЕ ЧИСЛО)
```

Рис. 5.9. Пример передачи параметров через стек

чего программа возвращается к инструкции 9. Шестая колонка показывает результат выполнения инструкции 9. В конце стек возвращается к первоначальному состоянию.

Числовой пример, иллюстрирующий пересылку параметров или данных через стек, представлен на рис. 5.9. Обратите внимание, что при этом способе передачи параметров подпрограмма имеет дело только со стековой памятью.

5.3. ОБЩИЙ ФОРМАТ ДЛЯ ДОКУМЕНТИРОВАНИЯ ПОДПРОГРАММЫ И ГЛАВНОЙ ПРОГРАММЫ

До сих пор мы излагали основные принципы методов программирования подпрограмм, причем для удобства главная программа в наших примерах содержала подпрограмму. Однако на практике главная программа, как правило, подпрограмму или подпрограмм не содержит. Более того, подпрограммы нередко пишутся другими программистами и могут быть доступны только в виде исходных или объектных файлов.

Если мы хотим воспользоваться чужими подпрограммами, то должны знать: 1) имена точек входа или таблицы стартовых адресов подпрограмм; 2) используемые подпрограммами связующие регистры; 3) способы передачи параметров, применяемые в подпрограммах. Следовательно, было бы вполне разумно, если бы мы как для подпрограмм, так и для главных программ придерживались общих форматов. Дадим описание формата, обычно используемого при работе с операционной системой RSX-11M.

ФОРМАТ ПОДПРОГРАММЫ

Воспользуемся типичной подпрограммой, использующей вышеописанный способ передачи параметров. Типичный формат такой подпрограммы будет следующим:

```
        ;ВЕКТОР ВЫЗОВА ПОДПРОГРАММЫ:
        ;JSR     R5,SUM
        ;.WORD   N - РАЗМЕР МАССИВА ДАННЫХ
        ;.WORD   DATA - НАЧАЛЬНЫЙ АДРЕС МАССИВА ДАННЫХ
        ;КРОМЕ ТОГО, ПОДПРОГРАММА ИСПОЛЬЗУЕТ РЕГИСТР R0
        ;КАК АККУМУЛЯТОР, РЕГИСТР R1 - КАК СЧЕТЧИК ДАННЫХ,
        ;РЕГИСТР R2 - КАК УКАЗАТЕЛЬ ДАННЫХ.
        .GLOBL  SUM
SUM:    CLR     R0              ;ИНИЦИАЛИЗАЦИЯ АККУМУЛЯТОРА
        MOV     (R5)+,R1        ;ИНИЦИАЛИЗАЦИЯ СЧЕТЧИКА ДАННЫХ
        MOV     (R5)+,R2        ;#DATA -> R2
LOOP:   ADD     (R2)+,R0        ;R0 - АККУМУЛЯТОР
        SOB     R1,LOOP
        RTS     R5
        .END    SUM
```

Обратите внимание, что мы используем точку с запятой для начала каждой строки комментариев. Это необходимо для того, чтобы показать машине, что последующие символы в строке представляют просто комментарии, служащие для целей документирования программы. Они нужны лишь для удобства пользователя, а машина должна их игнорировать. Ассемблер не будет ассемблировать строки комментариев, начинающиеся с точки с запятой. Псевдоинструкция или директива .GLOBL – это еще одно важное средство взаимодействия с операционной системой, позволяющее нам показать точку входа или метку стартового адреса подпрограммы, которая сообщает машине и пользователю, что SUM – это метка глобального адреса, предназначенная для связывания главной программы с подпрограммой. В данном случае SUM является глобальной меткой, а LOOP – локальной.

ФОРМАТ ГЛАВНОЙ ПРОГРАММЫ

Задокументировав подпрограмму указанным выше способом, мы можем записать главную программу так:

```
;ЭТО ГЛАВНАЯ ПРОГРАММА, КОТОРАЯ ВЫЗЫВАЕТ ПОДПРОГРАММУ SUM
;ДЛЯ ВЫЧИСЛЕНИЯ ПОЛНОЙ СУММЫ МАССИВА ДАННЫХ ИЗ N ЭЛЕМЕНТОВ
;И ПОМЕЩАЕТ РЕЗУЛЬТАТ ПО АДРЕСУ НАЗНАЧЕНИЯ TOTAL.
        .GLOBL  SUM
DATA:   .WORD   D1,D2,...,DN
TOTAL:  .BLKW   1
START:  JSR     R5,SUM          ;ПЕРЕЙТИ К ПОДПРОГРАММЕ
        .WORD   N               ;ПЕРЕДАЧА РАЗМЕРА МАССИВА
        .WORD   DATA            ;ПЕРЕДАЧА НАЧАЛЬНОГО АДРЕСА
                                ;МАССИВА ДАННЫХ
        MOV     R0,TOTAL        ;РЕЗУЛЬТАТ -> TOTAL
        HALT
        .END    START
```

Обратите внимание, что здесь мы вновь использовали псевдоинструкцию .GLOBL для указания ассемблеру, что метка SUM является символом глобального адреса, который хотя и не определяется в главной программе, но определяется в подпрограмме. При использовании директивы .GLOBL как в главной программе, так и в подпрограмме построитель задач (ТКВ) получает возможность разрешить эту "неясность". В противном случае ассемблеру пришлось бы рассматривать ее как ошибку неопределенного символа, поскольку метка SUM в главной программе не определяется.

Имеет смысл отметить и то, что в этом примере мы перенесли секцию данных в начало программы. Такая конфигурация обладает некоторым достоинством в отношении перезапуска программы. поскольку после выполнения программа обычно останавливается на следующем слове после инструкции останова HALT. Если инструкцию HALT заменить на псевдоинструкцию .EXIT, то (PC) будет указывать на метку START. Однако, если за псевдоинструкцией .EXIT следует секция данных, ЭВМ не будет знать, что ей делать дальше.

Из документации подпрограммы мы узнаем также, что в процессе работы она использует регистры R0, R1, R2 и R5. Нам известно, что инструкция JSR R5, SUM автоматически сохраняет старое содержимое регистра (R5). Если старое содержимое регистров R0, R1 и R2 важно для главной программы, то перед вызовом подпрограммы она должна сохранять (R0), (R1) и (R2) в стеке и восстанавливать их после завершения подпрограммы. В нашем примере главной программе сохранять содержимое этих регистров не нужно, поэтому соответствующие действия здесь опущены.

5.4. СВЯЗЫВАНИЕ ПОДПРОГРАММ С ГЛАВНОЙ ПРОГРАММОЙ

Чтобы собрать образ памяти выполняемой задачи или файл типа TSK, есть два способа скомпоновать подпрограммы с главной программой; они зависят от типа доступного подпрограммного файла или файлов. Если доступен только исходный файл подпрограммы, то подпрограмму и главную программу можно одновременно ассемблировать с помощью следующей команды:

≥MAC <HOST.OBJ>,<HOST.LST> = <SUBR.MAC>,<HOST.MAC>

В результате ее выполнения мы получим как объективный файл, так и файл листинга скомбинированной программы. Для генерации файла задачи типа TSK надо просто воспользоваться следующей командой:

≥TKB HOST = HOST

И мы получим выполнимый файл HOST.TSK, пригодный к загрузке и запуску с помощью соответствующей команды (LGO).

Если же доступен объектный файл подпрограммы, то мы можем ассемблировать с помощью ассемблера одну главную программу, чтобы получить объектный файл <HOST.OBJ>. Затем можно воспользоваться командой TKB для создания файла типа TSK скомпонованной программы:

≥TKB HOST = <SUBR.OBJ>,<HOST.OBJ>

Эту процедуру проиллюстрируют два примера.

Пример 1. Нужно написать главную программу, которая будет вызывать подпрограмму по имени SCREEN. Главная программа предназначена для выбора группы мужчин из массива кандитатов в футбольную команду по их весам. Должны выбираться только те кандидаты, которые весят от 180 до 220 фунтов включительно. Если кандидат выбирается, то его данные сохраняются; если нет, то его первоначальные данные заменяются нулевым весом. Сначала нам необходимо внимательно изучить подпрограмму SCREEN и определить, нужна ли ее модификация. Предположим, что информация о подпрограмме дана нам в следующем виде:

```
[ПОДПРОГРАММА]
;ЭТА ПОДПРОГРАММА ПРОСМАТРИВАЕТ МАССИВ ДАННЫХ ИЗ N ЭЛЕМЕНТОВ.
;ЕСЛИ ДАННЫЕ ВЫХОДЯТ ЗА ПРЕДЕЛЫ ЗАДАННОГО ДИАПАЗОНА, ОНИ
;ЗАМЕНЯЮТСЯ НУЛЯМИ. В ПРОТИВНОМ СЛУЧАЕ НИКАКИХ МОДИФИКАЦИЙ
;НЕ ПРОИЗВОДИТСЯ. МАССИВ ПРЕДПОЛАГАЕТСЯ УПОРЯДОЧЕННЫМ ТАКИМ ОБРАЗОМ,
;ЧТО В ПЕРВЫХ ТРЕХ ЭЛЕМЕНТАХ РАСПОЛАГАЮТСЯ СООТВЕТСТВЕННО РАЗМЕР
;МАССИВА ДАННЫХ, ВЕРХНИЙ ПРЕДЕЛ И НИЖНИЙ ПРЕДЕЛ, А СЛЕД ЗА НИМИ
;РАЗМЕЩЕНЫ N ЭЛЕМЕНТОВ ДАННЫХ, ПОДЛЕЖАЩИЕ ПРОВЕРКЕ.
```

```
;КРОМЕ ТОГО, ПОДПРОГРАММА ИСПОЛЬЗУЕТ РЕГИСТР R5 В КАЧЕСТВЕ
;УКАЗАТЕЛЯ ДАННЫХ И РЕГИСТРЫ R0, R1 И R2 ДЛЯ УДЕРЖАНИЯ
;ПРОМЕЖУТОЧНЫХ ДАННЫХ. ВЕКТОР ВЫЗОВА:
;       SCREEN: MOV     #DATA,R5
;               JSR     PC,SCREEN
        .GLOBL  SCREEN
SCREEN: MOV     (R5)+,R0        ;РАЗМЕР МАССИВА -> R0
        MOV     (R5)+,R1        ;ВЕРХНИЙ ПРЕДЕЛ -> R1
        MOV     (R5)+,R2        ;НИЖНИЙ ПРЕДЕЛ -> R2
LOOP:   CMP     (R5),R1         ;СЛИШКОМ ТЯЖЕЛЫЙ?
        BGT     REJECT          ;ДА, ОТБРОСИТЬ
        CMP     (R5),R2         ;СЛИШКОМ ЛЕГКИЙ?
        BLT     REJECT          ;ДА, ОТБРОСИТЬ
        TST     (R5)+           ;ПРОДВИНУТЬ УКАЗАТЕЛЬ ДАННЫХ
        BR      CHECK           ;ПРОВЕРИТЬ НА ЗАВЕРШЕНИЕ
REJECT: CLR     (R5)+           ;ОЧИСТИТЬ ДАННЫЕ,
                                ;ПРОДВИНУТЬ УКАЗАТЕЛЬ ДАННЫХ
CHECK:  SOB     R0,LOOP         ;ПРОВЕРКА НА ЗАВЕРШЕНИЕ
        RTS     PC
        .END
[ГЛАВНАЯ ПРОГРАММА]
        .GLOBL  SCREEN
DATA:   .WORD   N,220.,180.
        .WORD   D1,D2,...,DN
START:  MOV     #DATA,R5
        JSR     PC,SCREEN
        HALT
        .END    START
```

Числовой пример этой программы показан на рис. 5.10.

```
 1                      ;ПРИМЕР
 2                      ;
 3                      ;
 4                      ;
 5                      ;
 6                      ;(ПОДПРОГРАММА: SCREEN)
 7                              .GLOBL  SCREEN
 8 000000  012500  SCREEN: MOV   (R5)+,R0        ;РАЗМЕР МАССИВА -> R0
 9 000002  012501          MOV   (R5)+,R1        ;ВЕРХНИЙ ПРЕДЕЛ -> R1
10 000004  012502          MOV   (R5)+,R2        ;НИЖНИЙ ПРЕДЕЛ -> R2
11 000006  021501  LOOP:   CMP   (R5),R1         ;СЛИШКОМ ТЯЖЕЛЫЙ?
12 000010  003004          BGT   REJECT          ;ДА, ОТБРОСИТЬ
13 000012  021502          CMP   (R5),R2         ;СЛИШКОМ ЛЕГКИЙ?
14 000014  002402          BLT   REJECT          ;ДА, ОТБРОСИТЬ
15 000016  005725          TST   (R5)+           ;ПРОДВИНУТЬ УКАЗАТЕЛЬ ДАННЫХ
16 000020  000401          BR    CHECK           ;ПРОВЕРИТЬ НА ЗАВЕРШЕНИЕ
17 000022  005025  REJECT: CLR   (R5)+           ;ОЧИСТИТЬ ДАННЫЕ,
18                                              ;ПРОДВИНУТЬ УКАЗАТЕЛЬ ДАННЫХ
19 000024  077010  CHECK:  SOB   R0,LOOP         ;ПРОВЕРКА НА ЗАВЕРШЕНИЕ
20 000026  000207          RTS   PC
21         000001          .END
```

а)

```
 1                      ;ПРИМЕР
 2                      ; ЭТО ГЛАВНАЯ ПРОГРАММА ДЛЯ ПОДПРОГРАММЫ SCREEN
 3                      ;
 4                      ;
 5                      ;
 6                      ;(ГЛАВНАЯ ПРОГРАММА)
 7                              .GLOBL  SCREEN
 8 000000  000010  DATA:    .WORD   10,220.,180.
   000002  000334
   000004  000264
```

```
 9 000006  000310           .WORD   200.,150.,190.,230.,210.,170.,215.,240.
   000010  000226
   000012  000276
   000014  000346
   000016  000322
   000020  000252
   000022  000327
   000024  000360
10 000026  012705  START:   MOV     #DATA,R5        ;R5 - УКАЗАТЕЛЬ ДАННЫХ
           000000G
11 000032  004767           JSR     PC,SCREEN       ;ПЕРЕХОД К ПОДПРОГРАММЕ
           000000G
12 000036  000000           HALT
13         000026'          .END    START

   РЕЗУЛЬТАТ ВЫПОЛНЕНИЯ:
         DATA: 10, 220.,180., 200., 0, 190., 0, 210., 0, 215., 0
   ЗАМЕТИМ, ЧТО ЗНАЧЕНИЯ МЕНЬШЕ 180. ИЛИ БОЛЬШЕ 220. БЫЛИ СБРОШЕНЫ В 0
```

б)

Рис. 5.10. Числовой пример программы для составления списка футболистов

Пример 2. Нужно модифицировать пример 1 так, чтобы построить отдельный массив, показывающий результаты просмотра, а первоначальный массив данных оставить неизменным. Для иллюстрации воспользуемся другим способом передачи параметров.

```
<ПОДПРОГРАММА SCREEN>
;       ВЕКТОР ВЫЗОВА:
;               JSR     PC,SCREEN
;               .WORD   DATA    ;НАЧАЛЬНЫЙ АДРЕС МАССИВА ДАННЫХ
;               .WORD   N       ;РАЗМЕР МАССИВА ДАННЫХ
;               .WORD   ВЕРХНИЙ_ПРЕДЕЛ
;               .WORD   НИЖНИЙ_ПРЕДЕЛ
;               .WORD   RESULT  ;НАЧАЛЬНЫЙ АДРЕС МАССИВА РЕЗУЛЬТАТА
        .GLOBL  SCREEN
SCREEN: MOV     (R5)+,R0        ;R0 - УКАЗАТЕЛЬ ДАННЫХ
        MOV     (R5)+,R1        ;R1 - СЧЕТЧИК ДАННЫХ
        MOV     (R5)+,R2        ;ВЕРХНИЙ ПРЕДЕЛ --> R2
        MOV     (R5)+,R3        ;НИЖНИЙ ПРЕДЕЛ --> R3
        MOV     (R5)+,R4        ;R4 - УКАЗАТЕЛЬ РЕЗУЛЬТАТА
LOOP:   CMP     (R0),R2         ;СЛИШКОМ ТЯЖЕЛЫЙ?
        BGT     REJECT          ;ДА, ОТБРОСИТЬ
        CMP     (R0),R3         ;СЛИШКОМ ЛЕГКИЙ?
        BLT     REJECT          ;ДА, ОТБРОСИТЬ
        MOV     (R0)+,(R4)+     ;СКОПИРОВАТЬ ПЕРВОНАЧАЛЬНЫЕ ДАННЫЕ
                                ;В МАССИВ РЕЗУЛЬТАТА
        BR      CHECK           ;ПРОВЕРИТЬ НА ЗАВЕРШЕНИЕ
REJECT: CLR     (R4)+           ;ОБНУЛИТЬ ЭТОТ ЭЛЕМЕНТ РЕЗУЛЬТАТА
        TST     (R0)+           ;ПРОДВИНУТЬ УКАЗАТЕЛЬ ДАННЫХ
CHECK:  SOB     R1,LOOP         ;ПРОВЕРКА НА ЗАВЕРШЕНИЕ
        RTS     R5
        .END
<ГЛАВНАЯ ПРОГРАММА>
        .GLOBL  SCREEN
DATA:   .WORD   D1,D2,...,DN
RESULT: .BLKW   N
START:  JSR     R5,SCREEN
        .WORD   DATA,N,ВЕРХНИЙ_ПРЕДЕЛ,НИЖНИЙ_ПРЕДЕЛ
        .WORD   RESULT
        HALT
        .END    START
```

Числовой пример этой программы показан на рис. 5.11.

```
 1                      ;ПРИМЕР
 2                      ;
 3                      ;
 4                      ;
 5                      ;<ПОДПРОГРАММА SCREEN>
 6                              .GLOBL  SCREEN
 7 000000  012500  SCREEN: MOV   (R5)+,R0        ;R0 - УКАЗАТЕЛЬ ДАННЫХ
 8 000002  012501          MOV   (R5)+,R1        ;R1 - СЧЕТЧИК ДАННЫХ
 9 000004  012502          MOV   (R5)+,R2        ;ВЕРХНИЙ ПРЕДЕЛ -> R2
10 000006  012503          MOV   (R5)+,R3        ;НИЖНИЙ ПРЕДЕЛ -> R3
11 000010  012504          MOV   (R5)+,R4        ;R4 - УКАЗАТЕЛЬ РЕЗУЛЬТАТА
12 000012  021002  LOOP:   CMP   (R0),R2         ;СЛИШКОМ ТЯЖЕЛЫЙ?
13 000014  003004          BGT   REJECT          ;ДА, ОТБРОСИТЬ
14 000016  021003          CMP   (R0),R3         ;СЛИШКОМ ЛЕГКИЙ?
15 000020  002402          BLT   REJECT          ;ДА, ОТБРОСИТЬ
16 000022  012024          MOV   (R0)+,(R4)+     ;СКОПИРОВАТЬ ПЕРВОНАЧАЛЬНЫЕ ДАННЫЕ
17                                                ;В МАССИВ РЕЗУЛЬТАТА
18 000024  000402          BR    CHECK           ;ПРОВЕРИТЬ НА ЗАВЕРШЕНИЕ
19 000026  005024  REJECT: CLR   (R4)+           ;ОЧИСТИТЬ ЭТОТ ЭЛЕМЕНТ ДАННЫХ
20 000030  005720          TST   (R0)+           ;ПРОДВИНУТЬ УКАЗАТЕЛЬ ДАННЫХ
21 000032  077111  CHECK:  SOB   R1,LOOP        ;ПРОВЕРКА НА ЗАВЕРШЕНИЕ
22 000034  000205          RTS   R5
23         000001          .END
```

а)

```
 1                      ;
 2                      ;ПРИМЕР
 3                      ; ЭТО ГЛАВНАЯ ПРОГРАММА ДЛЯ ПОДПРОГРАММЫ SCREEN
 4                      ;
 5                      ;
 6                      ;
 7                      ;<ГЛАВНАЯ ПРОГРАММА>
 8                              .GLOBL  SCREEN
 9 000000  000310  DATA:   .WORD   200.,150.,190.,230.,210.,170.,215.,240.
   000002  000226
   000004  000276
   000006  000346
   000010  000322
   000012  000252
   000014  000327
   000016  000360
10 000020          RESULT: .BLKW   10
11 000040  004567  START:  JSR     R5,SCREEN
           000000G
12 000044  000000'         .WORD   DATA,10.,220.,180.,RESULT
   000046  000010
   000050  000334
   000052  000264
   000054  000020'
13 000056  000000         HALT
14         000040'         .END    START

РЕЗУЛЬТАТ ВЫПОЛНЕНИЯ: DATA: (В НЕИЗМЕНИВШИХСЯ ЗНАЧЕНИЯХ)
                       RESULT: 200.,0,190.,0,210.,0,215.,0
ЗАМЕТИМ, ЧТО МАССИВ DATA НЕ ИЗМЕНИЛСЯ, ТОГДА КАК В МАССИВЕ RESULT
ОТВЕРГНУТЫЕ ДАННЫЕ БЫЛИ СБРОШЕНЫ В НУЛЬ
```

б)

Рис. 5.11. Модифицированный числовой пример программы для составления списка футболистов

5.5. ПРИМЕРЫ

В этом разделе мы рассмотрим примеры, иллюстрирующие, как программу общего вида можно преобразовать в подпрограмму и как описанные в предыдущих разделах способы передачи параметров могут применяться к одной и той же программе, состоящей из главной программы и подпрограммы, при некоторой модификации. Для удобства определим способы передачи параметров следующим образом:

способ 1 Δ параметры помещаются после инструкции JSR
способ 2 Δ параметры помещаются в специально выделенной области
способ 3 Δ пересылка параметров через стековую память

Каждый способ программисту разрешается немного варьировать. Рисунки 5.12 – 5.16 представляют примеры передачи параметров в виде исходной программы с указанием деталей. Сначала мы покажем, как программа BIGNUM может быть преобразована в подпрограмму, которая содержит общие области данных N1 и N2 для временного хранения данных. Этот способ имеет некоторые недостатки. Сравните примеры, использующие способы 1 и 2, с примерами, использующими способ 3.

Во-первых, подпрограмма во втором случае использует большее пространство памяти. Во-вторых, подпрограмма во втором случае не может быть записана в только читаемую память. В-третьих, она не является **повторно входимой подпрограммой.** Так, в мультипрограммном применении возможны случаи, когда главная программа вызывает подпрограмму, но работа этой программы временно прекращается еще до завер-

```
;ПРОГРАММА, КОТОРАЯ СРАВНИВАЕТ ДВА ЧИСЛА
;И ЗАПОМИНАЕТ НАИБОЛЬШЕЕ ИЗ НИХ
;
;
.TITLE  BIGNUM
START:  CMP     N1,N2    ;СРАВНИТЬ ДВА ЧИСЛА
        BGE     POS
        MOV     N2,RESULT
        HALT
POS:    MOV     N1,RESULT
        HALT
N1:     .WORD   11
N2:     .WORD   7
RESULT: .BLKW   1
        .END    START
;
;
;ПРЕВРАЩЕНИЕ ПРОГРАММЫ В ОБЩУЮ ПОДПРОГРАММУ.
;ВЕКТОР ВЫЗОВА ПОДПРОГРАММЫ:
;       JSR     R5,SUB
;       .WORD   N1
;       .WORD   N2
;RESULT: .BLKW  1
;
        .GLOBL  SUB
SUB:    MOV     @(R5)+,N1
        MOV     @(R5)+,N2
        CMP     N1,N2
        BGE     POS
        MOV     N2,(R5)+        ;(N2) --> RESULT
        RTS     R5
POS:    MOV     N1,(R5)+        ;(N1) --> RESULT
        RTS     R5
N1:     .BLKW   1
N2:     .BLKW   1
        .END
```

Рис. 5.12. Пример написания подпрограммы

шения. В то же время другая главная программа может вызвать ту же самую подпрограмму, в результате чего данные в ячейках N1 и N2 будут потеряны. Поэтому общую область данных вставлять в подпрограмму не рекомендуется.

```
;ПРОГРАММА, КОТОРАЯ ПРЕВРАЩАЕТ ПРОГРАММУ BIGNUM В ПОДПРОГРАММУ
;ПРИ ИСПОЛЬЗОВАНИИ МЕТОДА ПЕРЕДАЧИ ПАРАМЕТРОВ НОМЕР 1
;
;
START:  NOP             ;ИМИТАЦИЯ ЛЮБОЙ ИНСТРУКЦИИ
        NOP             ;ИМИТАЦИЯ ЛЮБОЙ ИНСТРУКЦИИ
        JSR     R5,SUB
N1:     .WORD   11
N2:     .WORD   7
RESULT: .BLKW   1
        NOP             ;ИМИТАЦИЯ ЛЮБОЙ ИНСТРУКЦИИ
        HALT
;
SUB:    CMP     (R5)+,(R5)
        BGE     POS
        MOV     (R5)+,(R5)+     ;(N2) --> RESULT
        RTS     R5
POS:    TST     (R5)+           ;ТЕПЕРЬ (R5) УКАЗЫВАЕТ НА RESULT
        MOV     -4(R5),(R5)+    ;(N1) --> RESULT
        RTS     R5
        .END    START
;
;ПРОГРАММА, КОТОРАЯ ПРЕВРАЩАЕТ ПРОГРАММУ BIGNUM В ПОДПРОГРАММУ
;ПРИ ИСПОЛЬЗОВАНИИ МЕТОДА ПЕРЕДАЧИ ПАРАМЕТРОВ НОМЕР 1А
;
;
START:  NOP             ;ИМИТАЦИЯ ЛЮБОЙ ИНСТРУКЦИИ
        NOP             ;ИМИТАЦИЯ ЛЮБОЙ ИНСТРУКЦИИ
        JSR     R5,SUB
A:      .WORD   N1
B:      .WORD   N2
C:      .WORD   RESULT
        NOP             ;ИМИТАЦИЯ ЛЮБОЙ ИНСТРУКЦИИ
        HALT
N1:     .WORD   -2
N2:     .WORD   7
RESULT: .BLKW   1
        ;
        ;
SUB:    CMP     @(R5)+,@(R5)
        BGE     POS
        MOV     @(R5)+,@(R5)+
        RTS     R5
POS:    TST     (R5)+           ;ТЕПЕРЬ (R5) УКАЗЫВАЕТ НА RESULT
        MOV     @-4(R5),@(R5)+  ;(N1) --> RESULT
        .END    START
```

Рис. 5.13. Примеры передачи параметров подпрограмме

```
;ПРОГРАММА, КОТОРАЯ ПРЕВРАЩАЕТ ПРОГРАММУ BIGNUM В ПОДПРОГРАММУ
;ПРИ ИСПОЛЬЗОВАНИИ МЕТОДА ПЕРЕДАЧИ ПАРАМЕТРОВ НОМЕР 1Б
;
;
START:  NOP             ;ИМИТАЦИЯ ЛЮБОЙ ИНСТРУКЦИИ
        NOP             ;ИМИТАЦИЯ ЛЮБОЙ ИНСТРУКЦИИ
        JSR     R5,SUB
D1:     .WORD   11
D2:     .WORD   7
```

```
RESULT: .BLKW   1
        NOP             ;ИМИТАЦИЯ ЛЮБОЙ ИНСТРУКЦИИ
        HALT
        ;
SUB:    MOV     (R5)+,N1        ;ПОЛУЧИТЬ ДАННЫЕ
        MOV     (R5)+,N2
        CMP     N1,N2
        BGE     POS
        MOV     N2,(R5)+        ;(N2)  --> RESULT
        RTS     R5
POS:    MOV     N1,(R5)+        ;(N1)   ) RESULT
        RTS     R5
        .END    START
;
;ПРОГРАММА, КОТОРАЯ ПРЕВРАЩАЕТ ПРОГРАММУ BIGNUM В ПОДПРОГРАММУ
;ПРИ ИСПОЛЬЗОВАНИИ МЕТОДА ПЕРЕДАЧИ ПАРАМЕТРОВ НОМЕР 1В
;
;
START:  NOP             ;ИМИТАЦИЯ ЛЮБОЙ ИНСТРУКЦИИ
        NOP             ;ИМИТАЦИЯ ЛЮБОЙ ИНСТРУКЦИИ
        JSR     R5,SUB
A1:     .WORD   D1
A2:     .WORD   D2
RESULT: .BLKW   1
        NOP             ;ИМИТАЦИЯ ЛЮБОЙ ИНСТРУКЦИИ
        HALT
D1:     .WORD   -2
D2:     .WORD   7
        ;
SUB:    MOV     @(R5)+,N1       ;ПОЛУЧИТЬ ДАННЫЕ
        MOV     @(R5)+,N2
        CMP     N1,N2
        BGE     POS
        MOV     N2,(R5)+        ;(N2)  --> RESULT
        RTS     R5
POS:    MOV     N1,(R5)+        ;(N1)   -> RESULT
        RTS     R5
N1:     .BLKW   1
N2:     .BLKW   1
        .END    START
```

Рис. 5.14. Примеры передачи параметров подпрограмме

```
;ПРОГРАММА, КОТОРАЯ ПРЕВРАЩАЕТ ПРОГРАММУ BIGNUM В ПОДПРОГРАММУ
;ПРИ ИСПОЛЬЗОВАНИИ МЕТОДА ПЕРЕДАЧИ ПАРАМЕТРОВ НОМЕР 2
;
;
START:  NOP             ;ИМИТАЦИЯ ЛЮБОЙ ИНСТРУКЦИИ
        NOP             ;ИМИТАЦИЯ ЛЮБОЙ ИНСТРУКЦИИ
        MOV     #DATA,R5
        JSR     PC,SUB
        NOP             ;ИМИТАЦИЯ ЛЮБОЙ ИНСТРУКЦИИ
        HALT
DATA    .WORD   -11
        .WORD   -3
RESULT: .BLKW   1
SUB:    CMP     (R5)+,(R5)
        BGE     POS
        MOV     (R5)+,(R5)+     ;(DATA+2)  --> RESULT
        RTS     PC
POS:    TST     (R5)+           ;ТЕПЕРЬ (R5) УКАЗЫВАЕТ НА RESULT
        MOV     -4(R5),(R5)+    ;(DATA)  --> RESULT
```

```
        RTS     PC
        .END    START
        ;
;ПРОГРАММА, КОТОРАЯ ПРЕВРАЩАЕТ ПРОГРАММУ BIGNUM В ПОДПРОГРАММУ
;ПРИ ИСПОЛЬЗОВАНИИ МЕТОДА ПЕРЕДАЧИ ПАРАМЕТРОВ НОМЕР 2А
;
;
START:  NOP             ;ИМИТАЦИЯ ЛЮБОЙ ИНСТРУКЦИИ
        NOP             ;ИМИТАЦИЯ ЛЮБОЙ ИНСТРУКЦИИ
        MOV     #DATA,R5
        JSR     PC,SUB
        NOP             ;ИМИТАЦИЯ ЛЮБОЙ ИНСТРУКЦИИ
        HALT
DATA:   .WORD   -11
        .WORD   -3
RESULT: .BLKW   1
SUB:    MOV     (R5)+,N1        ;(DATA) --> N1
        MOV     (R5)+,N2        ;(DATA) --> N2
        CMP     N1,N2
        BGE     POS
        MOV     N2,(R5)+        ;(DATA+2) --> RESULT
        RTS     PC
POS:    MOV     N1,(R5)+        ;(DATA) --> RESULT
        RTS     PC
N1:     .BLKW   1
N2:     .BLKW   1
        .END    START
```

Рис. 5.15. Примеры передачи параметров подпрограмме

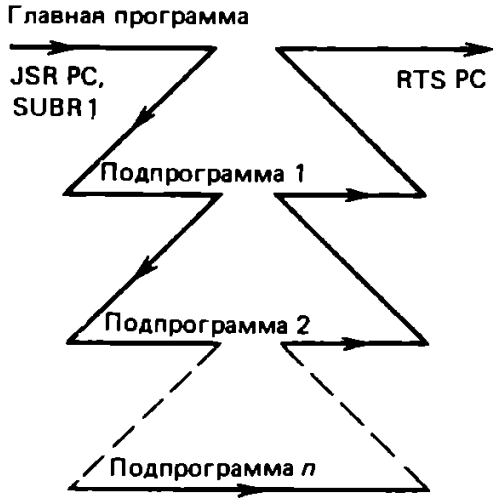
```
;ПРОГРАММА, КОТОРАЯ ПРЕВРАЩАЕТ ПРОГРАММУ BIGNUM В ПОДПРОГРАММУ
;ПРИ ИСПОЛЬЗОВАНИИ МЕТОДА ПЕРЕДАЧИ ПАРАМЕТРОВ НОМЕР 3
;
;
START:  NOP
        NOP
PREP:   MOV     N2,-(SP)        ;(N2) --> СТЕК
        MOV     N1,-(SP)        ;(N1) --> СТЕК
JMPSUB: JSR     PC,SUB
        MOV (SP)+,RESULT
        NOP
        HALT
N1:     .WORD   13
N2:     .WORD   -13
RESULT: .BLKW   1
        ;
SUB:    MOV     (SP)+,R0        ;СОХРАНИТЬ (PC) ВОЗВРАТА
        CMP     (SP)+,(SP)      ;СРАВНИТЬ (N1) И (N2)
        BGE     POS
        MOV     R0,-(SP)        ;ВОССТАНОВИТЬ (PC) ВОЗВРАТА
        RTS     PC
POS:    TST     -(SP)           ;УМЕНЬШИТЬ (SP) НА 2
        MOV     R0,-(SP)        ;ВОССТАНОВИТЬ (PC) ВОЗВРАТА
        RTS     PC
        .END    START
;
;ПРОГРАММА, КОТОРАЯ ПРЕВРАЩАЕТ ПРОГРАММУ BIGNUM В ПОДПРОГРАММУ
;ПРИ ИСПОЛЬЗОВАНИИ МЕТОДА ПЕРЕДАЧИ ПАРАМЕТРОВ НОМЕР 3А
;
;
START:  NOP
        NOP
```

```
PREP:   MOV     D2,-(SP)        ;(D2) -> СТЕК
        MOV     D1,-(SP)        ;(D1) -> СТЕК
JMPSUB: JSR     PC,SUB
        MOV     (SP)+,RESULT
        NOP
        HALT
D1:     .WORD   13
D2:     .WORD   -13
RESULT: .BLKW   1
        ;
SUB:    MOV     (SP)+,R0        ;СОХРАНИТЬ (PC) ВОЗВРАТА
        MOV     (SP)+,N1        ;(D1) -> N1
        MOV     (SP)+,N2        ;(D2) -> N2
        CMP     N1,N2           ;СРАВНИТЬ (N1) И (N2)
        BGE     POS
        MOV     N2,-(SP)
        MOV     R0,-(SP)        ;ВОССТАНОВИТЬ (PC) ВОЗВРАТА
        RTS     PC
POS:    MOV     N1,-(SP)
        MOV     R0,-(SP)        ;ВОССТАНОВИТЬ (PC) ВОЗВРАТА
        RTS     PC
N1:     .BLKW   1
N2:     .BLKW   1
        .END    START
```

Рис. 5.16. Примеры передачи параметров подпрограмме

5.6. ВЛОЖЕННЫЕ ПОДПРОГРАММЫ

Возможность использования в системе PDP-11 стековой памяти делает операцию вложения подпрограмм друг в друга очень простой. Одна подпрограмма может вызывать другую, та, в свою очередь, следующую и т. д. до тех пор, пока будет хватать стековой памяти для сохранения всех адресов возврата. Рис. 5.17 иллюстрирует вложение подпрограмм и показывает карту стековой памяти в то время, когда система выполняет *n*-ю подпрограмму. Предполагается, что стековая память была инициализирована с адреса 1172. Первым в стек был протолкнут адрес возврата к главной программе,



Карта стековой памяти

Адрес	Содержимое
	x x x x x x
	x x x x x x
•	
•	
•	
•	
•	
	Возврат к подпрограмме ($n-1$)
•	
•	
•	
•	
001164	Возврат к подпрограмме 2
001166	Возврат к подпрограмме 1
001170	Возврат к главной программе
001172	x x x x x x

Рис. 5.17. Выполнение вложенных подпрограмм

за ним последовали адреса возврата к подпрограммам 1, 2, . . . , n–1. По мере завершения работы подпрограмм сохраненные в стеке адреса возврата будут один за другим выталкиваться назад в регистр PC в правильном порядке.

5.7. СОПРОГРАММЫ

Ранее мы ввели специальную инструкцию для связывания сопрограмм. В случае сопрограмм инструкция вызова совпадает с инструкцией возврата, а главной программы как таковой нет: две подпрограммы могут вызывать друг друга. Таким образом каждая из них выступает как подпрограмма другой, и такая ситуация известна как **сопрограммная** конфигурация. Типичное приложение сопрограмм – процесс ассемблирования, в котором одна подпрограмма может обрабатывать символьный адрес, а другая – мнемонический код операции.

5.8. РЕКУРСИЯ

Рекурсия – это процесс, при котором подпрограмма вызывает саму себя. Лучше всего понять это можно с помощью примера.

Пример. Вычисление факториала числа $n!$ при использовании стековой памяти в качестве среды для передачи параметра. Определим сначала

$n \geqslant 1$
$n! = n(n-1)! = n(n-1) \ldots 1$
$0! = 1! = 1$

Программа вычисления $n!$

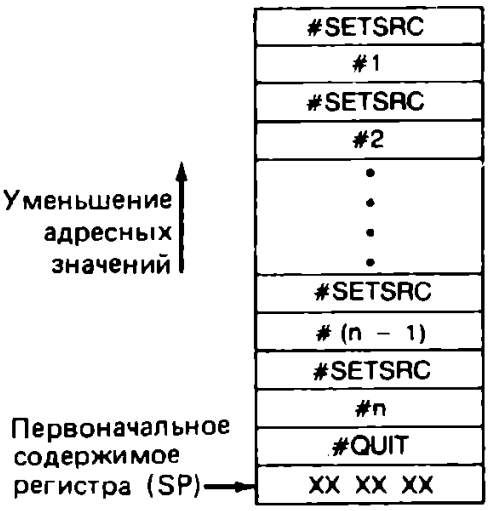
```
;СЕКЦИЯ 1: ИНИЦИАЛИЗАЦИЯ ДЛЯ ОПЕРАЦИИ ФАКТОРИАЛА
1) START: MOV   #N,R0          ;R0 БУДЕТ СЧЕТЧИКОМ ДАННЫХ
2)        CLR   R2             ;РЕГИСТРОВАЯ ПАРА (R2,R3)
                               ;БУДЕТ МЕСТОМ НАЗНАЧЕНИЯ
3)        MOV   #1,R3          ;ДЛЯ ПРОИЗВЕДЕНИЯ, (R3) - МЛАДШЕЕ СЛОВО
4)        JSR   PC,FACTOR      ;ПЕРЕХОД К ПОДПРОГРАММЕ FACTOR
5) QUIT:  HALT
;СЕКЦИЯ 2: ГЕНЕРАЦИЯ МАССИВА ДАННЫХ В СТЕКОВОЙ ПАМЯТИ
6) FACTOR: TST  R0             ;(R0)=0 ИЛИ ЗАВЕРШЕНО ЛИ
                               ;СОЗДАНИЕ МАССИВА В СТЕКЕ?
7)        BEQ   BEGIN          ;ДА, НАЧАТЬ УМНОЖЕНИЕ
8)        MOV   R0,-(SP)       ;НЕТ, ПРОДОЛЖАТЬ СОЗДАНИЕ МАССИВА
9)        DEC   R0             ;УМЕНЬШИТЬ СЧЕТЧИК ДАННЫХ
10)       JSR   PC,FACTOR      ;ПОДПРОГРАММА ВЫЗЫВАЕТ СЕБЯ
;СЕКЦИЯ 3: ПРОЦЕСС УМНОЖЕНИЯ
11) SETSRC: MOV (SP)+,R1       ;УСТАНОВИТЬ ИСХОДНЫЙ ОПЕРАНД
                               ;ДЛЯ УМНОЖЕНИЯ
12)       JSR   PC,MULT        ;ВЫЗВАТЬ ПОДПРОГРАММУ УМНОЖЕНИЯ,
                               ;ПРОТОЛКНУТЬ В СТЕК #BEGIN
13) BEGIN: RTS  PC             ;ПЕРЕЙТИ К SETSRC, ЕСЛИ НЕ КОНЕЦ,
                               ;ИНАЧЕ #QUIT -> PC
14) MULT: MUL   R1,R2          ;(R1) * (R2,R3) -> (R2,R3)
15)       RTS   PC             ;#BEGIN -> PC
16)       .END  START
```

Обратите внимание, что программа состоит из трех секций. Секция 1 служит для инициализации, секция 2 – для установки в стеке массива данных, секция 3 – для умножения.

В секции 1 мы инициализируем регистр R0 значением $\#n$, которое будет использоваться в качестве счетчика данных при установке массива данных в стековой памяти, а регистровую пару (R2, R3) – значениями 0 и 1, причем эта пара будет служить для хранения получаемого произведения. Таким образом, после инструкции 3 мы имеем:

(R0) = $\#n$
(R2), (R3) = 0,1

Инструкция 4 передает управление инструкции 6. Инструкции 6 – 9 устанавливают в стеке массив данных в нужном порядке:



Давайте проследим по этой карте несколько шагов. Замечаем, что инструкция 4 проталкивает в стек скорректированное содержимое регистра PC, представляющее собой адресное значение #QUIT, и программа переходит к инструкции 6. Поскольку первоначально (R0) = #*n*, что не равно нулю, программа переходит к инструкции 8 и таким образом выполняется пересылка #*n* → СТЕК. Затем инструкция 10 проталкивает в стек значение #SETSRC, а программа вновь переходит к инструкции 6. Но теперь (R0) = #(*n*–1), и в результате выполнения инструкции 8 в стек проталкивается #(*n*––1). Этот процесс повторяется, пока содержимое регистра R0 не станет равным нулю.

Таким образом получена окончательная карта стека; в ней мы имеем числа, смешанные с адресным значением #SETSRC. В этом месте программа с инструкции 7 переходит на инструкцию 13, которая переносит в регистр PC верхний элемент стека, #SETSRC. Центральный процессор начинает выполнять инструкцию 11. Она выталкивает из стека верхний элемент, который в это время равен #1, и помещает его в регистр R1. Эта инструкция устанавливает исходный операнд для подпрограммы умножения, т. е. инструкции 14, которая умножает (R1) на (R2, R3) и помещает произведение в пару регистров R2 и R3, причем младшее слово оказывается в регистре R3. Поскольку инструкция 12 поместила в стек адресное значение #BEGIN, инструкция 15 приведет программу к инструкции 13, которая, в свою очередь, поместит в регистр PC значение #SETSRC. Этот процесс повторяется до тех пор, пока не будет достигнут самый нижний элемент стека, #QUIT, и пока он не будет перенесен в регистр PC. После этого программа завершается на инструкции 5, оставляя окончательный результат в паре регистров R2, R3.

На рис. 5.18 представлена числовая программа для этого примера. Для простоты мы полагаем *n* = 5.

```
 1                      ;ПРИМЕР
 2                      ;
 3                      ;
 4                      ;
 5                      ;
 6 000000  012700  START: MOV     #5,R0          ;R0 БУДЕТ СЧЕТЧИКОМ ДАННЫХ
           000005
 7 000004  012702         MOV     #1,R2          ;МЕСТО ДЛЯ ПРОИЗВЕДЕНИЯ
           000001
 8 000010  004767         JSR     PC,FACTOR
           000002
 9 000014  000000  QUIT:  HALT
10 000016  005700  FACTOR: TST     R0             ;(R0)=0? (МАССИВ СОЗДАН?)
12 000020  001407         BEQ     BEGIN          ;ДА, НАЧАТЬ УМНОЖЕНИЕ
```

```
13 000022  010046           MOV     R0,-(SP)        ;НЕТ, ПРОДОЛЖАТЬ СОЗДАНИЕ МАССИВА
14 000024  005300           DEC     R0              ;УМЕНЬШИТЬ СЧЕТЧИК ДАННЫХ
15 000026  004767           JSR     PC,FACTOR       ;ПОДПРОГРАММА ВЫЗЫВАЕТ СЕБЯ
           177764
16 000032  012601  SETSRC:  MOV     (SP)+,R1        ;УСТАНОВИТЬ ИСХОДНЫЙ ОПЕРАНД
17                                                  ;ДЛЯ УМНОЖЕНИЯ
18 000034  004767           JSR     PC,MULT         ;ВЫЗВАТЬ ПОДПРОГРАММУ УМНОЖЕНИЯ
           000002
19 000040  000207  BEGIN:   RTS     PC              ;ПЕРЕЙТИ К SETSRC, ЕСЛИ НЕ КОНЕЦ,
20                                                  ;ИНАЧЕ #QUIT -> PC
21 000042  070402  MULT:    MOV     R2,R4
22 000044  005002           CLR     R2
23 000046  060402  LOOP:    ADD     R4,R2
24 000050  077102           SOB     R1,LOOP
25 000052  000207           RTS     PC
26         000000'          .END    START

   РЕЗУЛЬТАТ ВЫПОЛНЕНИЯ:    R2 СОДЕРЖИТ 170 (ВОСЬМЕРИЧНОЕ ЗНАЧЕНИЕ)
```

Рис. 5.18. Числовой пример рекурсивной процедуры для вычисления 5!

5.9. ПРОГРАММА "ПУЗЫРЬКОВОЙ" СОРТИРОВКИ

Программой сортировки является такая, которая берет упорядоченный в случайном порядке массив данных и выстраивает его в числовом порядке. Как следует из названия, программа "пузырьковой" сортировки сортирует массив данных так, что элементы с меньшими значениями будут "всплывать" к меньшим адресам, а наименьший элемент данных окажется расположенным по наименьшему адресу или на вершине массива. Эта программа может быть полезной также для сортировки массива в алфавитном порядке, поскольку значения букв в коде ASCII расположены в числовом порядке, когда буква А имеет наименьшее числовое значение.

На рис. 5.19, *а* показана карта памяти для массива данных из n элементов. Определим параметры следующим образом:

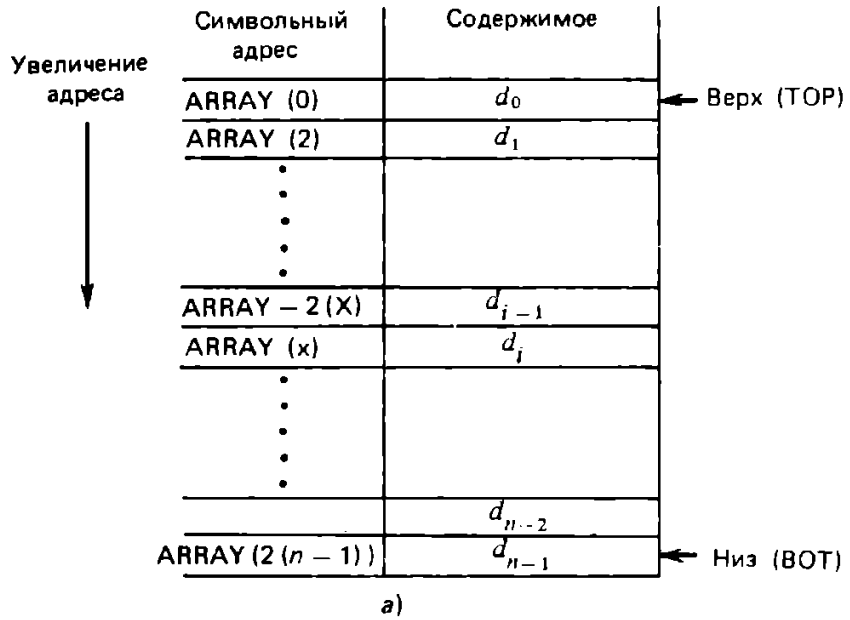
ARRAY $\triangleq$ начальный адрес массива

ARRAY + 2(n–1) $\triangleq$ максимальный адрес массива

TOP $\triangleq$ указатель на верхний элемент неотсортированной части массива данных

BOT $\triangleq$ указатель на самый нижний элемент массива данных

IDX $\triangleq$ адрес первоначального значения индекса



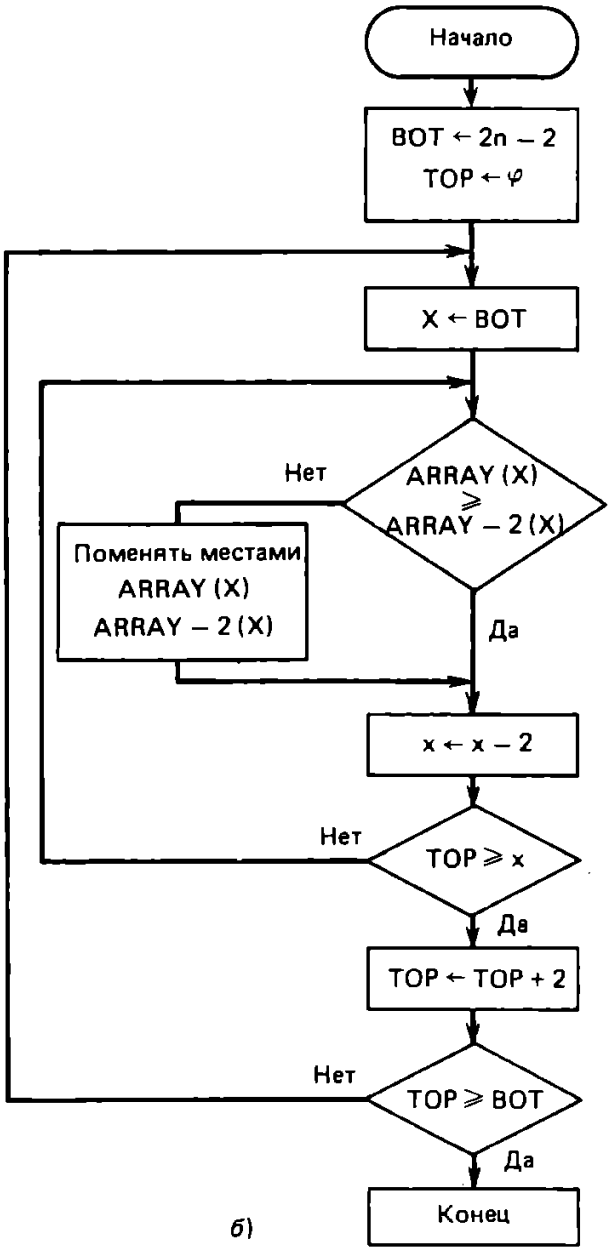
а)

Рис. 5.19, *а*. Карта памяти

Обратите внимание, что в адресной колонке мы использовали индексированный адресный символ, т. е. ARRAY(2(n–1)) имеет адресное значение #ARRAY + 2(n–1). Напоминаем, что каждый элемент данных занимает одно слово или два байта памяти; именно поэтому мы умножаем n–1 на 2.

Теперь мы сравним два нижних элемента, а именно d_{n-1} и d_{n-2}, с адресами ARRAY (2(n-1)) и ARRAY(2(n–2)) соответственно. Если $d_{n-1} < d_{n-2}$, то эти два элемента меняются местами; в противном случае они остаются неизменными. Продолжаем сравнение d_{n-1} и d_{n-2} с d_{n-2} и d_{n-3} и т. д. В конце концов элемент с наименьшим значением "всплывает" к самому верху и остается там навсегда. Поскольку после "всплытия" наименьшего элемента на самый верх сравнивать его больше не нужно, мы можем продвинуть указатель TOP на одно слово по направлению к "дну" массива, чтобы избежать повторяющегося сравнения наименьшего элемента с другими. Процесс завершится при TOP = BOT.

На рис. 5.19, *б* показана блок-схема этого алгоритма, а листинг программы для n = = 10_8 представлен на рис. 5.19, *в*. Эта программа понятна, и ее работу легко можно проследить. Однако она не имеет формата подпрограммы. Давайте в качестве упражнения модифицируем ее, чтобы сделать подпрограммой. На рис. 5.19, *г* показаны главная программа и подпрограмма. На рис. 5.19, *д* дана другая альтернатива.



б)

Рис. 5.19, *б*. Алгоритм "пузырьковой"сортировки

```
 1                                      ;ПРИМЕР ПРОГРАММЫ СОРТИРОВКИ
 2                                      ;
 3                                      ;
 4                                              .MCALL  .WRITE,.EXIT
 5 000000  000000                       TOP:    .WORD   0
 6 000002  000016                       BOT:    .WORD   16
 7 000004  000016                       IDX:    .WORD   16
 8 000006  000006  177773  177777      ARRAY:  .WORD   6,-5,-1,0,3,7,-6,1
   000014  000000  000003  000007
   000022  177772  000001
 9 000026  016701  177746              START:  MOV     TOP,R1          ;R1 - УКАЗАТЕЛЬ ВЕРХА ДАННЫХ
10 000032  016702  177744                      MOV     BOT,R2          ;R2 - УКАЗАТЕЛЬ НИЗА ДАННЫХ
11 000036  016700  177742              LOOP1:  MOV     IDX,R0          ;(R0) = ИНДЕКС
12 000042  026060  000006' 000004'    LOOP2:  CMP     ARRAY(R0),ARRAY-2(R0)   ;СРАВНИТЬ СОСЕДНИЕ ЭЛЕМЕНТЫ
13 000050  002005                              BGE     INDEX
14 000052  016003  000006'             SWAP:   MOV     ARRAY(R0),R3            ;МЕНЬШЕЕ ЧИСЛО --> R3
15 000056  016060  000004' 000006'             MOV     ARRAY-2(R0),ARRAY(R0)   ;БОЛЬШЕЕ ЧИСЛО ОПУСТИТЬ НИЖЕ
16 000064  005740                      INDEX:  TST     -(R0)           ;R0 УКАЗЫВАЕТ НА МЕНЬШИЙ АДРЕС
17 000066  020067  177706                      CMP     R0,TOP          ;"ПУЗЫРЕК" ДОСТИГ ВЕРХА?
18 000072  005721                              TST     (R1)+           ;(TOP)+2 --> TOP
19 000074  020102                              CMP     R1,R2           ;СДЕЛАНО?
20 000076  002757                              BLT     LOOP1           ;СЛЕДУЮЩИЙ КРУГ
21 000100  005000                              CLR     R0
22 000102                              OUTPUT: .WRITE  ARRAY(R0)       ;ВЫВЕСТИ МАССИВ
23 000106  005720                              TST     (R0)+
24 000110  020067  177666                      CMP     R0,BOT          ;ВЫВОД ЗАКОНЧЕН?
25 000114  003772                              BLE     OUTPUT
26 000116                                      .EXIT
27         000026'                             .END    START

   РЕЗУЛЬТАТ ВЫПОЛНЕНИЯ:   -6 -5 -1 0 1 3 6 7
```

Рис. 5.19, *в*. Программа пузырьковой сортировки для массива из 10_8 элементов

```
 1                                          ;ПРИМЕР ПРОГРАММЫ СОРТИРОВКИ
 2                                          ;
 3                                          ;ВЕКТОР ВЫЗОВА ПОДПРОГРАММЫ:
 4                                          ;       JSR     R5,SORT
 5                                          ;       .WORD   N       ;РАЗМЕР МАССИВА
 6                                                  .GLOBL  SORT,ARRAY
 7                                                  .MCALL  .WRITE,.EXIT
 8 000000  005001                           SORT:   CLR     R1              ;R1 - УКАЗАТЕЛЬ ВЕРХА ДАННЫХ
 9 000002  012502                                   MOV     (R5)+,R2        ;РАЗМЕР МАССИВА -> R2
10 000004  006302                                   ASL     R2              ;2*(R2) -> R2
11 000006  005742                                   TST     -(R2)           ;(R2)-2 -> R2 КАК УКАЗАТЕЛЬ НИЗА ДАННЫХ
12 000010  010200                           LOOP1:  MOV     R2,R0           ;(R0) = ИНДЕКСНЫЙ РЕГИСТР
13 000012  026060  000056' 000054'          LOOP2:  CMP     ARRAY(R0),ARRAY-2(R0)   ;СРАВНИТЬ СОСЕДНИЕ ЭЛЕМЕНТЫ
14 000020  002007                                   BGE     INDEX
15 000022  016003  000056'                  SWAP:   MOV     ARRAY(R0),R3            ;МЕНЬШЕЕ ЧИСЛО -> R3
16 000026  016060  000054' 000056'                 MOV     ARRAY-2(R0),ARRAY(R0)   ;БОЛЬШЕЕ ЧИСЛО ОПУСТИТЬ НИЖЕ
17 000034  010360  000054'                          MOV     R3,ARRAY-2(R0)          ;МЕНЬШЕЕ ЧИСЛО ПОДНЯТЬ ВЫШЕ
18 000040  005740                           INDEX:  TST     -(R0)           ;R0 УКАЗЫВАЕТ НА МЕНЬШИЙ АДРЕС
19 000042  020001                                   CMP     R0,R1           ;"ПУЗЫРЕК" ДОСТИГ ВЕРХА?
20 000044  003362                                   BGT     LOOP2           ;СРАВНИТЬ СЛЕДУЮЩИЕ СОСЕДНИЕ ЭЛЕМЕНТЫ
21 000046  005721                                   TST     (R1)+           ;(TOP)+2 -> TOP
22 000050  020102                                   CMP     R1,R2           ;СДЕЛАНО?
23 000052  002756                                   BLT     LOOP1           ;СЛЕДУЮЩИЙ КРУГ
24 000054  000205                                   RTS     R5
25                                          ;
26                                          ;ГЛАВНАЯ ПРОГРАММА
27                                                  .GLOBL  SORT,ARRAY
28 000056  000006  177773  177777          ARRAY:  .WORD   6,-5,-1,0,3,7,-6,1
   000064  000000  000003  000007
   000072  177772  000001
29 000076  004567  177676                  START:  JSR     R5,SORT
30 000102  000010                                  .WORD   10
31 000104  005000                                  CLR     R0
32 000106                                  OUTPUT: .WRITE  ARRAY(R0)       ;ВЫВЕСТИ МАССИВ
33 000112  005720                                  TST     (R0)+
34 000114  020027  000016                          CMP     R0,#16          ;ВЫВОД ЗАКОНЧЕН?
35 000120  003772                                  BLE     OUTPUT
36 000122                                          .EXIT
37         000076'                                 .END    START

   РЕЗУЛЬТАТ ВЫПОЛНЕНИЯ:    -6 -5 -1 0 1 3 6 7
```

Рис. 5.19, *г*. Главная программа, вызывающая подпрограмму пузырьковой сортировки

```
                                ;ПРИМЕР ПРОГРАММЫ СОРТИРОВКИ
                                ;АЛЬТЕРНАТИВНЫЙ ВАРИАНТ ГЛАВНОЙ ПРОГРАММЫ И ПОДПРОГРАММЫ
                                ;
                                ;ВЕКТОР ВЫЗОВА ПОДПРОГРАММЫ:
                                ;       JSR     R5,SORT
000000  000050'                 ;       .WORD   ARRAY
                                ;       .WORD   N       ;РАЗМЕР МАССИВА
                                        .GLOBL  SORT
                                        .MCALL  .WRITE,.EXIT
000002  012501                  SORT:   MOV     (R5)+,R1        ;R1 - УКАЗАТЕЛЬ ВЕРХА ДАННЫХ
000004  012502                          MOV     (R5)+,R2        ;РАЗМЕР МАССИВА -> R2
000006  006302                          ASL     R2              ;2*(R2) -> R2
000010  005742                          TST     -(R2)           ;(R2)-2 -> R2 КАК УКАЗАТЕЛЬ НИЗА ДАННЫХ
000012  060102                          ADD     R1,R2           ;(R2) = #ARRAY+2(N-1) КАК УКАЗАТЕЛЬ НИЗА ДАННЫХ
000014  010200                  LOOP1:  MOV     R2,R0           ;(R0) = ИНДЕКСНЫЙ РЕГИСТР
000016  021040                  LOOP2:  CMP     (R0),-(R0)      ;СРАВНИТЬ СОСЕДНИЕ ЭЛЕМЕНТЫ
000020  002004                          BGE     INDEX           ;R0 УКАЗЫВАЕТ НА НИЖНИЙ АДРЕС
000022  012003                  SWAP:   MOV     (R0)+,R3        ;МЕНЬШЕЕ ЧИСЛО -> R3
000024  011040                          MOV     (R0),-(R0)      ;БОЛЬШЕЕ ЧИСЛО ОПУСТИТЬ НИЖЕ
000026  010360  000002                  MOV     R3,2(R0)        ;МЕНЬШЕЕ ЧИСЛО ПОДНЯТЬ ВЫШЕ
000032  020001                  INDEX:  CMP     R0,R1           ;СРАВНИТЬ УКАЗАТЕЛЬ ВЕРХА С ИНДЕКСОМ
000036  003367                          BGT     LOOP2           ;СРАВНИТЬ СЛЕДУЮЩИЕ СОСЕДНИЕ ЭЛЕМЕНТЫ
000040  005721                          TST     (R1)+           ;(TOP)+2 -> TOP
000042  020102                          CMP     R1,R2           ;СДЕЛАНО?
000044  002763                          BLT     LOOP1           ;СЛЕДУЮЩИЙ КРУГ
000046  000205                          RTS     R5
                                ;
                                ;ГЛАВНАЯ ПРОГРАММА
                                        .GLOBL  SORT
000050  000006  177773  177777  ARRAY:  .WORD   6,-5,-1,0,3,7,-6,1
000056  000000  000003  000007
000064  177772  000001
000070  004567  177706          START:  JSR     R5,SORT
000074  000050'                         .WORD   ARRAY
000076  000010                          .WORD   10
000100  005000                          CLR     R0
000102                          OUTPUT: .WRITE  ARRAY(R0)       ;ВЫВЕСТИ МАССИВ
000106  005720                          TST     (R0)+
000110  020027  000016                  CMP     R0,#16          ;ВЫВОД ЗАКОНЧЕН?
000114  003772                          BLE     OUTPUT
000116                                  .EXIT
        000070'                         .END    START

РЕЗУЛЬТАТ ВЫПОЛНЕНИЯ:   -6 -5 -1 0 1 3 6 7
```

Рис. 5.19, *д*. Альтернативный вариант главной программы и подпрограммы для пузырьковой сортировки

5.10. УПРАЖНЕНИЯ

1. Кратко опишите функцию директивы .GLOBL.

2. Кратко объясните, почему желательно иметь некий общий формат для любой подпрограммы.

3. Модифицируйте программу из разд. 5.4 так, чтобы отвергнутые кандидаты идентифицировались по своему порядковому номеру в первоначальном массиве. То есть первый кандидат идентифицируется как 1 и т. д. Они должны быть перечислены в блоке памяти, начинающемся с адресной метки NONQUAL, означающей "неквалифицированные".

4. Напишите подпрограмму в общем формате (используя три разных способа передачи параметров), которая будет вычислять среднее значение для массива данных из 2N элементов, где N – положительное целое число.

5. Напишите подпрограмму в общем формате, которая будет добавлять соответствующий бит четности к старшему биту массива 7-битовых символов в коде ASCII. Она должна быть гибкой настолько, чтобы пользователь мог путем передачи соответствующего параметра задавать, как должна производиться оценка (по четности или нечетности).

6. Напишите подпрограмму в общем формате, которая будет умножать два числа без знака.

7. Напишите подпрограмму в общем формате, которая сможет вызывать подпрограмму упр. 6 для двух целых чисел со знаком, представленных в виде дополнения до двух.

8. Напишите подпрограмму в общем формате, которая будет делить два целых числа без знака.

9. Напишите подпрограмму в общем формате, которая сможет вызывать подпрограмму упр. 8 для двух целых чисел со знаком, представленных в виде дополнения до двух.

10. Напишите подпрограмму в общем формате, которая сможет вычислять статистические параметры: среднее значение (математическое ожидание) и среднеквадратическое отклонение для массива из N целых чисел со знаком.

ГЛАВА 6

МАКРОИНСТРУКЦИИ ИЛИ МАКРОСЫ

6.1. ВВЕДЕНИЕ

Макроинструкция или макрос – это средство, обеспечивающее пользователю или системному программисту гибкость и удобство. Практически оно дает возможность пользователю создавать свои собственные "составные инструкции" из основных инструкций, доступных в системе PDP-11. Макросы иногда путают с подпрограммами. Хотя и те, и другие можно применять повторно (без необходимости переписывать каждый раз составляющие их инструкции), все же есть значительные различия в структуре, свойствах, объеме памяти и времени выполнения.

Подпрограммы, описанные в предыдущей главе, обычно называют **закрытыми подпрограммами**, тогда как макросы называют **открытыми подпрограммами**. Вспомним, что для вызова подпрограммы главная программа должна использовать пару связующих инструкций, JSR и RTS, для перехода к подпрограмме и для возврата из нее. Помимо этого главная программа ответственна за пересылку правильного набора параметров к подпрограмме и от нее. Все эти процессы требуют дополнительного объема памяти и добавочного времени выполнения, поэтому о них говорят как о "накладных расходах", которые приходится платить за использование подпрограмм. Однако сама подпрограмма занимает память только в одном месте, независимо от того, сколько раз она вызывается.

В противоположность подпрограмме макроинструкция – это просто группа основных инструкций ЭВМ PDP-11, собранных в одну "суперинструкцию". Поскольку макрос принадлежит к категории инструкций, он имеет поля для кода операции и операндов. При каждом использовании макроса ассемблер "транслирует" соответствующие ему основные инструкции в машинный код с целью построения объектного файла. Следовательно, если какой-то конкретный макрос, машинный код которого занимает

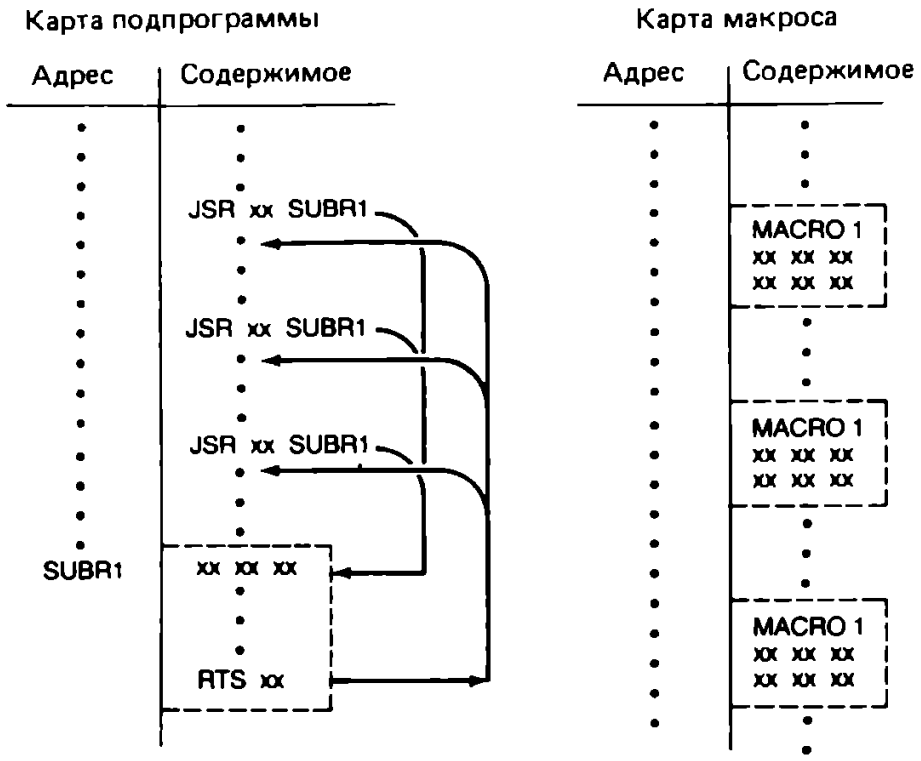


Рис. 6.1. Карты памяти для подпрограммы и макроса

10 слов памяти, используется в программе 10 раз, то он будет "расширен" или "оттранслирован" 10 раз и для него потребуется, таким образом, 100 слов памяти.

На рис. 6.1 показано различие карт памяти подпрограммы и макроса. Очевидно, что объем памяти, необходимый для макроса, пропорционален частоте его использования, тогда как для подпрограммы требуемый объем памяти от частоты вызова практически не зависит. Однако в последнем случае нужно большее время для выполнения. В качестве общего правила можно руководствоваться следующим соображением: метод подпрограмм больше подходит для длинных подпрограмм с высокой частотой вызова, а метод макросов больше пригоден для коротких групп инструкций. Определение того, какой метод лучше для конкретного приложения, сводится к поиску компромисса между временем выполнения и объемом памяти.

6.2. КЛАССИФИКАЦИЯ МАКРОИНСТРУКЦИЙ

Есть два существенных типа макросов: макросы, определенные системой, и макросы, определенные пользователем. Системно определенный макрос – это макроинструкция, определенная разработчиком системы и поставляемая ее производителем. Или это макроинструкция, определенная системным программистом, который установил ее в системе для прикладных программистов. Например, для базовой системы PDP-11 мы можем располагать следующими системно определенными макросами:

1. .PNUM N – распечатывает на выходном терминале восьмеричное значение со знаком из ячейки N.
2. .RNUM N – считывает восьмеричное значение со знаком с клавиатуры.
3. .EXIT – используется для замены инструкции HALT (чтобы после завершения программы производился выход в операционную систему, а не останов процессора).

Сколько и каких системно определенных макросов доступно, пользователь может узнать из соответствующих системных руководств (чтобы не изобретать их повторно в своих прикладных программах). Однако во многих случаях системные программисты могут создавать системно определенные макросы для собственных прикладных

программ. Для этого системный программист сначала определяет макросы, затем устанавливает их в системе для всех пользователей. Например, в нашей лаборатории мы создали и установили удобные для нас системно определенные макросы .WRITE, .RAD и .READ. Для единообразия и во избежание путаницы созданные и установленные в системе системно определенные макросы всегда начинаются с точки.

Процедура использования системно определенных макросов:

1. Если мы намереваемся использовать в своей программе какие-то системно определенные макроинструкции, то записываем их в первой строке программы: .MCALL .PNUM, .EXIT, .WRITE. Здесь .MCALL – это псевдоинструкция, информирующая ассемблер, что мы будем использовать эти системные макросы.
2. Затем мы пользуемся макросами точно так же, как и базовыми инструкциями.

Например:

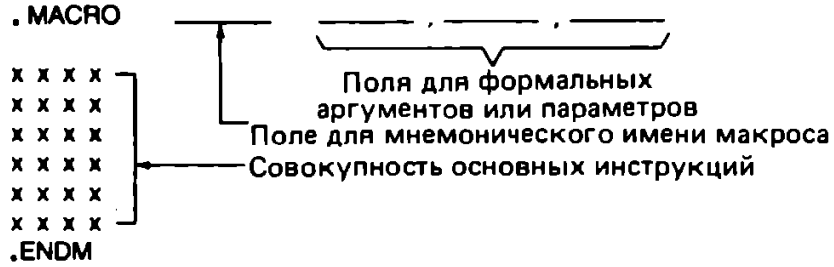
```
        .MCALL  .PNUM,.EXIT
START:  .PNUM   N       ;ИСПОЛЬЗОВАНИЕ СИСТЕМНОГО МАКРОСА
                        ;В КАЧЕСТВЕ ИНСТРУКЦИИ
        .EXIT
N:      .WORD   6
```

После выполнения программы на экране ЭЛТ отобразится число 6. Как можно создавать макросы, определенные пользователем, будет описано в следующем разделе.

6.3. МАКРОСЫ, ОПРЕДЕЛЕННЫЕ ПОЛЬЗОВАТЕЛЕМ

ФОРМАТ МАКРОСОВ

Общий формат макроса, определенного пользователем, показан ниже. Буквами x обозначены операторы основных инструкций.



Пример:

```
.MACRO  SUMINC  A,B,C   ;МАКРОС ДЛЯ СУММИРОВАНИЯ
                        ;И УВЕЛИЧЕНИЯ
        ADD     A,B
        INC     B
        ADD     B,C
        .ENDM
;ОСНОВНАЯ ПРОГРАММА
START:  MOV     #4,R0
        MOV     #1,R1
        SUMINC  #10,R0,R1
        SUMINC  X,Y,Z
        HALT
X:      .WORD   10.
Y:      .WORD   4
Z:      .WORD   2
        .END    START
```

После ассемблирования этой программы будет построен объектный файл. Для удобства покажем здесь мнемонический эквивалент объектного файла:

```
START:  MOV     #4,R0
        MOV     #1,R1
        ADD     #10,R0          ;ПЕРВОЕ РАСШИРЕНИЕ МАКРОСА SUMINC
        INC     R0
        ADD     R0,R1
        ADD     X,Y             ;ВТОРОЕ РАСШИРЕНИЕ МАКРОСА SUMINC
        INC     Y
        ADD     Y,Z
        HALT
X:      .WORD   10
Y:      .WORD   4
Z:      .WORD   2
        .END    START
```

Обратите внимание, что всякий раз, когда макрос используется, он заменяется или "расширяется" ассемблером тремя основными инструкциями. После выполнения программы получается следующее:

(R0) = 15
(R1) = 16
(X) = 10
(Y) = 15
(Z) = 17

АВТОМАТИЧЕСКИ СОЗДАВАЕМЫЕ ЛОКАЛЬНЫЕ СИМВОЛЫ

В некоторых случаях в макросе нам приходится использовать одну или несколько символьных адресных меток. Однако при многократном использовании такого макроса эти локальные символьные адреса могут создавать определенные трудности.

Например, в приведенной ниже программе метод макросов применяется для копирования блока данных из одного места в другое:

```
.MACRO  COPY ARRAY1,ARRAY2,M
                                ;M - ПОЛОЖИТЕЛЬНОЕ ЦЕЛОЕ
        MOV     #ARRAY1,R0      ;R0 - УКАЗАТЕЛЬ ИСХОДНЫХ ДАННЫХ
        MOV     #ARRAY2,R1      ;R1 - УКАЗАТЕЛЬ ДАННЫХ НАЗНАЧЕНИЯ
        MOV     #M,R2           ;R2 - СЧЕТЧИК ДАННЫХ, (R2)=M
LOOP:   MOV     (R0)+,(R1)+     ;ПЕРЕСЛАТЬ ДАННЫЕ
        SOB     R2,LOOP         ;ПРОДОЛЖИТЬ
        .ENDM
;ОСНОВНАЯ ПРОГРАММА
A1:     .WORD   D1,D2,...,DM
B1:     .BLKW   M
A2:     .WORD   DD1,DD2,...,DDN
B2:     .BLKW   N
START:  COPY    A1,B1,M
        COPY    A2,B2,N
        .END    START
```

После раширения программа будет иметь следующий вид:

```
;ОСНОВНАЯ ПРОГРАММА
A1:     .WORD   D1,D2 ...,DM
B1:     .BLKW   M
A2:     .WORD   DD1,DD2,...,DDN
B2:     .BLKW   N
START:  MOV     #A1,R0
        MOV     #B1,R1
        MOV     #M,R2   ;ПЕРВОЕ РАСШИРЕНИЕ
LOOP:   MOV     (R0)+,(R1)+
        SOB     R2,LOOP
        MOV     #A2,R0
```

```
        MOV     #B2,R1
        MOV     #N,R2    ;ВТОРОЕ РАСШИРЕНИЕ
LOOP:   MOV     (R0)+,(R1)+
        SOB     R2,LOOP
        .END    START
```

Обратите внимание, что после расширения в программе содержатся две идентичные символьные адресные метки LOOP. Конечно, это создает проблему, поскольку ассемблер будет рассматривать такую ситуацию как ошибочную из-за того, что адресное значение метки LOOP не является уникальным. Эту проблему можно разрешить, используя два метода. Первый метод вполне очевиден, но довольно неудобен; второй – более удобен. Эти два метода иллюстрируются следующими примерами.

Пример 1:

```
.MACRO  COPY ARRAY1,ARRAY2,M,LOOP
        MOV     #ARRAY1,R0
        MOV     #ARRAY2,R1
        MOV     #M,R2
LOOP:   MOV     (R0)+,(R1)+
        SOB     R2,LOOP
.ENDM
```

Обратите внимание, что первоначально LOOP – это "локальная" адресная метка в макросе. В данном случае мы рассматриваем ее как операнд макроса или внешнюю адресную метку, так что пользователь может управлять ею следующим образом:

```
;ОСНОВНАЯ ПРОГРАММА
A1:     .WORD   D1,D2,...,DM
B1:     .BLKW   M
A2:     .WORD   DD1,DD2,...,DDN
B2:     .BLKW   N
START:  COPY    A1,B1,M,L1
        COPY    A2,B2,N,L2
        .END    START
```

После расширения будем иметь:

```
A1:     .WORD   D1,D2,...,DM
B1:     .BLKW   M
A2:     .WORD   DD1,DD2,...,DDN
B2:     .BLKW   N
START:  MOV     #A1,R0
        MOV     #B1,R1
        MOV     #M,R2
L1:     MOV     (R0)+,(R1)+
        SOB     R2,L1
        MOV     #A2,R0
        MOV     #B2,R1
        MOV     #N,R2

L2:     MOV     (R0)+,(R1)+
        SOB     R2,L2
        .END    START
```

Обратите внимание на то, что каждая адресная метка определена уникально.

Пример 2. Этот пример показывает, как можно использовать возможность автоматического создания локальных символов, предоставляемую для решения этой проблемы операционной системой RSX-11M, разработанной для ЭВМ PDP-11. Мы лишь информируем ассемблер о том, какие и сколько локальных адресных символов используется внутри макроса, а задачу сделать все эти локальные адресные символы уникальными оставляем ассемблеру.

```
.MACRO  COPY ARRAY1,ARRAY2,M,?LOOP
        MOV     #ARRAY1,R0
        MOV     #ARRAY2,R1
        MOV     #M,R2
LOOP:   MOV     (R0)+,(R1)+
        SOB     R2,LOOP
.ENDM
```

Обратите внимание на то, что в этом примере перед локальным адресным символом мы использовали знак вопроса. Он означает, что ассемблер должен назначать уникальную адресную метку каждый раз, когда вызывается макрос.

```
;ОСНОВНАЯ ПРОГРАММА
A1:     .WORD   D1,D2,...,DM
B1:     .BLKW   M
A2:     .WORD   DD1,DD2,...,DDN
B2:     .BLKW   N
START:  COPY    A1,B1,M
        COPY    A2,B2,N
        HALT
        .END    START

;ОСНОВНАЯ ПРОГРАММА ПОСЛЕ РАСШИРЕНИЯ
A1:     .WORD   D1,D2,...,DM
B1:     .BLKW   M
A2:     .WORD   DD1,DD2,...,DDN
B2:     .BLKW   N
START:  MOV     #A1,R0
        MOV     #B1,R1
        MOV     #M,R2
64$:    MOV     (R0)+,(R1)+     ;НАЗНАЧЕН ПЕРВЫЙ АВТОМАТИЧЕСКИЙ
                                ;ЛОКАЛЬНЫЙ АДРЕС
        SOB     R2,64$
        MOV     #A2,R0
        MOV     #B2,R1
        MOV     #N,R2

65$:    MOV     (R0)+,(R1)+     ;НАЗНАЧЕН ВТОРОЙ АВТОМАТИЧЕСКИЙ
                                ;ЛОКАЛЬНЫЙ АДРЕС
        SOB     R2,65$
        HALT
        .END    START
```

Обратите внимание на то, что локальные адресные символы, автоматически назначенные ассемблером, имеют формат n \$, где $64 \leqslant n \leqslant 127$ в десятичном представлении.

Пример 3. Во многих приложениях нам требуется временно сохранить содержимое всех регистров общего назначения R0, . . . , R5 в стеке и через какое-то время восстановить это содержимое. Поскольку для этого необходимо простые действия, вполне применим метод макросов. Ниже показано, как можно дополнить программу примера 2 макросами, чтобы обеспечить сохранение содержимого регистров (SAVE) и восстановление (UNSAVE). В данном случае мы предполагаем, что перед использованием макроса COPY основная программа держит в регистрах R0, R1 и R2 важную информацию:

```
.MACRO  SAVREG          ;СОХРАНИТЬ РЕГИСТРЫ
        MOV     R0,-(SP)
        MOV     R1,-(SP)
        MOV     R2,-(SP)
.ENDM   SAVREG
.MACRO  RETRREG         ;ВОССТАНОВИТЬ РЕГИСТРЫ
        MOV     (SP)+,R2
        MOV     (SP)+,R1
        MOV     (SP)+,R0
.ENDM   RETRREG
```

Обратите внимание, что из-за организации стековой памяти по принципу "последним вошел – первым вышел" процесс восстановления содержимого регистров мы должны записывать в обратном порядке.

```
.MACRO  COPY ARRAY1,ARRAY2,N,?LOOP
        MOV     #ARRAY1,R0
        MOV     #ARRAY2,R1
        MOV     #N,R2    ;ЧТОБЫ ПОКАЗАТЬ АЛЬТЕРНАТИВНЫЙ ВАРИАНТ,
                         ;МЫ ЗДЕСЬ ТРАКТУЕМ N КАК АДРЕСНУЮ МЕТКУ
LOOP:   MOV     (R0)+,(R1)+
        SOB     R2,LOOP
.ENDM   COPY

;ОСНОВНАЯ ПРОГРАММА
A1:     .WORD   D1,D2,...,DM
B1:     .BLKW   M
A2:     .WORD   DD1,DD2,...,DDM
B2:     .BLKW   M
N:      .WORD   M
START:  .
        .
        .
        SAVREG
        COPY    A1,B1,N
        COPY    A2,B2,N
        RETREG
        .
        .
        .
        .END  START
```

Пример 4. Это – пример применения и системных макросов, и макросов, определенных пользователем. Программа использует системные макросы: 1).WRITE – для отображения на экране ЭЛТ содержимого конкретной ячейки памяти; 2).RAD – для определения пользователем того основания системы счисления, которое он хочет использовать. Эта программа является интерактивной.

```
.MCALL  .RAD,.READ,.WRITE,.EXIT ;ВЫЗОВ СИСТЕМНЫХ МАКРОСОВ
.MACRO  SAVREG                  ;МАКРОСЫ, ОПРЕДЕЛЕННЫЕ ПОЛЬЗОВАТЕЛЕМ
        MOV     R0,-(SP)
        MOV     R1,-(SP)
        MOV     R2,-(SP)
.ENDM   SAVREG
.MACRO  RETREG
        MOV     (SP)+,R2
        MOV     (SP)+,R1
        MOV     (SP)+,R0
.ENDM   RETREG
.MACRO  COPY ARRAY1,ARRAY2,N,?LOOP
        MOV     #ARRAY1,R0
        MOV     #ARRAY2,R1
        MOV     #N,R2
LOOP:   MOV     (R0)+,(R1)+
        SOB     R2,LOOP
.ENDM   COPY
;ОСНОВНАЯ ПРОГРАММА
1) START: .WRITE "\\ВВЕДИТЕ_ЖЕЛАЕМОЕ_ОСНОВАНИЕ_И_ФОРМАТ:_
2) .RAD
3) .WRITE "\\ВВЕДИТЕ_3_ЧИСЛА:_
4) .READ #A1
5) .WRITE #^B"ВВЕДИТЕ_3_СИМВОЛА:_
6) .READ #A2
7) SAVREG
```

```
8) COPY A1,B1,N
9) RETREG
10) .WRITE B1,B1+2,B1+4
11) SAVREG
12) COPY A2,B2,N
13) RETREG
14) .WRITE #B2
15) .EXIT
16) A1: .BLKW 3
    A2: .BLKW 3
    B1: .BLKW 3
    B2: .BLKW 3
    N:  .WORD 3
    .END START
```

Давайте проследим работу программы и посмотрим, как функционируют эти системные макросы.

Строка 1 отобразит на экране ЭЛТ следующее сообщение: ВВЕДИТЕ ЖЕЛАЕМОЕ ОСНОВАНИЕ И ФОРМАТ:.

Строка 2 вынуждает ЭВМ постоянно проверять ввод с клавиатуры. Если пользователь ничего не вводит с клавиатуры, то ЭВМ будет просто ожидать ввода. Предположим, пользователь печатает 8L <ВК>. ЭВМ интерпретирует это следующим образом: пользователь хочет использовать восьмеричное основание системы счисления и формат, выравниваемый по левому краю.

Строка 3 отобразит сообщение: ВВЕДИТЕ 3 ЧИСЛА:.

Строка 4 заставляет ЭВМ ждать, пока пользователь не введет три числа с клавиатуры. Предположим, что пользователь печатает 1, 2, 3. Тогда макрос .READ воспримет эти числа и запомнит их соответственно в ячейках A1, A1 + 2 и A1 +4.

Строка 5 отображает следующее сообщение: ВВЕДИТЕ 3 СИМВОЛА:.

Строка 6 заставляет ЭВМ ждать ввода с клавиатуры. Предположим, пользователь печатает LIN. Тогда макрос .READ принимает эти три напечатанных символа и запоминает их соответственно в ячейках A2, A2 + 2 и A2 + 4.

Строка 7 сохраняет в стеке (R0), (R1) и (R2).

Строка 8 копирует содержимое ячеек A1, A1 + 2 и A1 + 4 в ячейки B1, B1 + 2, и B1 +4.

Строка 9 восстанавливает (R2), (R1) и (R0).

Строка 10 отображает содержимое ячеек B1, B1 + 2 и B1 + 4. В этом месте работы программы мы ожидаем увидеть на экране следующее: 123.

Строка 11 сохраняет в стеке (R0), (R1) и (R2).

Строка 12 копирует (A2), (A2 + 2) и (A2 + 4) в ячейки B2, B2 + 2 и B2 +4.

Строка 13 восстанавливает (R2), (R1) и (R0).

Строка 14 отображает содержимое ячеек B2, B2 + 2 и B2 +4. В этом месте работы программы мы ожидаем увидеть на экране следующее: LIN.

Строка 15 осуществляет выход из программы.

На рис. 6.2, *а* показаны исходная программа и результат ее выполнения для вышеприведенного примера. На рис. 6.2, *б* дана исходная программа после расширения макросов.

```
;ПРИМЕР, ПОКАЗЫВАЮЩИЙ СОВМЕСТНОЕ ИСПОЛЬЗОВАНИЕ СИСТЕМНЫХ МАКРОСОВ
;И МАКРОСОВ, ОПРЕДЕЛЕННЫХ ПОЛЬЗОВАТЕЛЕМ
;
;
.MCALL  .RAD,.READ,.WRITE,.EXIT
.MACRO  SAVREG
        MOV     R0,-(SP)
        MOV     R1,-(SP)
```

```
        MOV     R2,-(SP)
.ENDM   SAVREG
.MACRO  RETRREG
        MOV     (SP)+,R2
        MOV     (SP)+,R1
        MOV     (SP)+,R0
.ENDM   RETRREG
.MACRO  COPY ARRAY1,ARRAY2,N,?LOOP
        SAVREG
        MOV     #ARRAY1,R0
        MOV     #ARRAY2,R1
        MOV     N,R2
LOOP:   MOV     (R0)+,(R1)+
        SOB     R2,LOOP
        RETRREG
.ENDM   COPY
;
;ОСНОВНАЯ ПРОГРАММА
START:  .WRITE "\\ВВЕДИТЕ_ЖЕЛАЕМОЕ_ОСНОВАНИЕ_И_ФОРМАТ:_
        .RAD
        .WRITE "\\ВВЕДИТЕ_3_ЧИСЛА:_
        .READ "A1
        .WRITE "\\ВВЕДИТЕ_3_СИМВОЛА:_
        .READ "A2
        .LIST
        COPY A1,B1,N
        .WRITE B1,B1+2,B1+4
        COPY A2,B2,N
        .WRITE "B2
        .EXIT
A1:     .BLKW 3
A2:     .BLKW 3
B1:     .BLKW 3
B2:     .BLKW 3
N:      .WORD 3
        .END    START

РЕЗУЛЬТАТ ВЫПОЛНЕНИЯ:   ВВЕДИТЕ ЖЕЛАЕМОЕ ОСНОВАНИЕ И ФОРМАТ: 10
                        ВВЕДИТЕ 3 ЧИСЛА: 1 3 4
                        ВВЕДИТЕ 3 СИМВОЛА: LIN
                        1 3 4 LIN
```

Рис. 6.2, *а*. Пример совместного использования макросов, определенных системно и определенных пользователем

```
;ПРИМЕР ВЛОЖЕНИЯ МАКРОСОВ
;
;
.MCALL  .RAD,.READ,.WRITE,.EXIT
.MACRO  SAVREG
        MOV     R0,-(SP)
        MOV     R1,-(SP)
        MOV     R2,-(SP)
.ENDM   SAVREG
.MACRO  RETRREG
        MOV     (SP)+,R2
        MOV     (SP)+,R1
        MOV     (SP)+,R0
.ENDM   RETRREG
.MACRO  COPY ARRAY1,ARRAY2,N,?LOOP
        SAVREG
        MOV     #ARRAY1,R0
        MOV     #ARRAY2,R1
        MOV     N,R2
```

```
LOOP:   MOV     (R0)+,(R1)+
        SOB     R2,LOOP
        RETURN
.ENDM   COPY
;
;ОСНОВНАЯ ПРОГРАММА
START:  .WRITE  #\\ВВЕДИТЕ ЖЕЛАЕМОЕ ОСНОВАНИЕ И ФОРМАТ:_
        .READ
        .WRITE  #\\ВВЕДИТЕ 3 ЧИСЛА:_
        .READ   #A1
        .WRITE  #\\ВВЕДИТЕ 3 СИМВОЛА:_
        .READ   #A2
        COPY    A1,B1,N
        SAVREG
        MOV     R0,-(SP)
        MOV     R1,-(SP)
        MOV     R2,-(SP)
        MOV     #A1,R0
        MOV     #B1,R1
        MOV     N,R2
64$:    MOV     (R0)+,(R1)+
        SOB     R2,64$
        RETURN
        MOV     (SP)+,R2
        MOV     (SP)+,R1
        MOV     (SP)+,R0
        .WRITE B1,B1+2,B1+4
        COPY A2,B2,N
        SAVREG
        MOV     R0,-(SP)
        MOV     R1,-(SP)
        MOV     R2,-(SP)
        MOV     #A2,R0
        MOV     #B2,R1
        MOV     N,R2
64$:    MOV     (R0)+,(R1)+
        SOB     R2,64$
        RETURN
        MOV     (SP)+,R2
        MOV     (SP)+,R1
        MOV     (SP)+,R0
        .WRITE #B2
        .EXIT
A1:     .BLKW 3
A2:     .BLKW 3
B1:     .BLKW 3
B2:     .BLKW 3
N:       .WORD 3
        .END    START
```

Рис. 6.2, *б*. Исходная программа рис. 6.2, *а* после расширения макросов

ВЛОЖЕНИЕ МАКРОСОВ

Как и в случае подпрограммы, макрос может содержать другой макрос, другой макрос — еще один макрос и т. д., т. е. макросы могут быть вложенными. Следующий пример показывает **вложение макросов.**

В этом примере мы демонстрируем, как может быть определен макрос, позволяющий подсчитывать реальное количество времени для 1с, для 1мин, для 1ч, если основываться на информации о времени выполнения набора инструкций из руководств для пользователей ЭВМ PDP-11 или из приложения В.

```
.MACRO  SEC ?L1
        MOV     R0,-(SP)        ;СМ. ЗАМЕЧАНИЕ 1
        CLR     R0              ;СМ. ЗАМЕЧАНИЕ 2
L1:     INC     R0              ;СМ. ЗАМЕЧАНИЕ 3
        CMP     #0,R0           ;СМ. ЗАМЕЧАНИЕ 4
        NOP                     ;СМ. ЗАМЕЧАНИЕ 5
        BNE     L1              ;СМ. ЗАМЕЧАНИЕ 6
        MOV     (SP)+,R0        ;СМ. ЗАМЕЧАНИЕ 7
.ENDM   SEC
```

Мы можем использовать этот макрос для построения "часов реального времени" на 1с. Ниже даются временные вычисления.

В общем случае время, требуемое для выполнения инструкции, можно оценить по следующей формуле:

$T(i) = t_1 + t_2 + t_3$; В МИКРОСЕКУНДАХ

где $t_1 = t_{10} + t_{11} + t_{12}$;

$T(i)$ – полное время извлечения и выполнения инструкции;

t_1 – основное время;

t_2 – время обработки исходного операнда;

t_3 – время обработки операнда назначения.

t_{10} – время извлечения;

t_{11} – время декодирования;

t_{12} – время выполнения;

Время, необходимое для выполнения показанных выше инструкций, вычисляется так.

Замечание 1: сохранение старого содержимого регистра R0
$T(1) = 2.45 + 0 + 2.8 = 5.25$

Замечание 2: $T(2) = 3.85 + 0 + 0 = 3.85$

Замечание 3: начало подсчета –
$T(3) = 4.2 + 0 + 0 = 4.2$

Замечание 4: $T(4) = 3.5 + 1.4 + 0 = 4.9$

Замечание 5: $T(5) = 3.5$

Замечание 6: $T(6) = 3.5 + 0 + 0 = 3.5$

Замечание 7: $T(7) = 3.5 + 1.4 + 0 = 4.9$

Обратите внимание, что инструкции 3–6 образуют временной цикл. Программа не выйдет из цикла подсчета до тех пор, пока содержимое регистра R0 вновь не будет равно нулю. Поскольку время, необходимое для инструкций 1, 2 и 7, незначительно по сравнению с временем цикла для инструкций 3 – 6, то им можно пренебречь. Таким образом, полное время цикла можно оценить так:

Полное время цикла для 16-битового регистра R0 приблизительно равно $2^{16} = 65.5 \times \times 10^3$.

Время каждого прохождения цикла равно $T(3) + T(4) + T(5) + T(6) = 4.2 + 4.9 + + 3.5 + 3.5 = 16.1$.

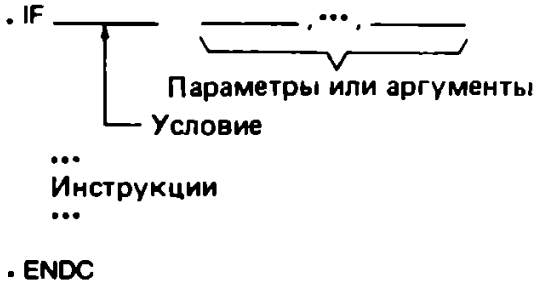
Полное время цикла равно $65.5 \times 10^3 \times 16.1 \approx 1054$ мс $\approx$ 1с. Теперь, используя вложение макросов, легко определить макрос для 1мин и макрос для 1ч:

```
.MACRO  MIN ?L2
        MOV     R1,-(SP)        ;СОХРАНИТЬ (R1)
        CLR     R1              ;ИНИЦИАЛИЗИРОВАТЬ R1
L2:     SEC                     ;ВЛОЖЕНИЕ МАКРОСА СЕКУНДЫ
        INC     R1
        CMP     #60.,R1         ;ДОСТИГЛИ 60. СЕКУНД?
        BNE     L2
        MOV     (SP)+,R1        ;ВОССТАНОВИТЬ (R1)
.ENDM   MIN
```

```
.MACRO  HOUR ?L3
        MOV     R2,-(SP)        ;СОХРАНИТЬ (R2)
        CLR     R2              ;ИНИЦИАЛИЗИРОВАТЬ R2
L3:     MIN                     ;ВЛОЖЕНИЕ МАКРОСА СЕКУНДЫ И МИНУТЫ
        INC     R2
        CMP     #60.,R2         ;ДОСТИГЛИ 60. МИНУТ?
        BNE     L3
        MOV     (SP)+,R2        ;ВОССТАНОВИТЬ (R2)
.ENDM   HOUR
```

УСЛОВНЫЕ МАКРОСЫ

На практике нам нужны очень гибкие макросы, обладающие многими возможностями. При этом макросы становятся все более длинными и поэтому требующими все большего объема памяти плюс время на ассемблирование и выполнение. Мы можем располагать многофункциональным макросом со многими возможностями, но мы редко пользуемся ими всеми в одно и то же время. Следовательно, когда макрос содержит особенности, реально не нужные нам в данном конкретном приложении, имеют место потери объема памяти и времени обработки. В этом случае система PDP-11 предлагает нам **условные макросы**. В качестве простого примера давайте определим макросы для сохранения и восстановления регистров общего назначения. Мы используем все регистры, но обеспечим условия, позволяющие пользователю выборочно сохранять и восстанавливать лишь те регистры, которые ему действительно необходимы, не пользуясь другими регистрами. Сначала нам необходим общий формат для определения условного макроса:



Пример 1. Условное сохранение-восстановление регистров.

```
.MACRO  SAVE R0,R1,R2,R3,R4,R5
.IF NB R0       ;АССЕМБЛИРОВАТЬ ПОСЛЕДУЮЩЕЕ, ЕСЛИ НЕ ПУСТО R0
MOV R0,-(SP)    ;ПРОТОЛКНУТЬ R0
.ENDC
.IF NB R1
MOV R1,-(SP)
.ENDC
.IF NB R2
MOV R2,-(SP)
.ENDC
.IF NB R3
MOV R3,-(SP)
.ENDC
.IF NB R4
MOV R4,-(SP)
.ENDC
.IF NB R5
MOV R5,-(SP)
.ENDC
.ENDM   SAVE
```

Аналогично можно определить соответствующий макрос для восстановления регистров:

```
.MACRO  RTRV R5,R4,R3,R2,R1,R0
.IF NB R5
MOV (SP)+,R5     ;ВЫТОЛКНУТЬ R5
.ENDC
.
.
.
.IF NB R0
MOV (SP)+,R0
.ENDC
.ENDM   RTRV
;ОСНОВНАЯ ПРОГРАММА
START:  .
        .
        .
        SAVE    R0,R5
        .
        .
        .
        RTRV    R5,R0
        .
        .
        .
.END    START
```

После ассемблирования основной программы макросы для сохранения и восстановления регистров будут расширены следующим образом:

MOV R0, –(SP)
MOV R5, –(SP)

для макроса SAVE и

MOV (SP)+, R5
MOV (SP)+, R0

для макроса RTRV.

Пример 2. В этом примере мы модифицируем макрос, предназначенный для подсчета интервалов времени, для иллюстрации использования вложенных условных макросов. На рис. 6.3 показана исходная программа с макросом, определенным пользователем, который может использоваться для формирования временной задержки в часах, минутах и секундах, задаваемых операндами. Обратите внимание, что в основной программе мы использовали директивы .LIST и .NLIST для управления выводом или невыводом информации о макросах в файл листинга. Поскольку расширение системного макроса .WRITE интереса для нас не представляет, мы расположили перед ним директиву .NLIST. Однако мы хотим увидеть, как происходит расширение макроса временной задержки при различных условиях. Поэтому мы расположили перед ним директиву .LIST. Результат расширения условного макроса для подсчета интервалов времени показан на рис. 6.4. Из рисунка ясно, каким образом при разных условиях влияют на расширение макроса локальные символьные адреса и макросы.

```
;ЭТА ПРОГРАММА ПОКАЗЫВАЕТ ИСПОЛЬЗОВАНИЕ УСЛОВНЫХ ВЛОЖЕННЫХ МАКРОСОВ
.MCALL  .WRITE,.EXIT
.MACRO  DELAY S,M,H,?LS,LM,LH   ;СЕКУНДЫ, МИНУТЫ И ЧАСЫ
.MACRO  SEC ?L1
        MOV     R0,-(SP)
        CLR     R0
L1:     INC     R0
        CMP     #0,R0
        BNE     L1
        MOV     (SP)+,R0
.ENDM   SEC
;
```

```
.MACRO  MIN ?L2
        MOV     R1,-(SP)
        CLR     R1
L2:     SEC
        INC     R1
        CMP     #60.,R1
        BNE     L2
        MOV     (SP)+,R1
.ENDM   MIN
;
.IF NB S
        MOV     R2,-(SP)
        MOV     #S,R2
LS:     SEC
        SOB     R2,LS
        MOV     (SP)+,R2
.ENDC
.IF NB M
        MOV     R3,-(SP)
        MOV     #M,R2
LM:     MIN
        SOB     R3,LM
        MOV     (SP)+,R3
.ENDC
.IF NB H
        MOV     R4,-(SP)
        MOV     #H,R4
        MOV     R5,-(SP)
        CLR     R5
LH:     MIN
        INC     R5
        CMP     #60.,R5
        BNE     LH
        SOB     R4,LH
        MOV     (SP)+,R5
        MOV     (SP)+,R4
.ENDC
.ENDM   DELAY
;ОСНОВНАЯ ПРОГРАММА
START:  .LIST
        DELAY   1       ;ЗАДЕРЖКА НА 1 С
        .NLIST
        .WRITE  "\\ПРОШЛА_ОДНА_СЕКУНДА
        .LIST
        DELAY   ,1      ;ЗАДЕРЖКА НА 1 М
        .NLIST
        .WRITE  "\\ПРОШЛА_ОДНА_МИНУТА
        .LIST
        DELAY   ,,1     ;ЗАДЕРЖКА НА 1 Ч
        .NLIST
        .WRITE  "\\ПРОШЕЛ_ОДИН_ЧАС
        .LIST
        DELAY   1,1,1   ;ЗАДЕРЖКА НА 1 Ч, 1 М И 1 С
        .NLIST
        .WRITE  "\\ПРОШЕЛ_ОДИН_ЧАС,_ОДНА_МИНУТА_И_ОДНА_СЕКУНДА
        .EXIT
        .END    START

РЕЗУЛЬТАТ ВЫПОЛНЕНИЯ:  ПРОШЛА ОДНА СЕКУНДА      ;ПОСЛЕ ОДНОЙ СЕКУНДЫ
                       ПРОШЛА ОДНА МИНУТА       ;ПОСЛЕ ОДНОЙ МИНУТЫ
                       ПРОШЕЛ ОДИН ЧАС          ;ПОСЛЕ ОДНОГО ЧАСА
                       ПРОШЕЛ ОДИН ЧАС, ОДНА МИНУТА И ОДНА СЕКУНДА
```

Рис. 6.3. Получение временной задержки при использовании вложенных условных макросов

```
;ЭТА ПРОГРАММА ПОКАЗЫВАЕТ ИСПОЛЬЗОВАНИЕ УСЛОВНЫХ ВЛОЖЕННЫХ МАКРОСОВ
.MCALL  .WRITE,.EXIT
.MACRO  DELAY S,M,H,?LS,LM,LH   ;СЕКУНДЫ, МИНУТЫ И ЧАСЫ
.MACRO  SEC ?L1
        MOV     R0,-(SP)
        CLR     R0
L1:     INC     R0
        CMP     #0,R0
        BNE     L1
        MOV     (SP)+,R0
.ENDM   SEC
;
.MACRO  MIN ?L2
        MOV     R1,-(SP)
        CLR     R1
L2:     SEC
        INC     R1
        CMP     #60.,R1
        BNE     L2
        MOV     (SP)+,R1
.ENDM   MIN
;
.IF NB S
        MOV     R2,-(SP)
        MOV     #S,R2
LS:     SEC
        SOB     R2,LS
        MOV     (SP)+,R2
.ENDC
.IF NB M
        MOV     R3,-(SP)
        MOV     #M,R3
LM:     MIN
        SOB     R3,LM
        MOV     (SP)+,R3
.ENDC
.IF NB H
        MOV     R4,-(SP)
        MOV     #H,R4
        MOV     R5,-(SP)
        CLR     R5
LH:     MIN
        INC     R5
        CMP     #60.,R5
        BNE     LH
        SOB     R4,LH
        MOV     (SP)+,R5
        MOV     (SP)+,R4
.ENDC
.ENDM   DELAY
;ОСНОВНАЯ ПРОГРАММА
START:  DELAY   1       ;ЗАДЕРЖКА НА 1 С
.MACRO  SEC ?L1
        MOV     R0,-(SP)
        CLR     R0
L1:     INC     R0
        CMP     #0,R0
        BNE     L1
        MOV     (SP)+,R0
.ENDM   SEC
;
.MACRO  MIN ?L2
        MOV     R1,-(SP)
        CLR     R1
```

```
L2:     SEC
        INC     R1
        CMP     #60.,R1
        BNE     L2
        MOV     (SP)+,R1
.ENDM   MIN
;
.IF NB 1
        MOV     R2,-(SP)
        MOV     #1,R2
64$:    SEC
        MOV     R0,-(SP)
        CLR     R0
67$:    INC     R0
        CMP     #0,R0
        BNE     67$
        MOV     (SP)+,R0
        SOB     R2,64$
        MOV     (SP)+,R2
.ENDC
.IF NB
        MOV     R3,-(SP)
        MOV     #,R3
65$:    MIN
        SOB     R3,65$
        MOV     (SP)+,R3
.ENDC
.IF NB
        MOV     R4,-(SP)
        MOV     #,R4
        MOV     R5,-(SP)
        CLR     R5
66$:    MIN
        INC     R5
        CMP     #60.,R5
        BNE     66$
        SOB     R4,66$
        MOV     (SP)+,R5
        MOV     (SP)+,R4
.ENDC
        .WRITE  "\\ПРОШЛА_ОДНА_СЕКУНДА
        DELAY   ,1      ;ЗАДЕРЖКА НА 1 М
.MACRO  SEC ?L1
        MOV     R0,-(SP)
        CLR     R0
L1:     INC     R0
        CMP     #0,R0
        BNE     L1
        MOV     (SP)+,R0
.ENDM   SEC
;
.MACRO  MIN ?L2
        MOV     R1,-(SP)
        CLR     R1
L2:     SEC
        INC     R1
        CMP     #60.,R1
        BNE     L2
        MOV     (SP)+,R1
.ENDM   MIN
;
.IF NB
        MOV     R2,-(SP)
        MOV     #,R2
```

```
64$:    SEC
        SOB     R2,64$
        MOV     (SP)+,R2
.ENDC
.IF NB 1
        MOV     R3,-(SP)
        MOV     #1,R3
65$:    MIN
        MOV     R1,-(SP)
        CLR     R1
67$:    SEC
        MOV     R0,-(SP)
        CLR     R0
68$:    INC     R0
        CMP     #0,R0
        BNE     68$
        MOV     (SP)+,R0
        INC     R1
        CMP     #60.,R1
        BNE     67$
        MOV     (SP)+,R1
        SOB     R3,65$
        MOV     (SP)+,R3
.ENDC
.IF NB
        MOV     R4,-(SP)
        MOV     #,R4
        MOV     R5,-(SP)
        CLR     R5
66$:    MIN
        INC     R5
        CMP     #60.,R5
        BNE     66$
        SOB     R4,66$
        MOV     (SP)+,R5
        MOV     (SP)+,R4
.ENDC
        .WRITE  "\\ПРОШЛА_ОДНА_МИНУТА
        DELAY   ,,1     ;ЗАДЕРЖКА НА 1 Ч
.MACRO  SEC ?L1
        MOV     R0,-(SP)
        CLR     R0
L1:     INC     R0
        CMP     #0,R0
        BNE     L1
        MOV     (SP)+,R0
.ENDM   SEC
;
.MACRO  MIN ?L2
        MOV     R1,-(SP)
        CLR     R1
L2:     SEC
        INC     R1
        CMP     #60.,R1
        BNE     L2
        MOV     (SP)+,R1
.ENDM   MIN
;
.IF NB
        MOV     R2,-(SP)
        MOV     #,R2
64$:    SEC
        SOB     R2,64$
        MOV     (SP)+,R2
```

```
.ENDC
.IF NB
        MOV     R3,-(SP)
        MOV     #,R3
65$:    MIN
        SOB     R3,65$
        MOV     (SP)+,R3
.ENDC
.IF NB 1
        MOV     R4,-(SP)
        MOV     #,R4
        MOV     R5,-(SP)
        CLR     R5
66$:    MIN
        MOV     R1,-(SP)
        CLR     R1
67$:    SEC
        MOV     R0,-(SP)
        CLR     R0
68$:    INC     R0
        CMP     #0,R0
        BNE     68$
        MOV     (SP)+,R0
        INC     R1
        CMP     #60.,R1
        BNE     67$
        MOV     (SP)+,R1
        INC     R5
        CMP     #60.,R5
        BNE     66$
        SOB     R4,66$
        MOV     (SP)+,R5
        MOV     (SP)+,R4
.ENDC
        .WRITE  "\\ПРОШЕЛ_ОДИН_ЧАС
        DELAY   1,1,1    ;ЗАДЕРЖКА НА 1 Ч, 1 М И 1 С
.MACRO  SEC ?L1
        MOV     R0,-(SP)
        CLR     R0
L1:     INC     R0
        CMP     #0,R0
        BNE     L1
        MOV     (SP)+,R0
.ENDM   SEC
;
.MACRO  MIN ?L2
        MOV     R1,-(SP)
        CLR     R1
L2:     SEC
        INC     R1
        CMP     #60.,R1
        BNE     L2
        MOV     (SP)+,R1
.ENDM   MIN
;
.IF NB 1
        MOV     R2,-(SP)
        MOV     #1,R2
64$:    SEC
        MOV     R0,-(SP)
        CLR     R0
67$:    INC     R0
        CMP     #0,R0
        BNE     67$
```

```
        MOV     (SP)+,R0
        SOB     R2,64$
        MOV     (SP)+,R2
.ENDC
.IF NB 1
        MOV     R3,-(SP)
        MOV     #1,R3
65$:    MIN
        MOV     R1,-(SP)
        CLR     R1
68$:    SEC
        MOV     R0,-(SP)
        CLR     R0
69$:    INC     R0
        CMP     #0,R0
        BNE     69$
        MOV     (SP)+,R0
        INC     R1
        CMP     #60.,R1
        BNE     68$
        MOV     (SP)+,R1
        SOB     R3,65$
        MOV     (SP)+,R3
.ENDC
.IF NB 1
        MOV     R4,-(SP)
        MOV     #1,R4
        MOV     R5,-(SP)
        CLR     R5
66$:    MIN
        MOV     R1,-(SP)
        CLR     R1
70$:    SEC
        MOV     R0,-(SP)
        CLR     R0
71$:    INC     R0
        CMP     #0,R0
        BNE     71$
        MOV     (SP)+,R0
        INC     R1
        CMP     #60.,R1
        BNE     70$
        MOV     (SP)+,R1
        INC     R5
        CMP     #60.,R5
        BNE     66$
        SOB     R4,66$
        MOV     (SP)+,R5
        MOV     (SP)+,R4
.ENDC
        .WRITE  "\\ПРОШЕЛ_ОДИН_ЧАС,_ОДНА_МИНУТА_И_ОДНА_СЕКУНДА
        .EXIT
        .END    START
```

Рис. 6.4. Макрорасширение условного макроса временно́й задержки

Возможные условия. Система PDP-11 помимо проиллюстрированного здесь условия NB (не пусто) обеспечивает еще целый ряд условий. Подробности можно найти в руководствах для пользователей. Выдержка из доступных условий показана в следующей таблице:

Условие	Аргументы	Блок расширяется, если аргументы:
NB	Выражения	не пусты
B	Выражения	пусты
EQ	Выражения	равны нулю
NE	Выражения	не равны нулю
GT	Выражения	больше нуля
LT	Выражения	меньше нуля
DF	Выражения	определены
NDF	Выражения	не определены

Пример:

```
.IF EQ ALPHA+1                  ;ЭТОТ БЛОК РАСШИРЯЕТСЯ, ЕСЛИ
.                               ;И ТОЛЬКО ЕСЛИ ALPHA+1 = 0
.
.
.ENDC
.IF DF СИМВОЛ1 & СИМВОЛ2        ;ЭТОТ БЛОК РАСШИРЯЕТСЯ, ЕСЛИ
.                               ;И ТОЛЬКО ЕСЛИ ОПРЕДЕЛЕНЫ
.                               ;ОБА СИМВОЛА
.
.ENDC
```

6.4. УПРАЖНЕНИЯ

1. Кратко объясните различия во времени выполнения и в занимаемом объеме памяти для подпрограмм и макроинструкций.
2. Кратко обсудите различие между макроинструкциями, определенными системой и определенными пользователем.
3. Используя приложение В, напишите макрос, определенный пользователем, который будет давать временну́ю задержку, равную 1мс.
4. Напишите условный макрос, который будет добавлять бит четности к самому старшему биту 7-битового символа кода ASCII в байтовом представлении. (По желанию пользователя может устанавливаться условие ЧЕТ или НЕЧЕТ.)

ГЛАВА 7
ПРОГРАММИРОВАНИЕ ВВОДА-ВЫВОДА

7.1. ВВЕДЕНИЕ

Программирование ввода-вывода — это одно из таких приложений, в котором широко используется язык ассемблера для обеспечения эффективности и высокой скорости работы. Связь периферийных устройств или устройств ввода-вывода с ЦП в немалой степени реализуется с помощью программного обеспечения. Без хотя бы одного устройства ввода-вывода ЦП бесполезен, независимо от того, насколько он мощный. Существует много типов устройств ввода-вывода. В зависимости от приложения они варьируются от устройств специального назначения, таких как оптический распознаватель символов или распознаватель-синтезатор речи, до устройств общего назначения, таких как терминал, печатающее устройство или привод магнитного диска.

Но любое устройство ввода-вывода может рассматриваться как некий элемент аппаратуры или черный ящик, преобразующий информацию физического мира в логические сигналы нулей и единиц, которые вводятся в вычислительную систему для обработки данных. И, наоборот, вычислительная система выдает информацию устройствам ввода-вывода в виде логических нулей и единиц, и уже эти устройства реконструи-

руют информацию в виде сигналов физического мира. Устройством ввода-вывода может быть просто реле, которое срабатывает от логической единицы и отпускает от логического нуля, или сложный цветной графический терминал, способный высвечивать на экране ЭЛТ движущиеся картинки.

Программистам нет необходимости интересоваться подробной аппаратной структурой устройства, а нужно знать только формат и время транспортировки данных. Основываясь на этой информации, программисты могут разработать программы ввода-вывода и установить их в системе для прикладных целей. В этой главе мы подробно, с примерами, опишем методику программирования ввода-вывода.

ОБЩИЕ УСТРОЙСТВА ВВОДА-ВЫВОДА

Существует набор устройств ввода-вывода, который в настоящее время считается основным, или общим, для любой вычислительной системы: терминал с клавиатурой и ЭЛТ, привод перфоленты, привод магнитной ленты, привод магнитного диска и АЦПУ. Среди них больше всего нам знаком терминал. Он генерирует и принимает коды ASCII по одному символу в одном байте.

Привод перфоленты – это устройство, которое может перфорировать и считывать перфоленту в коде ASCII или других кодах. Магнитная лента рассматривается как устройство массовой памяти, поскольку на нее можно записывать и с нее можно считывать информацию большого объема. Однако и привод перфоленты, и привод магнитной ленты обеспечивает только последовательный доступ к информации. То есть, если вы хотите осуществить доступ к информации, которая волей случая записана в конце ленты, то для ее извлечения может потребоваться большое количество времени.

Привод магнитного диска также является устройством массовой памяти, но он считается устройством прямого доступа к памяти. Его работа напоминает проигрыватель грампластинок. Информация записывается на круглой пластине, покрытой магнитным слоем; ее адресуют по номеру сектора и номеру дорожки. Запись и чтение информации производятся с помощью набора магнитных головок с фиксированной позицией или с помощью одной головки, перемещаемой в радиальном направлении. Поскольку стоимость таких устройств постоянно снижается, сейчас почти в любой вычислительной системе можно поставить хотя бы один привод гибких магнитных дисков. Все системное программное обеспечение и программные файлы хранятся на диске. Для выполнения они могут быть вызваны или загружены в оперативную память соответствующими системными программами. Программы ввода-вывода для дисковых приводов обычно разрабатываются на языке ассемблера и устанавливаются в системе фирмой-производителем.

АЦПУ – другое важное устройство, дающее нам возможность получать информацию в виде бумажных документов для долговременного хранения. Популярными устройствами являются аналого-цифровые и цифро-аналоговые преобразователи (АЦП и ЦАП). Если уж мы затронули эти специализированные устройства, то можно сказать, что в настоящее время во многих системах АЦП и ЦАП, занимающие одну печатную плату, доступны как стандартные устройства ввода-вывода. Они нужны потому, что мы живем в реальном аналоговом физическом мире и большинство физических датчиков генерируют электрические сигналы только в аналоговой форме. Чтобы цифровая ЭВМ могла обрабатывать эти сигналы, в системе в качестве входного порта требуется аналогово-цифровой преобразователь. И наоборот, чтобы выводить результаты обработки данных в реальный мир, нужен цифро-аналоговый преобразователь. Очевидным примером может служить осуществление в вычислительной системе ввода-вывода речи; здесь в составе терминала ввода-вывода необходимы АЦП и ЦАП. В настоящее время все более широкое распространение речевых терминалов является началом революционного преобразования промышленности обработки информации.

СВЯЗЬ МЕЖДУ ЦП И УСТРОЙСТВАМИ ВВОДА-ВЫВОДА

Чтобы все вышеприведенные устройства ввода-вывода могли работать в согласии с ЦП, должна быть обеспечена их аппаратная и программная поддержка. Поскольку каждому типу устройств ввода-вывода присущи свои электрические характеристики, для подключения этих устройств к ЦП может потребоваться основательное знание аппартных особенностей. Однако для устройств ввода-вывода общего назначения, описанных в предыдущем разделе, существуют стандартные интерфейсные печатные платы с соответствующими разъемами. Например, для пользователей системы PDP-11 доступен целый набор интерфейсных плат. Как только осуществлено правильное подключение этих устройств, можно приступать к разработке программного обеспечения ввода-вывода, которое будет выполнять транспортировку данных в систему и из нее.

Для транспортировки данных при вводе-выводе применяют два основных формата: последовательный и параллельный. Например, для терминала с клавиатурой и ЭЛТ используется последовательный формат. Поскольку большинство вычислительных систем имеет шинную структуру и пересылка данных к ЦП и от него в них производится в параллельном формате, то интерфейсная плата для последовательного формата должна содержать цепи для преобразования параллельного формата в последовательный и последовательного в параллельный. Хотя этот процесс происходит без участия программиста, важно, чтобы он о нем знал, поскольку для сбора последовательных данных и преобразования их в параллельный формат для ЦП и для преобразования параллельных данных от ЦП в последовательный формат для устройства ввода-вывода требуется определенное время.

В качестве типичного примера рассмотрим последовательную пересылку данных в коде ASCII от терминала. Хотя под один символ в коде ASCII нужно только семь бит, для последовательного формата обычно используются 11 бит: 7 бит для кода ASCII, один бит четности для обнаружения ошибок и три кадровых бита (один сигнализирует начало, а два или полтора — конец входящего кадра данных). На рис. 7.1 в последовательном формате показаны соответственно общий передающий сигнал и типичный символ А. В настоящее время большинство вычислительных систем располагает так называемым портом RS-232-C, готовым для подключения совместного с этим портом терминала с клавиатурой и ЭЛТ. В линии связи, работающей по стандарту RS-232-C, используется последовательный формат данных. Преобразование последовательного формата данных в параллельный и запись данных в регистр для передачи их ЦП через системную шину данных обычно выполняют схемы в интерфейсной печатной плате ввода-вывода. Эти схемы осуществляют также преобразование параллельных данных от ЦП в последовательный формат для пересылки на терминал с ЭЛТ.

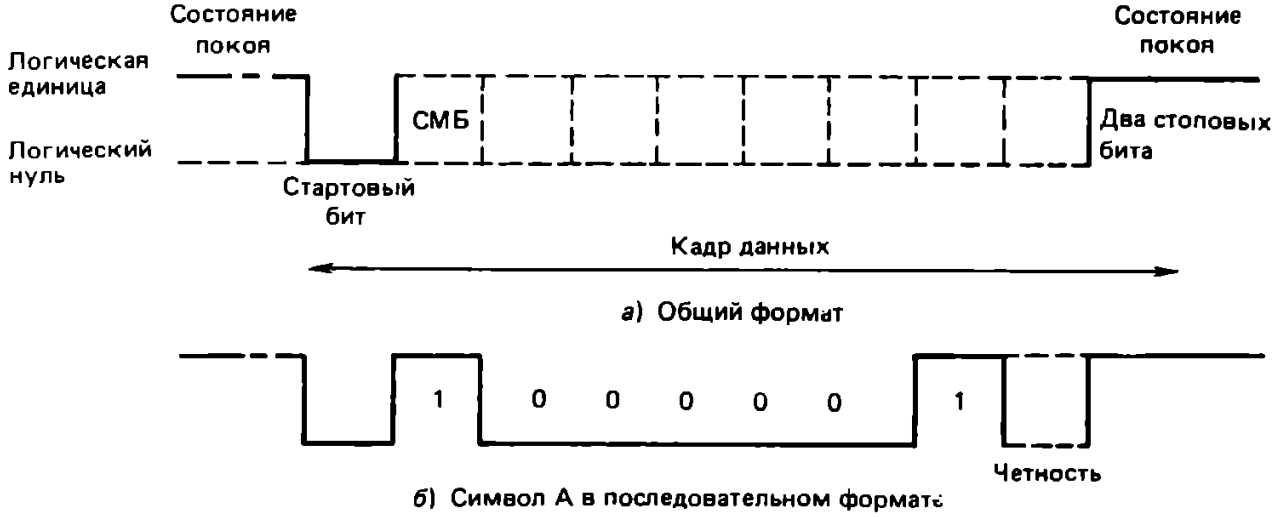


Рис. 7.1. Передача информации по линии связи в последовательном формате

Недостатком последовательной передачи данных является пониженная скорость передачи. Поскольку во многих приложениях требуется большая скорость передачи, в этих приложениях вместо последовательного формата применяется параллельный. В системе с параллельным форматом все биты данных бывают доступны в один и тот же момент времени, а схем, выполняющих преобразование последовательного формата в параллельный и параллельного в последовательный, не требуется. Очевидно, однако, что при параллельном формате для обеспечения параллельной передачи нужно намного больше линий связи, чем при последовательном формате, а это бывает неэкономично при большой длине линий связи.

Важно отметить тот факт, что большинство устройств ввода-вывода содержит электромеханические компоненты. В результате этого скорость операций варьируется от устройства к устройству. Кроме того, очевидно, что скорость работы электромеханического устройства ввода-вывода намного меньше, чем скорость работы ЦП. По этой причине связь между ЦП и устройством ввода-вывода является асинхронной. То есть, всякий раз, когда элемент данных, например символ в коде ASCII, должен быть передан, требуется "установление контакта" или так называемый процесс "рукопожатия". Он сводится к тому, что, прежде чем взять данные, ЦП должен проверять их наличие в регистре данных, а при выводе данных ЦП должен проверять, переданы ли уже предыдущие данные из регистра. Таким образом, на программисте лежит ответственность за то, чтобы разработанная программа ввода-вывода выполняла задачу транспортировки данных асинхронно в два шага, т. е. выполняла "рукопожатие" и пересылку данных.

7.2. ПРОГРАММИРОВАНИЕ ВВОДА-ВЫВОДА ДЛЯ СИСТЕМЫ PDP-11

В предыдущем разделе мы описали популярные устройства ввода-вывода и основные принципы осуществления связи между ЦП и устройствами ввода-вывода. В этом разделе мы опишем программирование ввода-вывода для системы PDP-11. В первую очередь познакомимся с общей структурой типичной интерфейсной платы ввода-вывода с точки зрения программиста.

ОБЩАЯ СТРУКТУРА ИНТЕРФЕЙСНОЙ ПЛАТЫ ВВОДА-ВЫВОДА

На интерфейсной плате ввода-вывода настолько много типов микросхем, резисторов и конденсаторов, что у программиста невольно возникает вопрос, как же все это можно запрограммировать? К счастью, нам не нужно знать подробности устройства аппаратуры. Для разработки необходимого программного обеспечения достаточно понимания основных структурных принципов. На рис. 7.2 показана система PDP-11 с шинной структурой. Для системы PDP-11 есть два типа шинных структур: общая шина и шина Q. Основное различие между ними состоит в том, что у первой структуры имеются раздельные линии адресов и данных, а у второй шины данных и адресов разделяют (совместно используют) одни и те же физические линии (провода). Однако с концептуальной точки зрения обе они структурированы таким образом, что имеют три группы проводов: адресную шину (шину A), шину данных (шину D) и шину управления (шину C), как это показано на рисунке.

К счастью, с точки зрения программиста, интерфейсная плата ввода-вывода может считаться не чем иным, как группой из двух или нескольких регистров. Для простого терминала с клавиатурой и ЭЛТ (как показано на рисунке) нужны четыре регистра на интерфейсной плате – два для операции ввода и два для операции вывода. Таким образом, связь интерфейсной платы с ЦП осуществляется программным обеспечением, а связь ее с устройством ввода-вывода – электронной схемой. Вспоминаем, что связываться с регистром мы можем так же, как и с ячейкой, содержащей слово памяти. Поэтому все, что необходимо для того, чтобы ЦП мог связываться с четырьмя регистрами интерфейсной платы как со словами памяти, – это назначить им соответственно

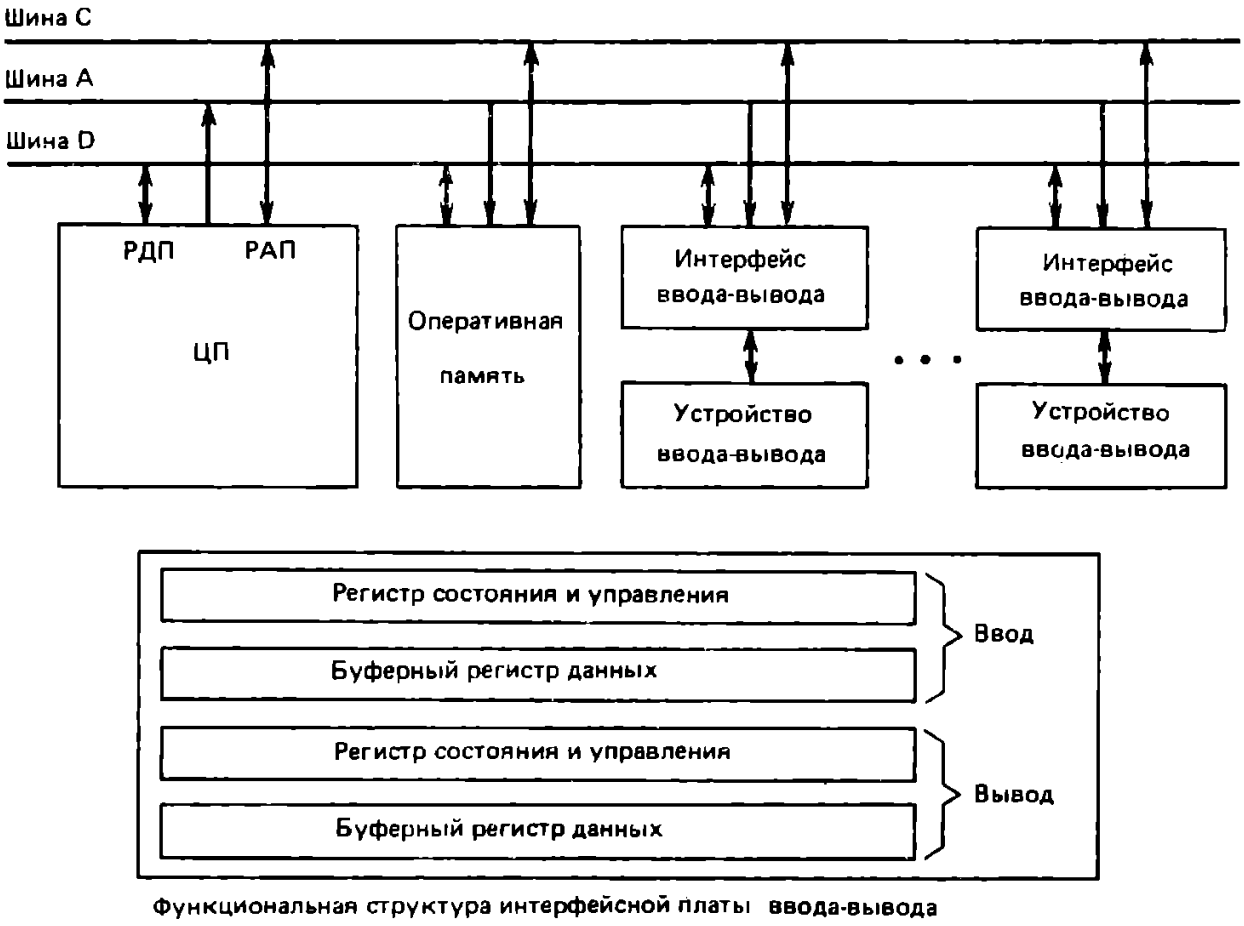


Рис. 7.2. Шинная структура системы PDP-11

четыре уникальных адреса памяти. Это называют **методом отображения адресации ввода-вывода на память.** Мы должны избегать назначения тех же адресов оперативной памяти.

Рассмотрим теперь регистры более подробно. Как показано на рис. 7.2 (в рамке), типичная плата ввода-вывода имеет четыре регистра: два для ввода и два для вывода. Как для ввода, так и для вывода один регистр предназначается для управления и состояния, а другой служит буферным регистром. Поэтому они и называются соответственно регистром управления и состояния и буферным регистром данных.

Общий формат регистров ввода-вывода показан на рис. 7.3. Конкретное определение каждого бита регистра от устройства к устройству может варьировать, но общий формат все-таки сохраняется. Как показано на рис. 7.3, буферный регистр данных имеет ширину 16 бит, так что в нем можно хранить любые данные размером 16 бит или меньше. В настоящее время большинство устройств ввода-вывода имеет 8-битовый формат, поэтому старший байт регистра используется не так уж часто. Регистр управления и состояния не так прост. В общем случае биты управления отображают те команды, которые выдает ЦП, а статусные биты показывают состояние устройства ввода-вывода. Например, на рисунке мы определили b_6 и b_0 как управляющие биты. Когда $b_6 = 1$, это подразумевает, что от системы требуется, чтобы она приняла сигнал запроса прерывания. Когда $b_0 = 1$, это означает, что ЦП разрешает работу того устройства, которое подсоединено к этому биту. Это может быть реле, звонок, красный фонарь и т. п. Кроме того, у нас есть три подмножества, управляемых ЦП: функциональные биты b_1, b_2, b_3; биты расширения памяти, b_4, b_5; биты выбора устройства ввода-вывода, b_8, b_9, b_{10}. Для некоторых устройств ввода-вывода их функции могут быть заданы ЦП с помощью бит b_1, b_2 и b_3.

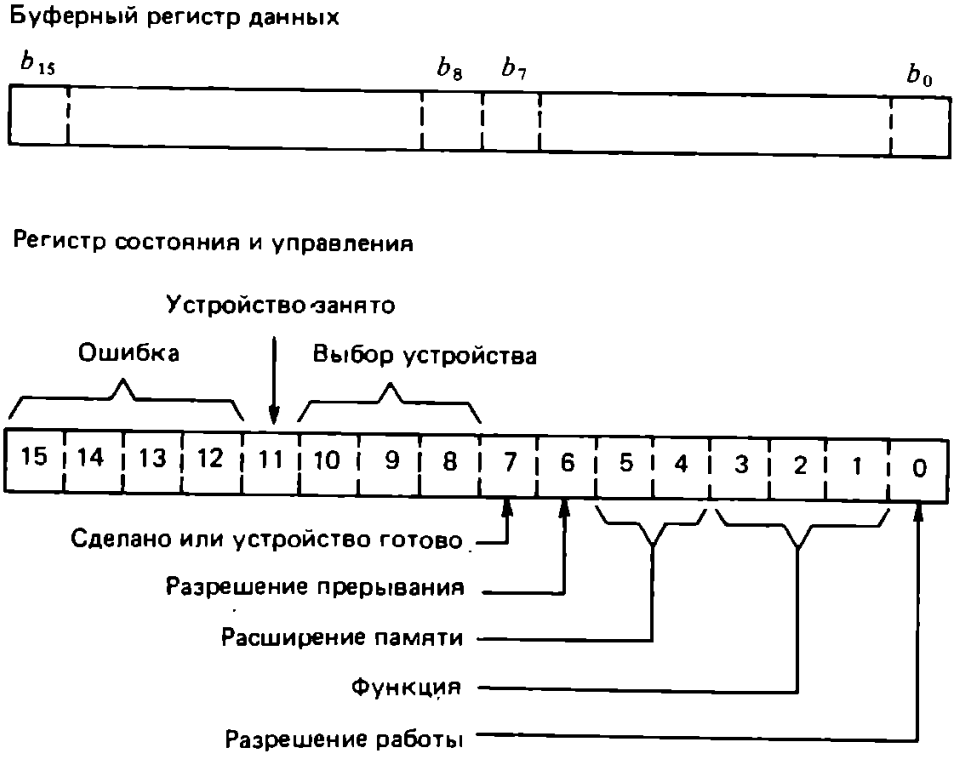


Рис. 7.3. Общий формат регистров ввода-вывода

Биты расширения памяти могут использоваться для увеличения размера памяти, а биты выбора устройства ввода-вывода могут определять выбор устройства в тех случаях, когда несколько устройств разделяют один и тот же регистр управления и состояния. Биты b_7, b_{11} и $b_{12}, \ldots, b_{15}$ используются как биты состояния. Эти биты показывают текущее состояние устройства ввода-вывода. Например, если $b_{11} = 1$, то это — указание ЦП, что устройство в данное время "занято" и ЦП не должен на него воздействовать. Состояние буферного регистра данных характеризует бит b_7. Для устройства ввода, если $b_7 = 1$, то буферный регистр заполнен данными, которые могут быть перенесены в ЦП; для устройства вывода это означает, что выходной буфер данных в настоящий момент пуст и ждет, когда ЦП загрузит в него новые данные. Биты ошибок указывают центральному процессору, что произошла ошибка и необходимо, чтобы ЦП на нее отреагировал.

ПРОГРАММИРОВАНИЕ ВВОДА-ВЫВОДА

В этом разделе мы рассмотрим несколько устройств ввода-вывода и интерфейсные платы. Мы покажем, как можно писать программы, специально предназначенные для выполнения ввода-вывода с помощью соответствующей аппаратуры.

Пример 1. Программирование ввода-вывода для телетайпа. Телетайп фактически содержит четыре устройства — клавиатуру, печатающее устройство, перфоратор и считыватель бумажной перфоленты. Клавиатура и считыватель перфоленты — это устройства ввода, обычно разделяющие (т. е. совместно использующие) одни и те же регистр управления и состояния и буферный регистр. Печатающее устройство и перфоратор — это устройства вывода, разделяющие другую пару регистров. Символическое соединение устройств и регистров ввода-вывода показано на рис. 7.4. В телетайпе используется последовательный формат передачи данных, а процесс ввода-вывода выполняется асинхронно. Регистры ввода-вывода определяются следующим образом.

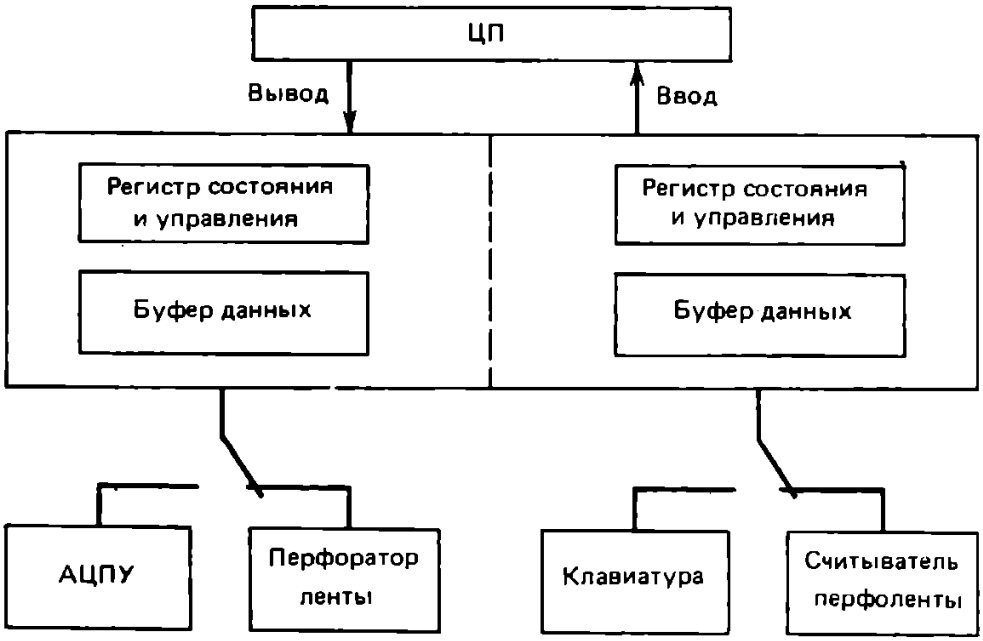


Рис. 7.4. Связь между телетайпом и ЦП

Регистры ввода

Буферный регистр данных ≙ ТКВ – символьный адрес буферного регистра клавиатуры или считывателя:

$b_0, \ldots, b_7$ данные
$b_8, \ldots, b_{15}$ не используются

Буфер управления и состояния ≙ TKS – символьный адрес регистра управления и состояния клавиатуры или считывателя:

b_0 разрешение чтения; устанавливается ЦП по указанию программного обеспечения, требующего чтения одного символа; очищается аппартно стартовым битом сигнала последовательного ввода.

b_7 сделано; устанавливается аппаратно, когда символ преобразован в параллельный формат и стал доступным в буферном регистре данных в ожидании, когда он будет передан в ЦП.

b_{11} занят; устанавливается битом разрешения чтения в то время, когда происходит последовательная загрузка символьного кода (ASCII) в буферный регистр; очищается битом b_7.

Другие биты не используются

Регистры вывода

Буферный регистр данных ≙ TPB – символьный адрес буферного регистра печатающего устройства или перфоратора:

$b_0, \ldots, b_7$ данные
$b_8, \ldots, b_{15}$ не используются

Регистр управления и состояния ≙ TPS – символьный адрес регистра управления и состояния печатающего устройства или перфоратора:

b_0 разрешение печати или перфорации.

b_2 эксплуатационный бит; соединяет последовательный выход TPB с последовательным входом ТКВ с целью проверки нормального функционирования устройства.

b_7 готов; устанавливается аппартно регистром TPB и показывает, что TPB пуст и готов к приему новых данных от ЦП; очищается, когда в TPB загружаются новые данные.

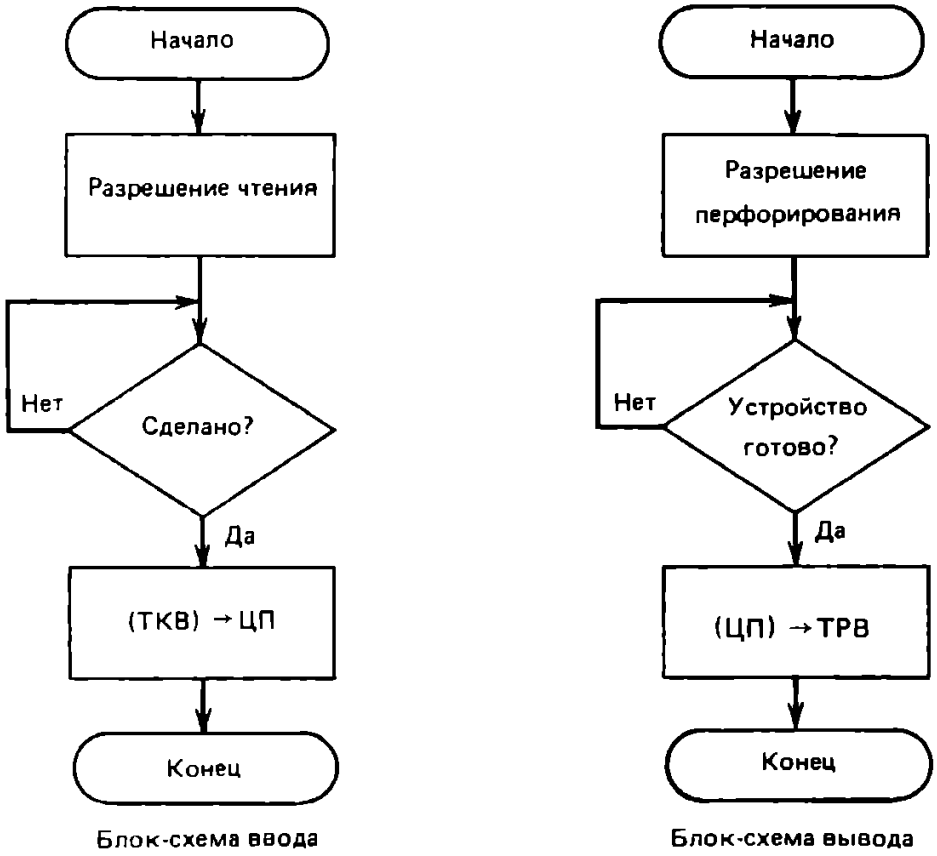


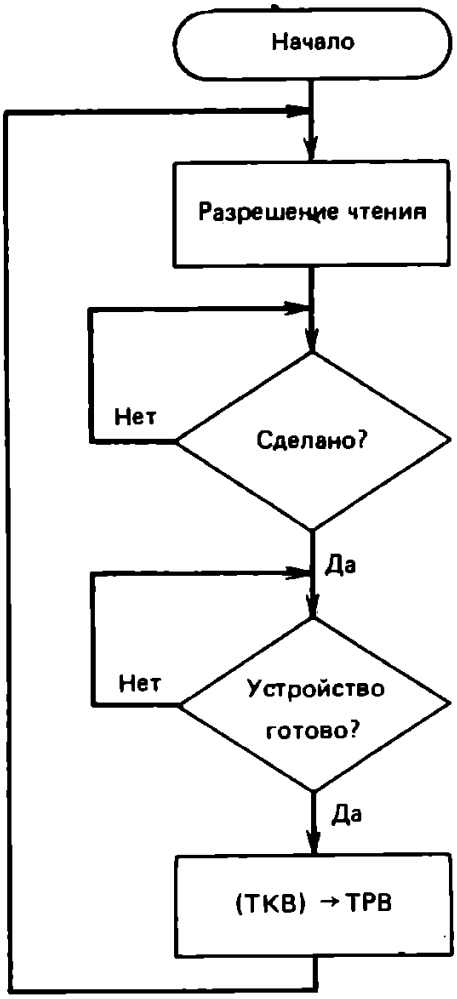
Рис. 7.5. Блок-схемы ввода-вывода для привода перфоленты

На рис. 7.5 показаны блок-схемы ввода-вывода для перфоленточных устройств. Соответствующая программа:

```
;СЧИТЫВАТЕЛЬ ПЕРФОЛЕНТЫ
TKS = 177560            ;НАЗНАЧЕНИЕ АДРЕСОВ РЕГИСТРАМ ВВОДА-ВЫВОДА:
TKB = 177562            ;ЭТО ПРИНЯТЫЕ В СИСТЕМЕ PDP-11
TPS = 177564            ;АДРЕСНЫЕ ЗНАЧЕНИЯ
TPB = 177566            ;ДЛЯ РЕГИСТРОВ ТЕЛЕТАЙПА
1) INPUT:    INC TKS          ;УСТАНОВИТЬ БИТ ГОТОВНОСТИ ЧТЕНИЯ,
                              ;ЧТОБЫ ПРОДВИНУТЬ ПЕРФОЛЕНТУ НА ОДИН
                              ;ШАГ И НАЧАТЬ ПОСЛЕДОВАТЕЛЬНУЮ
                              ;ПЕРЕСЫЛКУ ДАННЫХ В ТКВ
2) LOOP1:    TSTB TKS         ;ПРОВЕРИТЬ БИТ 7. СДЕЛАНО?
3)           BPL LOOP1        ;ЕСЛИ НЕТ, ВОЗВРАТ К ПРОВЕРКЕ
4)           MOV TKB,R0       ;ДАННЫЕ -> R0, Т.Е. В ЦП
5)           HALT
6)           .END INPUT
;ПЕРФОРАТОР
7) OUTPUT:   INC TPS          ;УСТАНОВИТЬ БИТ 0
8) LOOP2:    TSTB TPS         ;ПРОВЕРИТЬ БИТ 7. ГОТОВО?
9)           BPL LOOP2        ;ЕСЛИ НЕТ, ВОЗВРАТ К ПРОВЕРКЕ
10)          MOV R0,TPB       ;(R0) -> TPB
11)          HALT
12)          .END OUTPUT
```

Обратите внимание, что в строке 1 мы использовали инструкцию INC для установки в единицу нулевого бита регистра TKS. В результате такого действия перфолента продвигается на один шаг вперед в позицию, которая будет считываться. После этого аппаратура по кабелю последовательно пересылает данные в интерфейсную плату с добавлением стартового и стоповых бит. Соответствующие схемы на плате ввода-вывода проверяют четность, обнаруживают стартовый и стоповые биты, загружают данные в

Рис. 7.6. Блок-схема для непрерывного чтения перфоленты с выдачей эха



регистр ТКВ и устанавливают в единицу бит b_7 в регистре TKS. В строке 2 инструкция TSTB проверяет младший байт регистра TKS и устанавливает в слове состояния процессора PSW знаковый бит, если $b_7 = 1$ в регистре TKS; в противном случае знаковый бит будет нулевым. Это простой способ проверки бита b_7. В строке 3 инструкция BPL управляет дальнейшим ходом выполнения программы: если результат проверки в строке 2 отрицательный, это означает что бит $b_7 = 1$ и данные в ТКВ готовы для пересылки в ЦП, что и выполняется строкой 4; в противном случае строка 2 выполняется вновь и вновь. Строки 7—12 в программе вывода выполняют аналогичные действия, но в обратном порядке. В конце концов устройство ввода-вывода перфорирует ленту в соответствии с данными в регистре R0.

Пример 2. Программа ввода-вывода для чтения перфоленты и выдачи эха, того, что было считано, на печатающее устройство.

Блок-схема этой программы показана на рис. 7.6. Программа:

```
        TKS = 177560
        TKB = 177562
        TPS = 177564
        TPB = 177566
ECHO:   INC     TKS
LOOP1:  TSTB    TKS
        BPL     LOOP1
LOOP2:  TSTB    TPS
        BPL     LOOP2
        MOVB    TKB,TPB
        BR      ECHO
```

Обратите внимание, здесь предполагается, что "символический переключатель" выходного устройства на рис. 7.4 установлен в положение вывода на печатающее устройство. Эта программа, однако, никогда не остановится, даже если не будет перфоленты, подлежащей чтению. В качестве упражнения модифицируйте эту программу так, чтобы она останавливалась, когда кончится лента.

Пример 3. Подпрограмма для быстрого считывателя перфоленты с возможностью проверки ошибок. Телетайп, с которым мы имели дело в примерах 1 и 2, считается медленным устройством (10 символов в 1с); он может не требовать проверки ошибок в нашей программе. Однако для быстрого (например, 300 символов в 1с) перфоратора и считывателя перфоленты может понадобиться проверка ошибок. Пусть бит b_{15} регистра TRS (регистра управления и состояния перфосчитывателя) будет индикатором ошибки.

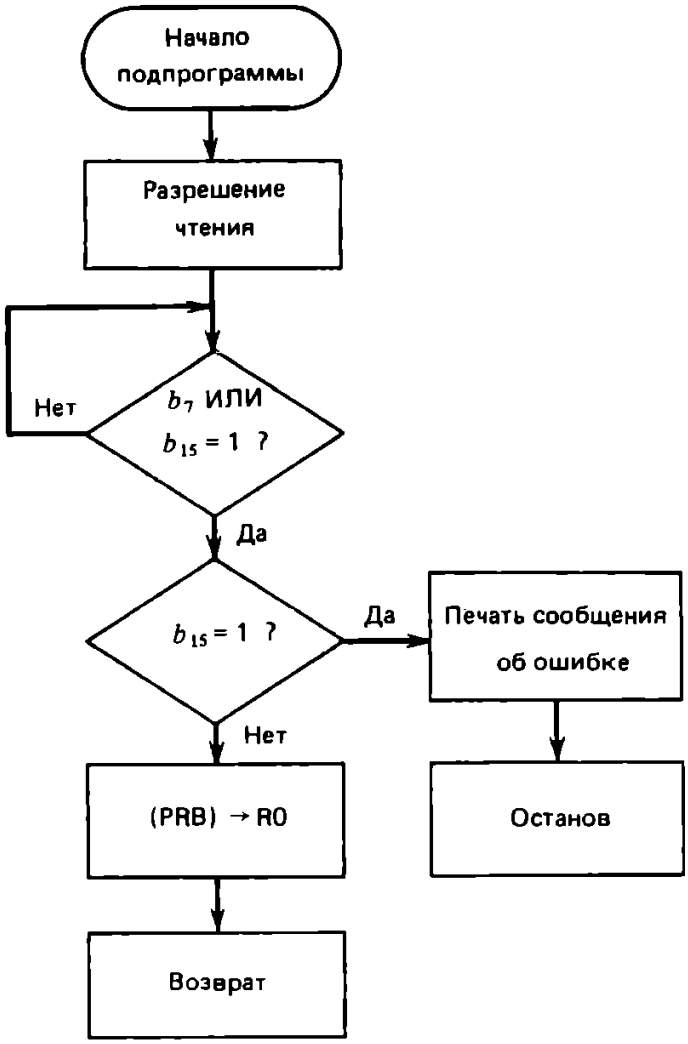


Рис. 7.7. Подпрограмма для высокоскоростного считывателя перфоленты

На рис. 7.7 показана блок-схема подпрограммы, а ее исходный текст следующий:

```
        PRS = 177550
        PRB = 177552
1) READ: INCB PRS               ;PRS - СИМВОЛЬНЫЙ АДРЕС РЕГИСТРА
                                ;СОСТОЯНИЯ ПЕРФОСЧИТЫВАТЕЛЯ
2) TEST: BIT #100200,PRS        ;ПРОВЕРКА БИТОВ 15 И 7 В PRS
3)       BEQ TEST               ;ВЕТВЛЕНИЕ К TEST, ЕСЛИ ОБА НУЛЕВЫЕ
4)       BMI ERROR              ;ЕСЛИ B15 = 1, ВЕТВЛЕНИЕ К ERROR
5)       MOVB PRB,R0            ;ЕСЛИ B15 = 0, B7 ДОЛЖНО БЫТЬ 1.
                                ;ПЕРЕСЛАТЬ ДАННЫЕ
6)       RTS PC
7) ERROR: (ВЫВОД СООБЩЕНИЯ)
8)       HALT
```

Обратите внимание, что в этой программе нет ничего нового, за исключением строк 2, 3 и 4. В строке 2, поскольку мы хотим проверить биты b_{15} и b_7 в регистре PRS, используется маска 100200, данная в восьмеричном представлении. В двоичном представлении эта маска имеет вид 1 000 000 010 000 000. То есть будут замаскированы все биты, кроме b_7 и b_{15}. Следовательно, программа продвинется дальше, если $b_7 = 1$ или $b_{15} = 1$, или и $b_7 = 1$, и $b_{15} = 1$. Строка 4 предназначена для выполнения проверки того, не установлен ли бит b_{15}. Если он установлен, то будет установлен и знаковый флаг в PSW, а это означает, что проверяемые данные имеют отрицательное значение или что $b_{15} = 1$; в противном случае в этом месте $b_7 = 1$ должен быть равен 1. В результате проверки

выполняется строка 5 программы. Здесь может возникнуть вопрос, а кто же устанавливает бит b_{15}, если случается ошибка. Обычно бит b_{15} устанавливается аппаратно схемой на плате ввода-вывода.

Пример 4. Отображение сообщения на ЭЛТ или печатающем устройстве при использовании директивы, задающей коды ASCII.

Для реализации этой программы давайте напишем следующие макросы:

```
.MACRO   CHAROUT R,?L1
L1:      TSTB    177564          ;177564 - ЭТО АДРЕС РЕГИСТРА
                                 ;УПРАВЛЕНИЯ И СОСТОЯНИЯ
         BPL     L1              ;ЕСЛИ НЕ СДЕЛАНО, ПЕРЕЙТИ К L1
         MOVB    R,177566        ;177566 - ЭТО АДРЕС ВЫХОДНОГО
                                 ;БУФЕРНОГО РЕГИСТРА ДАННЫХ
.ENDM    CHAROUT
.MACRO   DISPLAY INFO,?L2
         MOV     #INFO,R5        ;R5 - УКАЗАТЕЛЬ ДАННЫХ
L2:      CHAROUT (R5)+           ;ВЛОЖЕНИЕ МАКРОСА
         TSTB    (R5)            ;ПРОВЕРИТЬ, НЕ НУЛЕВЫЕ ЛИ ДАННЫЕ,
                                 ;НА КОТОРЫЕ УКАЗЫВАЕТ R5
         BNE     L2              ;ЕСЛИ НЕТ, СООБЩЕНИЕ ЕЩЕ НЕ КОНЧИЛОСЬ
.ENDM    DISPLAY
;ОСНОВНАЯ ПРОГРАММА
START:   DISPLAY MESSAGE
         HALT
MESSAGE: .ASCIZ /HI!/
         .EVEN
         .END    START
```

В этой программе два макроса: один – для однократной записи или вывода одного символа, другой – для отображения сообщений. Обратите внимание, что директивы .ASCIZ и .EVEN являются для нас новыми. Первая из них между двумя косыми чертами представляет то сообщение, которое мы хотим отобразить. У нас есть четыре символа: H, I, пробел, !, расположенные в четырех последовательных байтах памяти. Ассемблер, выполняя эту директиву, автоматически добавляет к сообщению нулевой байт:

Символьный адрес	Содержимое	
	Старший байт	Младший байт
MESSAGE	1 1 1 (I)	1 1 0 (H)
	0 4 1 (!)	0 4 0 (пробел)
	0 0 0	0 0 0

Директива .EVEN требует, чтобы ассемблер проверил, занимает полное сообщение четное число байт или нет, с тем, чтобы если это необходимо, добавить к нему один нулевой байт для четности числа байт. Это важно, ибо в противном случае следующая инструкция окажется расчлененной на два байта и одна половина инструкции будет в старшем байте одного слова памяти, а другая – в младшем байте следующего слова.

Возвращаясь к макросу для отображения сообщений, замечаем, что регистр R5 используется как указатель данных. Первоначально он указывает на символ H, в процессе работы автоматически увеличивается каждый раз на единицу, указывая на следующий байт. Инструкция BNE проверяет, не встретился ли нулевой байт. Если да, то это конец сообщения и надо остановить работу макроса отображения сообщений. В системе PDP-11 есть другая, аналогичная директива – .ASCII, но она не делает автоматического добавления к сообщению нулевого байта. Поэтому программист должен каким-то

```
 1                                      ;ПРИМЕР ОТОБРАЖЕНИЯ СООБЩЕНИЯ С ПОМОЩЬЮ ДИРЕКТИВЫ .ASCIZ
 2                                      ;
 3                                      ;
 4                                              .MACRO  CHAROUT R,?L1
 5                                      L1:     TSTB    177564
 6                                              BPL     L1
 7                                              MOVB    R,177566
 8                                              .ENDM   CHAROUT
 9                                      ;
10                                              .MACRO  DISPLAY INFO,?L2
11                                              MOV     #INFO,R5
12                                      L2:     CHAROUT (R5)+
13                                              TSTB    (R5)
14                                              BNE     L2
15                                              .ENDM   DISPLAY
16                                      ;
17                                              .MACRO  NEWLINE
18                                              CHAROUT #12     ;ВЫВОД КОДА ASCII ПЕРЕВОДА СТРОКИ
19                                              CHAROUT #15     ;ВЫВОД КОДА ASCII ВОЗВРАТА КАРЕТКИ
20                                              .ENDM   NEWLINE
21                                      ;
22                                      ;ОСНОВНАЯ ПРОГРАММА
23 000000                               START:  DISPLAY MESS1
24 000022                                       DISPLAY MESS2
25 000044                                       NEWLINE
26 000074                                       DISPLAY MESS3
27 000116  000000                               HALT
28 000120    110     111     040        MESS1:  .ASCIZ  /HI !/          ;МЕЖДУ "I" И "!" ЕСТЬ ПРОБЕЛ
   000123    041     000
29 000125    040     144     157        MESS2: .ASCIZ   / ДОБРОЕ УТРО/
   000130    162     162     157
   000133    145     040     165
   000136    164     162     157
   000141    000
30 000142    155     157     147        MESS3: .ASCIZ   /МОГУ Я ВАМ ПОМОЧЬ?/
   000145    165     040     161
   000150    040     167     141
   000153    155     040     160
   000156    157     155     157
   000161    176     170     077
   000164    000
31                                              .EVEN
32          000001                              .END    START

   РЕЗУЛЬТАТ ВЫПОЛНЕНИЯ:    HI ! ДОБРОЕ УТРО
                            МОГУ Я ВАМ ПОМОЧЬ?
```

Рис. 7.8. Пример использования директивы .ASCIZ для отображения информации

образом, например с помощью счетчика символов, определить конец отображения сообщения. На рис. 7.8 показан файл листинга программы, аналогичной программе данного примера, но с добавлением двух более "дружественных" сообщений.

7.3. БОЛЕЕ ПОДРОБНЫЕ ПРИМЕРЫ

В этом разделе мы приводим более практичные и завершенные примеры программ, чем до сих пор.

ИМИТАЦИЯ КОНТРОЛЛЕРА ЗАЩИЩЕННОГО ВХОДА В ЗДАНИЕ

В этом примере мы хотим показать, как ЭВМ PDP-11 может быть использована для имитации контроллера защищенного входа в здание. Структура системы приведена на рис. 7.9. Посетитель должен позвонить в звонок на двери, надавив на его кнопку. ЭВМ отвечает на это сообщением на ЭЛТ: ПОЖАЛУЙСТА, ВАШЕ ИМЯ. Посетитель вводит свое имя с помощью клавиатуры. Если введенное имя совпадает с именем, за-

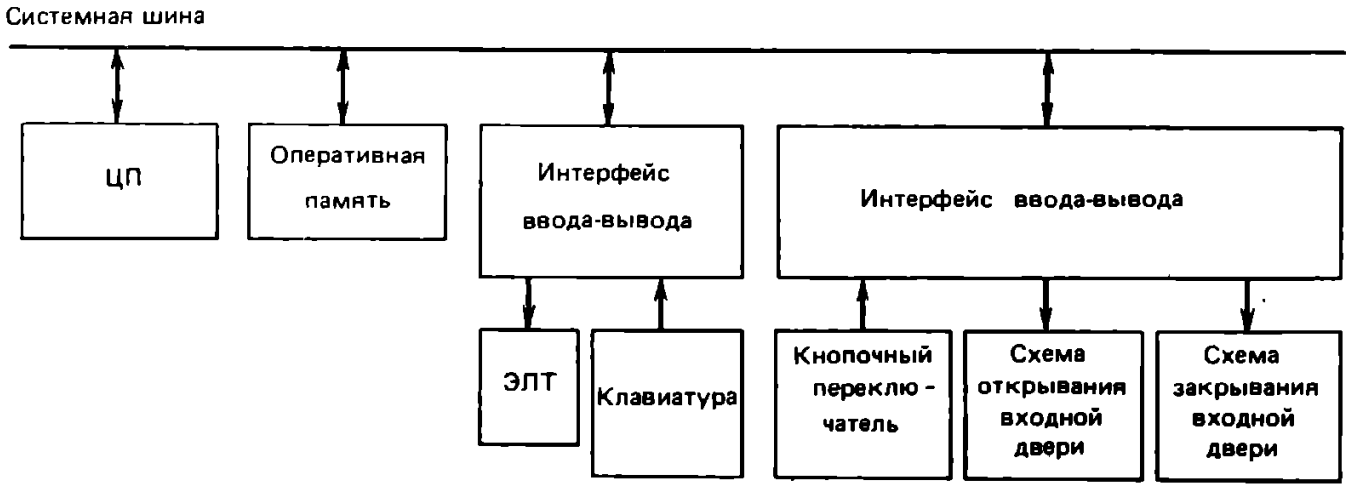


Рис. 7.9. Структура системы для контроллера защищенного входа в здание

ранее введенным в систему, ЭВМ запрашивает ПАРОЛЬ. Если пароль назван, входная дверь будет отперта, затем заперта вновь. В противном случае ЭВМ отобразит сообщение: ИЗВИНИТЕ, ВХОД ВОСПРЕЩЕН.

Для имитации в лабораторных условиях (когда нет аппаратуры для звонка и мотора для открывания и закрывания двери) для имитации кнопки звнока можно использовать, например, клавишу символа В на клавиатуре; для имитации работы мотора сообщения на ЭЛТ: ДВЕРЬ ОТКРЫВАЕТСЯ и ДВЕРЬ ЗАКРЫВАЕТСЯ. На рис. 7.10, *а* показана блок-схема программы, а на рис. 7.10, *б* – исходная программа, где для простоты L – имя, а P – пароль. Теперь читатель должен уметь модифицировать программу так, чтобы под имя и пароль вместо одиночного символа использовалась определенная группа символа.

ПРОГРАММИРОВАНИЕ ВВОДА-ВЫВОДА ДЛЯ АЦП И ЦАП

Во многих случаях полезна возможность обрабатывать на PDP-11 аналоговые сигналы и выводить полученные результаты в аналоговом виде. Чтобы это можно было сделать, мы должны установить в ЭВМ плату ввода-вывода со схемами **аналого-цифрового** и **цифро-аналогового преобразователей** (АЦП и ЦАП). К счастью, платы этого типа в настоящее время доступны за вполне приемлемую цену. Но программы ввода-вывода для приема оцифрованных данных от АЦП и пересылки обработанных цифровых данных в ЦАП с целью получения аналогового сигнала нам придется разработать самим.

На рис. 7.11 изображена функциональная блок-схема интерфейса аналого-цифрового и цифро-аналогового преобразователей. Хотя аппаратная структура и схемы преобразователей интерфейса заметно отличаются от тех периферийных устройств, которые были описаны в предыдущих разделах, программирование ввода-вывода и в данном случае выполняется в два шага: "рукопожатие" и транспортировка данных. Например, для получения данных через АЦП центральный процессор должен выдать сигнал "начало преобразования", чтобы инициировать процесс преобразования. Когда цифровой сигнал будет готов и загружен в буферный регистр данных, АЦП пошлет сигнал "конец преобразования", который проинформирует ЦП, что входные данные готовы к транспортировке. При выводе через ЦАП преобразователь должен сначала выставить сигнал "запрос новых данных" (чтобы проинформировать ЦП о своей готов-

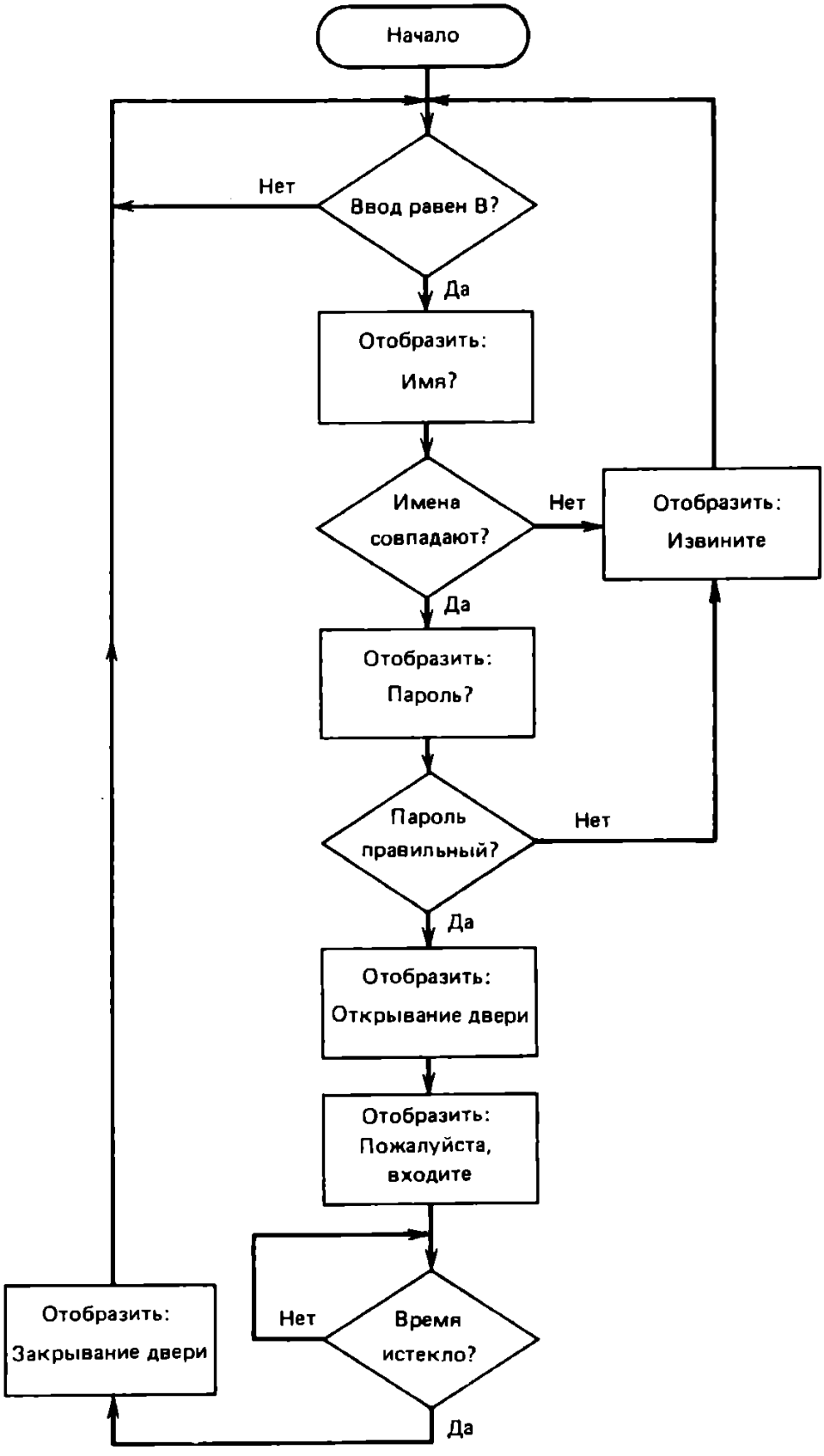


Рис. 7.10, *а*. Блок-схема для контроллера защищенного входа в здание

```
;ПРИМЕР ПРОГРАММЫ, КОТОРАЯ УПРАВЛЯЕТ ВХОДОМ В ЗДАНИЕ.
;ПРИ ВЫПОЛНЕНИИ ПРОГРАММА ОЖИДАЕТ, ЧТО ПОСЕТИТЕЛЬ
;НАЖМЕТ КЛАВИШУ "B", ЗАМЕНЯЮЩУЮ КНОПКУ ЗВОНКА.
;
.TITLE  КОНТРОЛЛЕР ВХОДА В ЗДАНИЕ
.MACRO  READ R,?L1
```

```
L1:     TSTB    177560
        BPL     L1
        MOVB    177562,R
.ENDM   READ
;
.MACRO  PRINT R,?L2
L2:     TSTB    177564
        BPL     L2
        MOVB    R,177566
.ENDM   PRINT
;
.MACRO  DISPLAY INFO,?L3
        MOV     #INFO,R5        ;R5 - УКАЗАТЕЛЬ ДАННЫХ
L3:     PRINT   (R5)+
        TSTB    (R5)            ;КОНЕЦ СООБЩЕНИЯ?
        BNE     L3
        PRINT   #12             ;ПЕРЕВОД СТРОКИ
        PRINT   #15             ;ВОЗВРАТ КАРЕТКИ
.ENDM   DISPLAY
;
;ОСНОВНАЯ ПРОГРАММА
START:  READ    TEMP            ;ПОЛУЧИТЬ ДАННЫЕ
        DISPLAY TEMP            ;ВЫДАТЬ ЭХО ДАННЫХ
        CMP     #102,TEMP       ;(TEMP) = B?
        BNE     START
        DISPLAY NAME
        READ    TEMP
        DISPLAY TEMP
        CMP     #114,TEMP       ;(TEMP) L?
        BNE     S
        DISPLAY PASSWD
        READ    TEMP
        DISPLAY TEMP
        CMP     #120,TEMP       ;(TEMP) P?
        BNE     S
        DISPLAY OPEN
        DISPLAY WELCOM
        CLR     R0
        CLR     R1
T:      INC     R0              ;ИМИТАЦИЯ ПОМЕЖУТКА ВРЕМЕНИ
        BNE     T               ;ПЕРЕД ЗАКРЫТИЕМ ДВЕРЕЙ
        INC     R1
        CMP     TLIMIT,R1
        BNE     T
        DISPLAY CLOSE
        BR      AGAIN
S:      DISPLAY SORRY
AGAIN:  JMP     START
TEMP:   .BLKB   1
        .EVEN
TLIMIT: .WORD   4
NAME:   .ASCIZ  /ВВЕДИТЕ, ПОЖАЛУЙСТА, СВОЕ ИМЯ/
PASSWD: .ASCIZ  /ВВЕДИТЕ, ПОЖАЛУЙСТА, ПАРОЛЬ/
SORRY:  .ASCIZ  /ИЗВИНИТЕ, ВХОД ВОСПРЕЩЕН/
OPEN:   .ASCIZ  /ДВЕРИ ОТКРЫВАЮТСЯ/
WELCOM: .ASCIZ  /ПОЖАЛУЙСТА, ВХОДИТЕ/
CLOSE:  .ASCIZ  /ДВЕРИ ЗАКРЫВАЮТСЯ/
        .EVEN
        .END    START
```

Рис. 7.10, *б*. Исходная программа для контроллера защищенного входа в здание

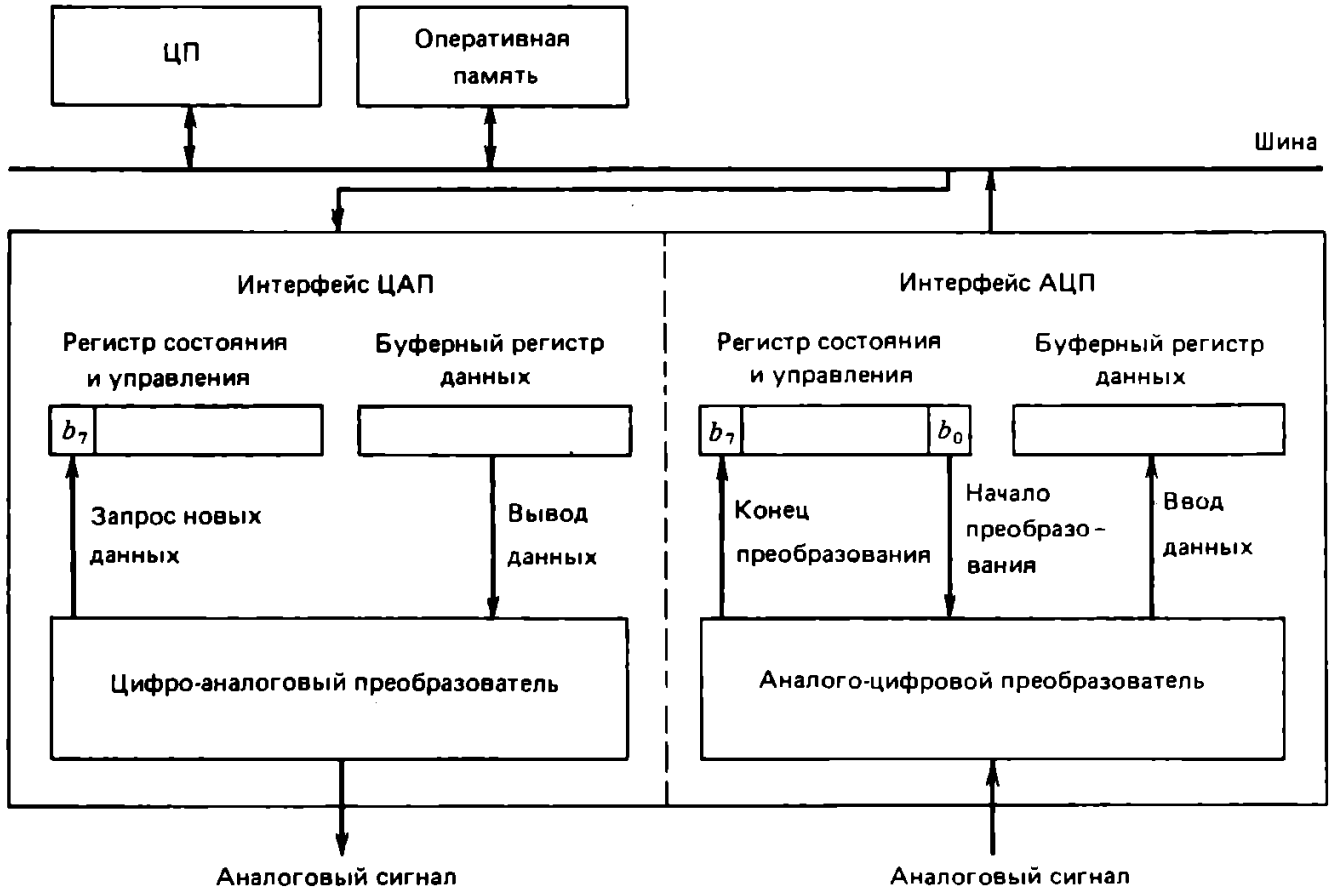


Рис. 7.11. Функциональная блок-схема интерфейса АЦП и ЦАП

ности принять от него новые цифровые данные), после чего ЦП перешлет данные в выходной буферный регистр данных. Соответствующая программа ввода-вывода:

```
;ПОДПРОГРАММА АЦП
ADC:    CLR     ADCSR           ;ОЧИСТИТЬ РУС
        INC     ADCSR           ;НАЧАТЬ ПРЕОБРАЗОВАНИЕ
L1:     TSTB    ADCSR           ;КОНЕЦ ПРЕОБРАЗОВАНИЯ? СМ. ЗАМЕЧ. 1
        BPL     L1              ;НЕТ, ВНОВЬ ПРОВЕРИТЬ БИТ ОКОНЧАНИЯ
        MOV     ADBUF,R0        ;ВЫВЕСТИ ДАННЫЕ В ЦП
        RTS     PC
;ЗАМЕЧАНИЕ: МЫ ПОЛАГАЕМ, ЧТО В КОНЦЕ ПРЕОБРАЗОВАНИЯ
;          БУДЕТ УСТАНОВЛЕН БИТ 7.
;ПОДПРОГРАММА ЦАП
DAC:    TSTB    DACSR           ;ЗАПРОС НОВЫХ ДАННЫХ?
        BPL     DAC             ;НЕТ, ВНОВЬ ПРОВЕРИТЬ БИТ ГОТОВНОСТИ
        MOV     R0,DABUF        ;ЦП ВЫВОДИТ ДАННЫЕ В БУФЕРНЫЙ РЕГИСТР
        RTS     PC
```

Здесь ADCSR $\triangleq$ регистр управления и состояния АЦП;
ADBUF $\triangleq$ буферный регистр данных АЦП;
DACSR $\triangleq$ регистр управления и состояния ЦАП;
DABUF $\triangleq$ буферный регистр данных ЦАП.

До сих пор мы описывали основные принципы работы интерфейса ввода-вывода для АЦП и ЦАП, упростив структуру интерфейса. На практике же плата АЦП и ЦАП может содержать канальный мультиплексор, позволяющий мультиплексировать до 64 линий аналоговых сигналов к одному преобразователю, в результате чего достигается разделение времени одного АЦП с хорошими характеристиками. Кроме того, на этой плате может также располагаться схема прерывания. Практический пример использования интерфейсной платы ввода-вывода для АЦП и ЦАП на ЭВМ PDP-11 мы приведем в следующей главе после того, как изучим прерывания.

7.4. УПРАЖНЕНИЯ

1. Перечислите существенную информацию, которая необходима вам для написания программы ввода-вывода.
2. Что такое процесс "рукопожатия" и почему он нужен при написании программ для асинхронного ввода-вывода?
3. Напишите подпрограмму вывода, которая будет отображать заданный центральным процессором какой-то конкретный визуальный образ на панели, представляющей собой матрицу из светодиодов размером 8 × 8. Вы можете использовать два выходных порта, регистры которых имеют следующие адреса:

   Порт 1: регистр управления и состояния – 177570;
   буферный регистр данных – 177572.
   Порт 2: регистр управления и состояния – 177574;
   буферный регистр данных – 177576.

   Вы, конечно, можете полагать, что аппаратура интерфейса уже разработана и способна функционировать.
4. Можете ли вы написать макрос вывода, который будет декодировать любой символ в коде ASCII, принимаемый от ЦП, и использовать подпрограмму, созданную вами для упр. 3, для отображения этого конкретного символа кода ASCII? Попробуйте это сделать.
5. Предположите, что к входному порту вашей ЭВМ PDP-11 подключена клавиатура, подобная клавиатуре пишущей машинке, которая может генерировать код ASCII. Адреса входных регистров:

   регистр управления и состояния – 177560;
   буферный регистр данных – 177562.

   Напишите программу ввода-вывода, которая будет выдавать эхо для вводимого знака на дисплейную панель, описанную в упр 3.
6. Напишите основную или главную программу для последовательного отображения символьной строки HI! на светодиодной панели, если эти символы вводятся с помощью клавиатуры, описанной в упр. 5. Чтобы выводимые символы можно было наблюдать, каждый из них должен отображаться по крайней мере в течение 1с, а все сообщение должно повторяться до тех пор, пока пользователь не нажмет на клавиатуре клавишу возврата каретки CR.
7. Модифицируйте программу из примера 2 (с. 155) так, чтобы процесс обработки останавливался при окончании ленты. Вы можете полагать, что при окончании ленты считыватель перфоленты будет постоянно считывать "все нули" в течение 10 шагов.
8. Модифицируйте программу контроллера защищенного входа в здание так, чтобы она позволила использовать для имени и пароля по три символа, а пользователь мог иметь список из 10 человек, которым разрешен вход в здание.

ГЛАВА 8
ПРЕРЫВАНИЯ И ЛОВУШКИ

8.1. ВВЕДЕНИЕ

Концепция прерывания – одно из величайших изобретений в области архитектуры ЭВМ, относящихся к устройствам ввода-вывода. До признания этой концепции в вычислительных системах обычно использовался метод **опроса**: когда ЦП "по кругу" проверяет каждый порт ввода-вывода, чтобы "увидеть", не нуждается ли в обслуживании какое-либо из подключенных к нему периферийных устройств. Если да, то ЦП переходит к соответствующей подпрограмме и обслуживает это устройство; в противном случае он продолжает циклическую проверку устройств. Этот процесс напоминает действия торговца, который ходит из квартиры в квартиру, опрашивая потенциальных клиентов и надеясь, что кто-нибудь заинтересуется и купит предлагаемые им товары.

Недостатком метода опроса является неэффективное использование процессорного времени, поскольку, осуществляя опрос, ЦП не делает ничего другого. Процессорное время мы, в частности, теряем тогда, когда большая часть устройств ввода-вывода не требует частого обслуживания. Метод **прерываний** заключается в том, что обслуживание выполняется по запросам. Это похоже на работу посылторга: если клиент хочет что-то купить, он посылает запрос по почте. В системе, приводимой в действие прерываниями, ЦП может заниматься чем-нибудь другим до тех пор, пока устройством ввода-вывода не будет инициирован запрос на обслуживание (или запрос прерывания). Для ЦП запрос прерывания является неожиданным событием, поскольку он может появиться в любое время, что порождает неопределенность в системе, однако этот метод дает возможность эффективно использовать процессорное время.

8.2. ПРИНЦИП РАБОТЫ

Прерывание – это процесс обработки неожиданных событий, происходящих внутри или вне вычислительной системы. Например, внутреннее прерывание вызывается, если в системе происходит переполнение в АЛУ, сброс питания или выполнение неверной инструкции. Внешние прерывания относятся обычно к событиям, сигнализирующим о возникновении определенных условий, связанных с устройствами ввода-вывода. Давайте детально изучим процесс внешнего прерывания.

Для выполнения процесса внешнего прерывания, порожденного одним или несколькими устройствами ввода-вывода, должны быть предприняты следующие шаги:

1. Устройство ввода-вывода выставляет сигнал **запроса прерывания** к ЦП.
2. Проверяет приоритетный ранг прерывания.
3. Если устройство имеет более высокий ранг, чем текущая программа, то ЦП посылает сигнал **согласия на прерывание** (или "подтверждение прерывания"), с целью извлечь идентификационное число устройства.
4. ЦП проталкивает в стек текущее состояние программы и адрес для возврата к главной программе. Под главной программой понимается программа, выполняемая в настоящее время.
5. ЦП переходит к подпрограмме обслуживания этого устройства и выполняет ее.
6. ЦП возвращается к главной программе, выталкивая назад из стека состояние и адрес возврата соответственно в слово состояния PSW и регистр PC.

АППАРАТНАЯ СТРУКТУРА СИСТЕМЫ, ПРИВОДИМОЙ В ДЕЙСТВИЕ ПРЕРЫВАНИЯМИ

Для реализации процесса прерывания необходима специальная аппаратура, дополненная соответствующим программным обеспечением. Аппаратная структура системы, приводимой в действие прерываниями, показана на рис. 8.1. Здесь мы видим два новых для нас элемента: векторы прерываний в блоке оперативной памяти и регистр, сохраняющий информацию об адресе вектора прерываний на интерфейсной плате ввода-вывода. Первый элемент содержит информацию о стартовых адресах подпрограмм обслуживания прерываний и о приоритетных рангах устройств ввода-вывода; второй – указывает процессору, где он может найти информцию об обслуживающей подпрограмме и приоритетном ранге конкретного устройства, связанного с этой платой ввода-вывода. То есть последняя информация обеспечивает идентификационное число устройства. Этот тип системы прерывания известен как **векторная система прерывания**. В блоке векторов прерываний в оперативной памяти каждому порту (устройству) ввода-вывода принадлежат два последовательных слова памяти. В первом слове хранится адрес подпрограммы обслуживания прерываний, во втором слове (с большим адресом) – приоритетный ранг.

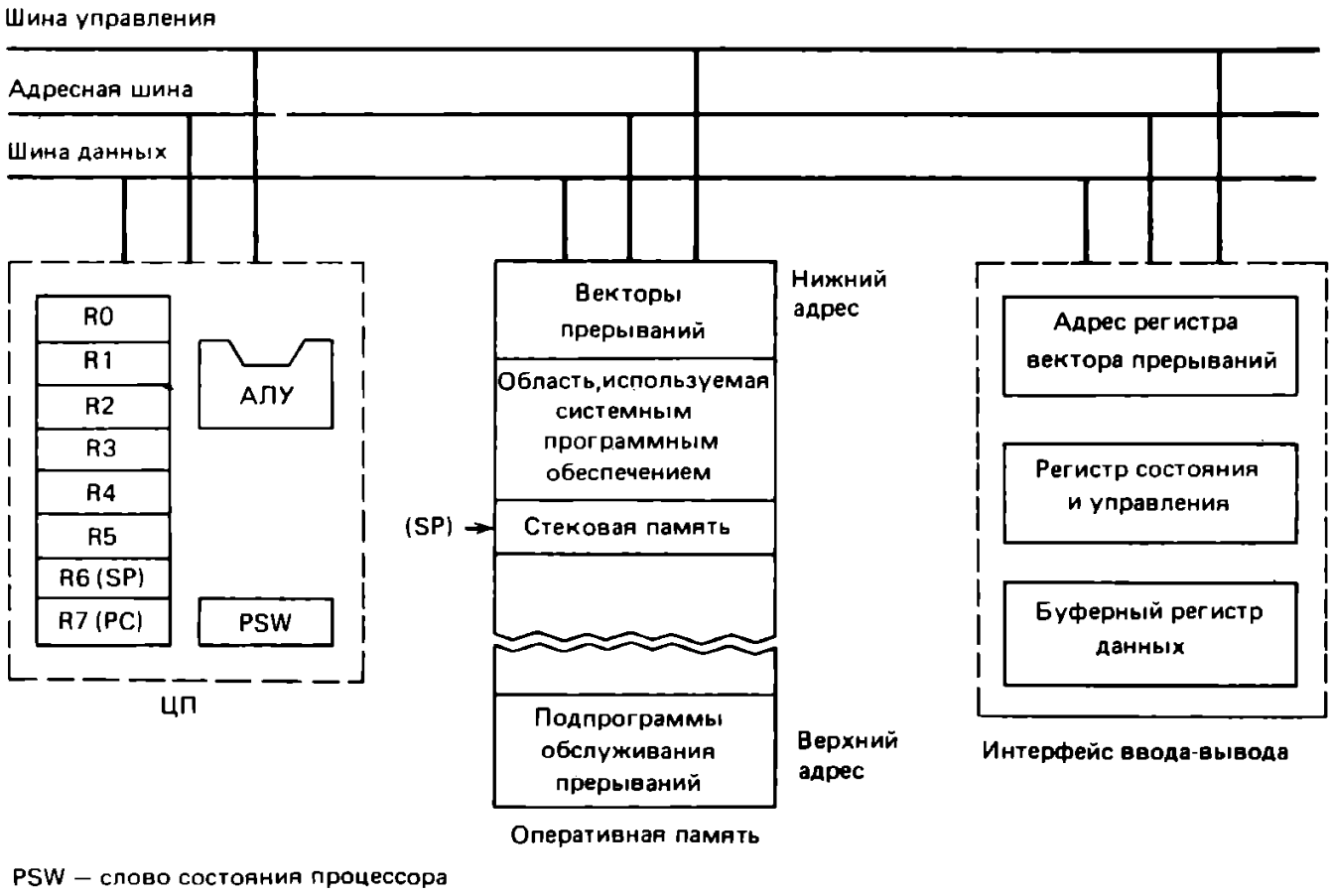


Рис. 8.1. Структура вычислительной системы, приводимой в действие прерываниями

ОБЩАЯ ПРОЦЕДУРА ПРОЦЕССА ПРЕРЫВАНИЯ

Предположим, что содержимое адреса, находящегося в регистре вектора прерываний на плате ввода-вывода, есть 60. Процесс прерывания начинается со следующих шагов:

1. Плата ввода-вывода посылает по шине управления сигнал запроса прерывания.
2. В предположении, что приоритетный ранг устройства ввод-вывода выше, чем у текущей программы, ЦП отвечает сигналом согласия на прерывание на шине управления, который стробирует перенос содержимого АДРЕСА из РЕГИСТРА ВЕКТОРА ПРЕРЫВАНИЙ на шину данных и затем в ЦП.
3. ЦП проталкивает в стек текущее (PSW) и скорректированное содержимое регистра PC.
4. ЦП извлекает вектор прерываний из ячейки 60 и загружает стартовый адрес подпрограммы обслуживания прерываний и приоритетный ранг соответственно в регистр PC и слово состояния PSW.
5. Как только ЦП "видит" инструкцию возврата из прерывания RTI, завершающую подпрограмму обслуживания прерываний, он выталкивает из стека скорректированное содержимое регистра PC и первоначальное содержимое слова состояния PSW назад в регистр PC и слово состояния PSW соответственно.
6. ЦП возобновляет первоначальный процесс, выполнявшийся до прерывания.

Шаги 3 и 5 показаны на рис. 8.2. Все указанные шаги реализуются аппаратно. Следовательно, программисту необходимо только инициализировать вектор прерываний и разработать соответствующую подпрограмму обслуживания прерываний. Блок-схема процесса прерывания представлена на рис. 8.3.

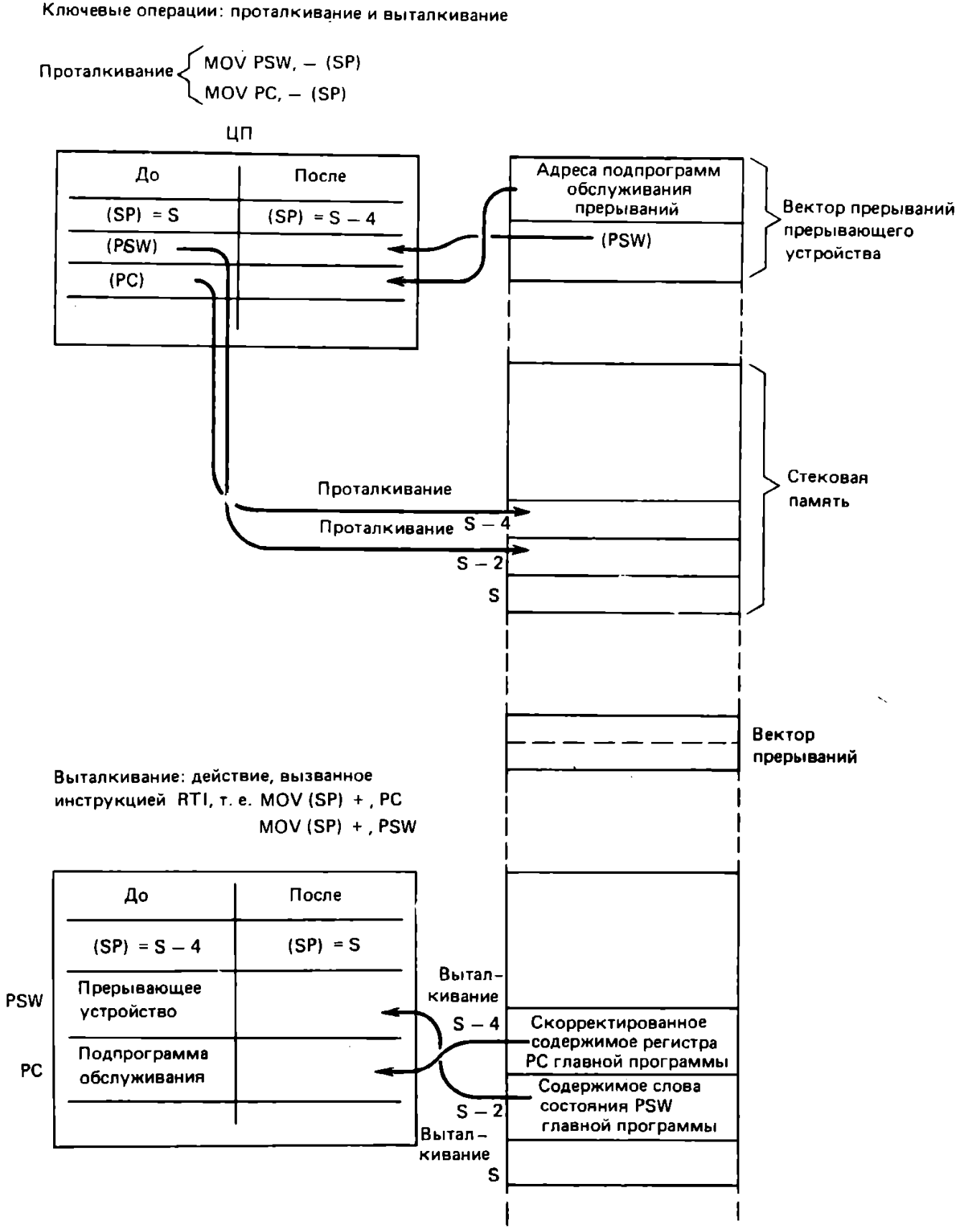


Рис. 8.2. Ключевые операции в течение процесса прерывания

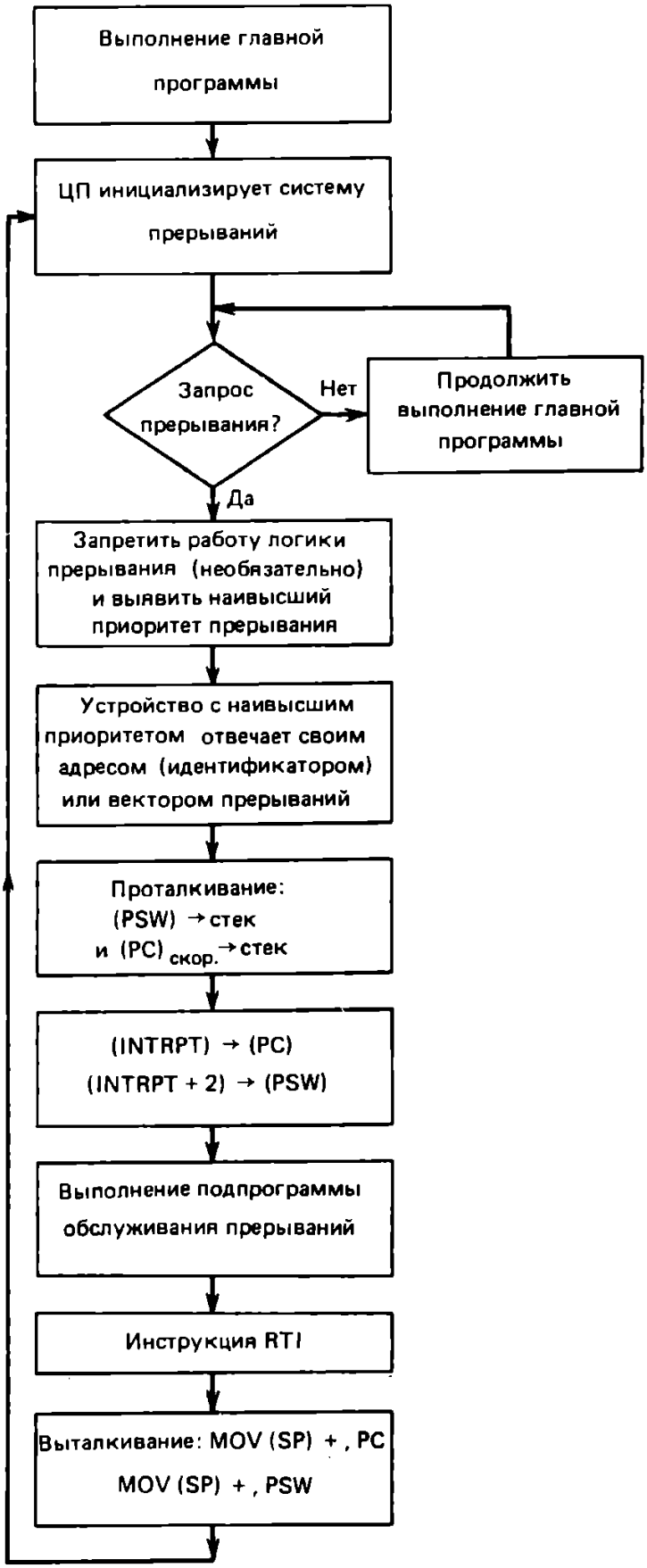


Рис. 8.3. Блок-схема процесса прерывания

ТИПИЧНЫЙ ПРИМЕР

В этом разделе для иллюстрации процесса прерывания мы приведем типичный пример на основе использования платы ввода-вывода DLV11-F (производимой фирмой Digital Equipment Corporation). Обратимся к рис. 8.1. Есть три ключевых регистра: 1) регистр адреса вектора прерываний, содержимое которого – идентификационное число платы или адрес ассоциированного с платой вектора прерываний (это только читаемый регистр, т. е. ЦП может только читать информацию из такого регистра, но не может записывать ее в этот регистр); 2) регистр управления и состояния (РСУ); 3) буферный регистр данных. Два последних являются регистрами чтения и записи (Ч/З), т. е. ЦП может читать информацию из такого регистра и записывать ее в этот регистр. На интерфейсной плате DLV11-F предусмотрены, однако, пять регистров:

1. *Регистр адреса вектора прерываний.* Он реализуется 8-битовым двухрядным переключателем, который может устанавливаться потребителем только вручную. При выпуске платы с предприятия фирмы переключатель первоначально обычно устанавливается равным 60. То есть по умолчанию действует идентификационное число платы ввода-вывода, равное 60.

2. *Регистр управления и состояния входа (приемника).* Предлагаемый фирмой DEC символьный адрес – RCSR, адресное значение которого по умолчанию производителем устанавливается равным 177560, но может изменяться вручную потребителем. Биты этого регистра определяются следующим образом:

b_0 – бит разрешения чтения. Может использоваться для продвижения считывателя перфоленты или для того чтобы привести в действие какое-либо другое устройство; очищается стартовым битом последовательных входных данных.

b_6 – бит разрешения прерываний. Обычно устанавливается ЦП с помощью инструкции MOV. Совместно с битом завершения (b_7) он генерирует сигнал запроса прерывания для ЦП.

b_7 – бит завершения. Показывает, что данные в буферном регистре готовы к пересылке в ЦП.

b_{11} – бит занятости. Устанавливается стартовым битом последовательного входного сигнала данных и очищается системой по сигналу завершения.

3. *Буферный регистр данных входа (приемника).* Предлагаемый фирмой символьный адрес – RBUF, адресное значение по умолчанию – 177562.

$b_7, b_6, \ldots, b_0$ – байт данных.

b_{12} – ошибка четности. Четность принятого байта не согласуется с ожидаемой.

b_{13} – ошибка передающего кадра. Отсутствует в последовательных данных стоповый бит.

b_{14} – ошибка наложения. Бит завершения не был очищен до поступления нового байта.

$b_{15} = b_{12} + b_{13} + b_{14}$, где знак "+" означает логическую операцию ИЛИ.

4. *Регистр управления и состояния выхода (передатчика).* Предлагаемый фирмой символьный адрес – XCSR; адресное значение по умолчанию – 177564.

b_0 – бит разрыва. Передает внешнему устройству продолжительную частоту.

b_2 – эксплуатационный бит. Подсоединяет последовательный выход передатчика к последовательному входу приемника.

b_6 – разрешение прерывания для вывода. При установленном бите готовности (b_7) генерирует сигнал запроса прерывания.

b_7 – бит готовности. Буферный регистр данных пуст и готов к приему новых данных от ЦП.

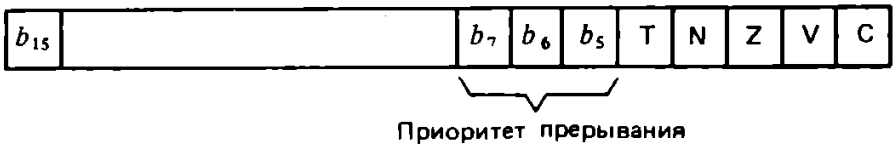
b_{11} – разрешение установки скорости передачи в бодах.

$b_{12}, \ldots, b_{15}$ – выбор скорости передачи в бодах. Скорость передачи в бодах (битах в 1с) – мера того, насколько быстро будет осуществляться последовательная передача цифровых сигналов.

5. *Выходной регистр данных выхода (передатчика).* Предлагаемый фирмой символьный адрес – XBUF, адресное значение по умолчанию – 177566.

$b_7, \ldots, b_0$ – байт данных.

Давайте напишем теперь программу для работы с прерываниями при использовании интерфейсной платы DLV11-F для считывания блока из 26 символов со считывателя перфоленты. Для простоты будем полагать, что скорость передачи в бодах уже установлена нужным образом. Вспоминаем, что битовый образ слова состояния процессора PSW выглядит так:



На рис. 8.4. показана блок-схема этой программы. Соответствующая исходная программа:

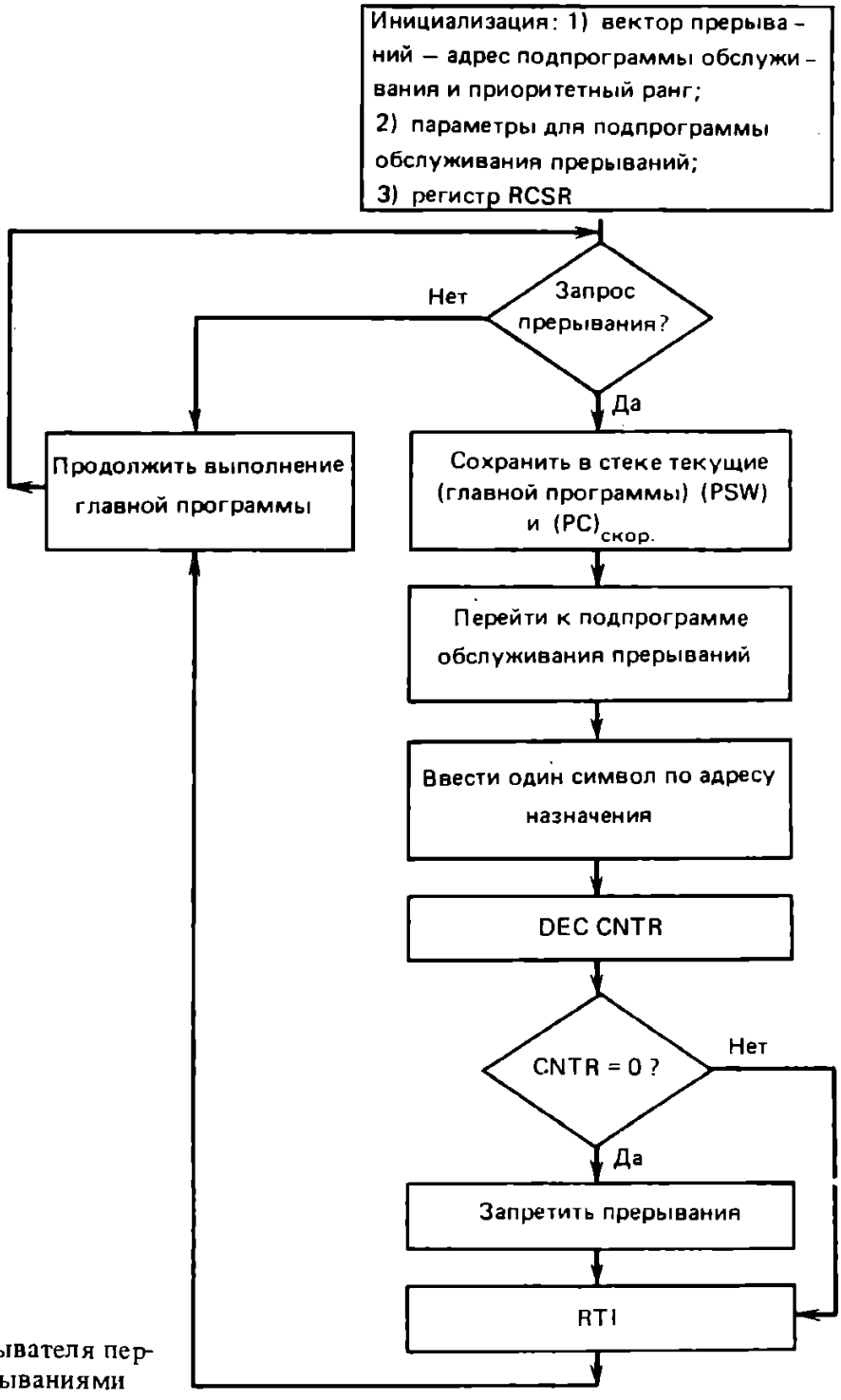


Рис. 8.4. Блок-схема для считывателя перфоленты, работающего с прерываниями

```
            RCSR = 177560           ;НАЗНАЧЕНИЕ АДРЕСНЫХ ЗНАЧЕНИЙ
            RBUF = 177562           ;КЛЮЧЕВЫМ РЕГИСТРАМ
ARRAY:      .BLKW   26
START:      .
            .
            .
1)          MTPS    #0              ;ОЧИСТИТЬ ТЕКУЩЕЕ PSW, ЧТОБЫ ГЛАВНАЯ
                                    ;ПРОГРАММА ИМЕЛА НАИМЕНЬШИЙ ПРИОРИТЕТ
                                    ;(ЭТОТ ШАГ НЕ ОБЯЗАТЕЛЬНЫЙ)
2)          MOV     #ITRPTSR,60     ;УСТАНОВИТЬ ВЕКТОР ПРЕРЫВАНИЙ
3)          MOV     #200,62         ;УСТАНОВИТЬ ПРИОРИТЕТ 4 ПЕРФОСЧИТЫВАТЕЛЯ
4)          MOV     #26,CNTR        ;УСТАНОВИТЬ СЧЕТЧИК СИМВОЛОВ
5)          MOV     #ARRAY,TEMP     ;(TEMP)' = АДРЕС НАЗНАЧЕНИЯ
6)          MOVB    #101,RCSR       ;УСТАНОВИТЬ БИТЫ РАЗРЕШЕНИЯ ЧТЕНИЯ
            .                       ;И ПРЕРЫВАНИЯ.
            .                       ;ТЕПЕРЬ ПРЕРЫВАНИЕ МОЖЕТ
            .                       ;ПРОИЗОЙТИ В ЛЮБОЙ МОМЕНТ
7)          .END    START
  (ПОДПРОГРАММА ОБСЛУЖИВАНИЯ ПРЕРЫВАНИЙ. ВАРИАНТ 1:
    ПЕРЕСЫЛКА ОДНОГО СИМВОЛА ЗА ОДНО ПРЕРЫВАНИЕ)
8)  ITRPTSR: MOV    TEMP,R0         ;R0 - УКАЗАТЕЛЬ ДАННЫХ
9)          MOVB:   RBUF,(R0)+      ;ЦП СЧИТЫВАЕТ ОДИН СИМВОЛ И ЗАПОМИНАЕТ
                                    ;ЕГО В ПАМЯТИ
10)         DEC     CNTR            ;УМЕНЬШИТЬ СЧЕТЧИК
11)         BEQ     QUIT
12)         MOV     R0,TEMP         ;СОХРАНИТЬ УКАЗАТЕЛЬ ДЛЯ ПРЕРЫВАНИЯ
                                    ;ПО СЛЕДУЮЩЕМУ СИМВОЛУ
13)         INC     RCSR            ;ПРОДВИНУТЬ ПЕРФОЛЕНТУ НА ОДИН ШАГ
14)  RETURN: RTI                    ;ВОЗВРАТ К ГЛАВНОЙ ПРОГРАММЕ
15)  QUIT:  CLR     RCSR            ;ЗАПРЕТИТЬ ПРЕРЫВАНИЯ
16)         BR      RETURN
17)  CNTR:  .BLKW   1
18)  TEMP:  .BLKW   1
            .END
```

Некоторые, не настолько очевидные строки этой программы нуждаются в пояснениях.

Строка 2: инициализирует первое слово вектора прерываний, адрес которого на плате ввода-вывода равен 60, стартовым адресом подпрограммы обслуживания прерываний.

Строка 3: инициализирует второе слово вектора прерываний, адрес которого равен 62, значением #200, устанавливающим в битах $b_7 b_6 b_5$ значение 4. Когда это слово загружается в слово состояния PSW, то приоритетный ранг становится равным 4.

Строка 4: передача параметра подпрограмме обслуживания прерываний.

Строка 5: передача параметра подпрограмме обслуживания прерываний.

Строка 6: инициализирует регистр RCSR таким образом, что $b_0 = 1$ и $b_6 = 1$.

Начиная с этой строки программы, ЦП будет постоянно выполнять главную программу до тех пор, пока не будет получен сигнал запроса прерывания. Заметим, что считыватель перфоленты представляет собой электромеханическое устройство, которому потребуется не менее 50 мс для ответа на команду разрешения прерывания. Другими словами, ЦП не получит сигнала запроса прерывания в течение десятков или даже сотен миллисекунд после строки 6. Во время этой задержки ЦП может успеть выполнить сотни инструкций главной программы.

Положим, что ЦП в конце концов получил сигнал запроса прерывания, вызванный битами готовности b_7 и разрешения прерываний b_6 в регистре RCSR. Тогда это (как описано в предыдущем разделе) приведет к выполнению процесса прерывания, производимому аппаратной логикой прерывания. В результате ЦП начнет выполнять строку 8. Адресное значение ARRAY стартового адреса массива данных назначения пересылается в регистр R0, который становится указателем данных. Строка 9 пересылает один

символ в массив назначения. Строка 10 проверяет, прочитано 26 символов или нет. Если да, то подпрограмма переходит к строке 15, в которой запишется работа логики прерывания, и затем осуществляет возврат в главную программу. Если нет, подпрограмма переходит к строке 12 и указатель сохраняется в ячейке TEMP, чтобы освободить регистр R0 для возможного использования в главной программе. Строка 13 вновь устанавливает бит разрешения прерываний от считывателя перфоленты, чтобы продвинуть перфоленту на один шаг и прочитать другой символ. Во время этой задержки ЦП может продолжить выполнение главной программы. Строка 14 осуществляет возврат из процесса прерывания в главную программу. Мы здесь полагаем, что если бит разрешения прерываний b_6 в регистре RCSR установлен (в строке 6), он остается установленным до выполнения строки *15*.

В этом примере мы позаимствовали один из регистров общего назначения (R0) для использования его в качестве указателя данных. Процесс прерывания является неожиданным, и у главной программы нет возможности узнать, в какой именно момент он начнется. Поэтому вполне возможно, что в момент возникновения прерывания в регистре R0 будет содержаться какая-то важная информация. После завершения процесса прерывания первоначальное содержимое этого регистра может оказаться потерянным. Программист должен принять меры для устранения этой проблемы. Он должен избегать использования регистра R0 для важных данных в главной программе или избегать его использования в подпрограмме обслуживания прерываний. Или в подпрограмму обслуживания прерываний должно быть включено проталкивание содержимого этого регистра в стек и выталкивание его обратно перед выходом из подпрограммы. Хотя сохранение содержимого регистров общего назначения в стеке с последующим восстановлением является общепринятым решением, которое будет проиллюстрировано в последующих примерах, давайте для простоты попытаемся в этом примере обойтись без использования регистра R0 в подпрограмме обслуживания прерываний. Мы можем модифицировать эту подпрограмму, например, так:

```
INTRPTSR:   MOVB    RBUF,@TEMP      ;TEMP БУДЕТ ИСПОЛЬЗОВАТЬСЯ
                                    ;КАК УКАЗАТЕЛЬ ДАННЫХ
            INC     TEMP
            DEC     CNTR
            BEQ     QUIT
            INC     RCSR
RETURN:     RTI
QUIT:       CLR     RCSR
            BR      RETURN
CNTR:       .BLKW   1
TEMP:       .BLKW   1
            .END
```

В этом месте читатель удивится, почему мы разработали подпрограмму обслуживания прерываний так, чтобы она пересылала один символ за одно прерывание. Почему бы за одно прерывание не переслать все 26 символов? Ниже приводится второй вариант подпрограммы обслуживания, который за одно прерывание будет пересылать полный блок данных.

```
    (ПОДПРОГРАММА ОБСЛУЖИВАНИЯ ПРЕРЫВАНИЙ. ВАРИАНТ 2:
      ПЕРЕСЫЛКА ЗА ОДНО ПРЕРЫВАНИЕ НАБОРА ДАННЫХ)
INTRPTSR: CLR   RCSR    ;ОЧИСТИТЬ (НЕ ОБЯЗАТЕЛЬНО)
                        ;БИТ РАЗРЕШЕНИЯ ПРЕРЫВАНИЙ
          MOV   TEMP,R0 ;R0 БУДЕТ УКАЗАТЕЛЕМ ДАННЫХ
L1:       MOV   RBUF,(R0)+
          DEC   CNTR
          BEQ   QUIT
          INC   RCSR
L2:       TSTB  RCSR    ;СДЕЛАНО?
```

```
        BPL     L2          ;ЕСЛИ НЕТ, ВЕТВЛЕНИЕ К L2
        BR      L1          ;ЕСЛИ ДА, ПЕРЕСЛАТЬ ДАННЫЕ ПО НАЗНАЧЕНИЮ
QUIT:   MOV     #100,RCSR   ;РАЗРЕШИТЬ ПРЕРЫВАНИЯ
        RTI
CNTR:   .BLKW   1
TEMP:   .BLKW   1
        .END
```

Имеет смысл отметить, что в этом варианте цикл L2 может занимать 50 мс или более. В течение этого времени ЦП не сможет выполнять главную программу. Следовательно, этот вариант намного медленнее, чем первый вариант.

8.3. ЕЩЕ НЕСКОЛЬКО ПРИМЕРОВ

СОХРАНЕНИЕ И ВОССТАНОВЛЕНИЕ СОДЕРЖИМОГО РЕГИСТРОВ ДЛЯ ПРОЦЕССА ПРЕРЫВАНИЯ

Поскольку момент возникновения прерывания непредсказуем, нам важно познакомиться с методами сохранения и восстановления содержимого регистров общего назначения R0, R1, . . . , R5 до и после процесса прерывания, чтобы подпрограмма обслуживания прерываний могла свободно использовать или заимствовать регистры общего назначения для своих потребностей без разрушения в регистрах данных, которые могут быть важными для главной программы. Различные методы иллюстрируются примерами, представленными на рисунках 8.5, *а* – *г*. Эти примеры полезны при рассмотрении вопросов, связанных с подпрограммами, сопрограммами, макросами и условными макросами. Они разделены на четыре части: 1) подпрограмма обслуживания прерываний; 2) программа для сохранения содержимого регистров в стековой памяти; 3) программа для восстановления содержимого регистров из стековой памяти; 4) главная (основная) программа. Мы используем системный макрос .WRITE, разработанный в нашей лаборатории, для обеспечения интерактивного режима. В вашей лаборатории такого удобного макроса может и не оказаться, но вас это не должно смущать, ведь вы уже в состоянии понять, что этот системный макрос состоит из набора макросов ввода-вывода, проиллюстрированных в предыдущей главе и использующих директиву .ASCIZ для отображения сообщений и подпрограмму преобразования двоичных чисел в код ASCII для отображения чисел. Важно, чтобы вы проследили работу программы шаг за шагом; это позволит вам познакомиться с процессом поближе. У вас может даже появиться желание иметь системный макрос .WRITE в своей лаборатории.

```
;ЭТОТ ПРИМЕР ПОКАЗЫВАЕТ ПОДПРОГРАММУ ОБРАБОТКИ ПРЕРЫВАНИЯ,
;ИСПОЛЬЗУЮЩУЮ ПОДПРОГРАММЫ ДЛЯ СОХРАНЕНИЯ И ВОССТАНОВЛЕНИЯ
;СОДЕРЖИМОГО РЕГИСТРОВ
        .TITLE  ПРЕРЫВАНИЯ (МЕТОД 1)
        .GLOBL  SAVE,RTRV
        .MCALL  .WRITE
INTRPT: JSR     PC,SAVE         ;ВЫЗОВ ПОДПРОГРАММЫ СОХРАНЕНИЯ
        .WRITE  "\\ОБРАБАТЫВАЕТСЯ_ПРЕРЫВАНИЕ_
        MOV     SP,R5           ;УСТАНОВИТЬ УКАЗАТЕЛЬ ОТОБРАЖЕНИЯ
        MOV     #6,R4           ;СЧЕТЧИК ОТОБРАЖЕНИЯ
        .WRITE  "\\СОХРАНЕННОЕ_СОДЕРЖИМОЕ_РЕГИСТРОВ:\\
L1:     .WRITE  (R5)+           ;ОТОБРАЗИТЬ СОХРАНЕННОЕ СОДЕРЖИМОЕ
        SOB     R4,L1
DMS1:   JSR     PC,RTRV         ;ВЫЗОВ ПОДПРОГРАММЫ ВОССТАНОВЛЕНИЯ
RTRN:   RTI
;
;ПОДПРОГРАММА СОХРАНЕНИЯ СОДЕРЖИМОГО РЕГИСТРОВ
        .GLOBL  SAVE
SAVE:   NOP
        MOV     R0,-(SP)        ;СОХРАНЕНИЕ СОДЕРЖИМОГО РЕГИСТРОВ
```

```
        MOV     R1,-(SP)
        MOV     R2,-(SP)
        MOV     R3,-(SP)
        MOV     R4,-(SP)
        MOV     R5,-(SP)
DMS2:   MOV     14(SP),PC       ;ВОЗВРАТ К ПОДПРОГРАММЕ ПРЕРЫВАНИЯ
;
;ПОДПРОГРАММА ВОССТАНОВЛЕНИЯ СОДЕРЖИМОГО РЕГИСТРОВ
        .GLOBL  RTRV
RTRV:   TST     (SP)+           ;ПРОПУСТИТЬ ВЕРХНИЙ ЭЛЕМЕНТ СТЕКА
        MOV     (SP)+,R5        ;ВОССТАНОВЛЕНИЕ РЕГИСТРОВ
        MOV     (SP)+,R4
        MOV     (SP)+,R3
        MOV     (SP)+,R2
        MOV     (SP)+,R1
        MOV     (SP)+,R0
DMS3:   MOV     -16(SP),(SP)    ;АДРЕС ВОЗВРАТА ДЛЯ ИНСТРУКЦИИ RTS
        RTS     PC              ;ВОЗВРАТ К ПОДПРОГРАММЕ ОБРАБОТКИ ПРЕРЫВАНИЯ
;
;ОСНОВНАЯ ПРОГРАММА
        .MCALL  .WRITE
MAIN:   MOV     #0,R0           ;ЗАПИСЬ В РЕГИСТРЫ ПРОИЗВОЛЬНЫХ ДАННЫХ
        MOV     #1,R1
        MOV     #2,R2
        MOV     #3,R3
        MOV     #4,R4
        MOV     #5,R5
        MOV     #INTRPT,60      ;ИНИЦИАЛИЗАЦИЯ ВЕКТОРА ПРЕРЫВАНИЯ
        MOV     #200,62         ;ПРИОРИТЕТ 4
        MOV     #100,177560     ;РАЗРЕШИТЬ ПРЕРЫВАНИЯ
        .WRITE  "\\НАЖМИТЕ_ЛЮБУЮ_КЛАВИШУ_ДЛЯ_ВЫЗОВА_ПРЕРЫВАНИЯ
        WAIT
        .WRITE  "\\СОДЕРЖИМОЕ_РЕГИСТРОВ_ПОСЛЕ_ПРЕРЫВАНИЯ:\\
        .WRITE  R0,R1,R2,R3,R4,R5
        HALT
        .END    MAIN
```

а)

```
;ЭТОТ ПРИМЕР ПОКАЗЫВАЕТ ПОДПРОГРАММУ ОБРАБОТКИ ПРЕРЫВАНИЯ,
;ИСПОЛЬЗУЮЩУЮ ПОДПРОГРАММЫ ДЛЯ СОХРАНЕНИЯ И ВОССТАНОВЛЕНИЯ
;СОДЕРЖИМОГО РЕГИСТРОВ
        .TITLE  ПРЕРЫВАНИЯ (МЕТОД 2)
;СОПРОГРАММА ОБРАБОТКИ ПРЕРЫВАНИЯ
        .MCALL  .WRITE
INTRPT: MOV     #SAVE,-(SP)     ;ИНИЦИАЛИЗАЦИЯ СОПРОГРАММЫ
        .WRITE  "\\ОБРАБАТЫВАЕТСЯ_ПРЕРЫВАНИЕ_
        JSR     PC,@(SP)+       ;ПЕРЕХОДА К СОПРОГРАММЕ СОХРАНЕНИЯ-
                                ;ВОССТАНОВЛЕНИЯ: #DS -) СТЕК , #SAVE -) PC
DS:     MOV     SP,R5           ;УСТАНОВИТЬ УКАЗАТЕЛЬ ОТОБРАЖЕНИЯ
        TST     (R5)+           ;ПРОПУСТИТЬ ВЕРХНИЙ ЭЛЕМЕНТ СТЕКА
        MOV     #6,R4           ;СЧЕТЧИК ОТОБРАЖЕНИЯ
        .WRITE  "\\СОХРАНЕННОЕ СОДЕРЖИМОЕ РЕГИСТРОВ:
L1:     .WRITE  (R5)+           ;ОТОБРАЗИТЬ СОХРАНЕННОЕ СОДЕРЖИМОЕ
        SOB     R4,L1
        JSR     PC,@(SP)+       ;ПЕРЕХОД К ПОДПРОГРАММЕ СОХРАНЕНИЯ-
                                ;ВОССТАНОВЛЕНИЯ: #DMS1 -) СТЕК, #RTRV -) PC
DMS1:   TST     (SP)+           ;УСТАНОВИТЬ ИСХОДНОЕ СОСТОЯНИЕ (SP)
RTRN:   RTI
;
;ПОДПРОГРАММА СОХРАНЕНИЯ СОДЕРЖИМОГО РЕГИСТРОВ
SAVE:   NOP
        MOV     R0,-(SP)        ;СОХРАНЕНИЕ СОДЕРЖИМОГО РЕГИСТРОВ
```

```
        MOV     R1,-(SP)
        MOV     R2,-(SP)
        MOV     R3,-(SP)
        MOV     R4,-(SP)
        MOV     R5,-(SP)
DMS2:   MOV     14(SP),-(SP)    ;ПОМЕСТИТЬ АДРЕС ВОЗВРАТА НА ВЕРШИНУ СТЕКА
        JSR     PC,@(SP)+       ;ПЕРЕХОД К СОПРОГРАММЕ INTRPT:
                                ;#RTRV -> СТЕК, #DS -> PC
RTRV:   TST     (SP)+           ;ПРОПУСТИТЬ ВЕРХНИЙ ЭЛЕМЕНТ СТЕКА
        MOV     (SP)+,R5        ;ВОССТАНОВЛЕНИЕ РЕГИСТРОВ
        MOV     (SP)+,R4
        MOV     (SP)+,R3
        MOV     (SP)+,R2
        MOV     (SP)+,R1
        MOV     (SP)+,R0
DMS3:   MOV     -16(SP),(SP)    ;АДРЕС ВОЗВРАТА К ПОДПРОГРАММЕ ПРЕРЫВАНИЙ,
                                ;#DMS1 -> СТЕК
        JSR     PC,@(SP)+       ;ВОЗВРАТ К ПОДПРОГРАММЕ ПРЕРЫВАНИЙ,
                                ;#MAIN -> СТЕК, #DMS1 -> PC
;
;ОСНОВНАЯ ПРОГРАММА
        .MCALL  .WRITE
MAIN:   MOV     #0,R0           ;ЗАПИСЬ В РЕГИСТРЫ ПРОИЗВОЛЬНЫХ ДАННЫХ
        MOV     #1,R1
        MOV     #2,R2
        MOV     #3,R3
        MOV     #4,R4
        MOV     #5,R5
        MOV     #INTRPT,60      ;ИНИЦИАЛИЗАЦИЯ ВЕКТОРА ПРЕРЫВАНИЯ
        MOV     #200,62         ;ПРИОРИТЕТ 4
        MOV     #100,177560     ;РАЗРЕШИТЬ ПРЕРЫВАНИЯ
        .WRITE  "\\НАЖМИТЕ ЛЮБУЮ КЛАВИШУ ДЛЯ ВЫЗОВА ПРЕРЫВАНИЯ
        WAIT
        .WRITE  "\\СОДЕРЖИМОЕ РЕГИСТРОВ ПОСЛЕ ПРЕРЫВАНИЯ:
        .WRITE  R0,R1,R2,R3,R4,R5
        HALT
        .END    MAIN
```

б)

```
;ЭТОТ ПРИМЕР ПОКАЗЫВАЕТ ПОДПРОГРАММУ ОБРАБОТКИ ПРЕРЫВАНИЙ,
;ИСПОЛЬЗУЮЩУЮ МАКРОСЫ ДЛЯ СОХРАНЕНИЯ И ВОССТАНОВЛЕНИЯ
;СОДЕРЖИМОГО РЕГИСТРОВ.
        .TITLE  ПРЕРЫВАНИЯ (МЕТОД 3)
;МАКРОС СОХРАНЕНИЯ СОДЕРЖИМОГО РЕГИСТРОВ
        .MACRO  SAVE
SAVE:   NOP
        MOV     R0,-(SP)        ;СОХРАНЕНИЕ СОДЕРЖИМОГО РЕГИСТРОВ
        MOV     R1,-(SP)
        MOV     R2,-(SP)
        MOV     R3,-(SP)
        MOV     R4,-(SP)
        MOV     R5,-(SP)
.ENDM   SAVE
;
;МАКРОС ВОССТАНОВЛЕНИЯ СОДЕРЖИМОГО РЕГИСТРОВ
        .MACRO  RTRV
RTRV:   NOP
        MOV     (SP)+,R5        ;ВОССТАНОВЛЕНИЕ СОДЕРЖИМОГО РЕГИСТРОВ
        MOV     (SP)+,R4
        MOV     (SP)+,R3
        MOV     (SP)+,R2
        MOV     (SP)+,R1
        MOV     (SP)+,R0
```

```
.ENDM   RTRV
;
        .TITLE  INTRPT
        .MCALL  .WRITE
INTRPT: SAVE
        .WRITE  "\\ОБРАБАТЫВАЕТСЯ ПРЕРЫВАНИЕ__
        MOV     SP,R5           ;УСТАНОВИТЬ УКАЗАТЕЛЬ ОТОБРАЖЕНИЯ
        MOV     #6,R4           ;СЧЕТЧИК ОТОБРАЖЕНИЯ
        .WRITE  "\\СОХРАНЕННОЕ_СОДЕРЖИМОЕ_РЕГИСТРОВ:\\
L1:     .WRITE  (R5)+           ;ОТОБРАЗИТЬ СОХРАНЕННОЕ СОДЕРЖИМОЕ
        SOB     R4,L1
M1:     RTRV
RTRN:   RTI
;
;ОСНОВНАЯ ПРОГРАММА
        .MCALL  .WRITE
MAIN:   MOV     #0,R0           ;ЗАПИСЬ В РЕГИСТРЫ ПРОИЗВОЛЬНЫХ ДАННЫХ
        MOV     #1,R1
        MOV     #2,R2
        MOV     #3,R3
        MOV     #4,R4
        MOV     #5,R5
        MOV     #INTRPT,60      ;ИНИЦИАЛИЗАЦИЯ ВЕКТОРА ПРЕРЫВАНИЯ
        MOV     #200,62         ;ПРИОРИТЕТ 4
        MOV     #100,177560     ;РАЗРЕШИТЬ ПРЕРЫВАНИЯ
        .WRITE  "\\НАЖМИТЕ_ЛЮБУЮ_КЛАВИШУ_ДЛЯ_ВЫЗОВА_ПРЕРЫВАНИЯ
        WAIT
        .WRITE  "\\СОДЕРЖИМОЕ_РЕГИСТРОВ_ПОСЛЕ_ПРЕРЫВАНИЯ:\\
        .WRITE  R0,R1,R2,R3,R4,R5
        HALT
        .END    MAIN
```

```
РЕЗУЛЬТАТ ВЫПОЛНЕНИЯ:   НАЖМИТЕ ЛЮБУЮ КЛАВИШУ ДЛЯ ВЫЗОВА ПРЕРЫВАНИЯ
                        ОБРАБАТЫВАЕТСЯ ПРЕРЫВАНИЕ
                        СОХРАНЕННОЕ СОДЕРЖИМОЕ РЕГИСТРОВ:
                        5 4 3 2 1 0
                        СОДЕРЖИМОЕ РЕГИСТРОВ ПОСЛЕ ПРЕРЫВАНИЯ:
                        0 1 2 3 4 5
```

в)

```
;ЭТОТ ПРИМЕР ПОКАЗЫВАЕТ ПОДПРОГРАММУ ОБРАБОТКИ ПРЕРЫВАНИЯ,
;ИСПОЛЬЗУЮЩУЮ УСЛОВНЫЕ МАКРОСЫ ДЛЯ СОХРАНЕНИЯ И ВОССТАНОВЛЕНИЯ
;СОДЕРЖИМОГО РЕГИСТРОВ.
        .TITLE  ПРЕРЫВАНИЯ (МЕТОД 4)
;УСЛОВНЫЙ МАКРОС СОХРАНЕНИЯ СОДЕРЖИМОГО РЕГИСТРОВ
        .MACRO  SAVE    R0,R1,R2,R3,R4,R5
.IF     NB      R0      ;NB ОЗНАЧАЕТ "НЕ ПУСТО"
        MOV     R0,-(SP)
.ENDC
.IF     NB      R1
        MOV     R1,-(SP)
.ENDC
.IF     NB      R2
        MOV     R2,-(SP)
.ENDC
.IF     NB      R3
        MOV     R3,-(SP)
.ENDC
.IF     NB      R4
        MOV     R4,-(SP)
.ENDC
.IF     NB      R5
        MOV     R5,-(SP)
```

```
.ENDC
.ENDM   SAVE
;
;УСЛОВНЫЙ МАКРОС ВОССТАНОВЛЕНИЯ СОДЕРЖИМОГО РЕГИСТРОВ
        .MACRO  RTRV    R5,R4,R3,R2,R1,R0
.IF     NB      R5
        MOV     (SP)+,R5
.ENDC
.IF     NB      R4
        MOV     (SP)+,R4
.ENDC
.IF     NB      R3
        MOV     (SP)+,R3
.ENDC
.IF     NB      R2
        MOV     (SP)+,R2
.ENDC
.IF     NB      R1
        MOV     (SP)+,R1
.ENDC
.IF     NB      R0
        MOV     (SP)+,R0
.ENDC
.ENDM   RTRV
;
        .TITLE  INTRPT
        .MCALL  .WRITE
INTRPT: SAVE    R0,R5
        .WRITE  #\\ОБРАБАТЫВАЕТСЯ_ПРЕРЫВАНИЕ__
        MOV     SP,R5           ;УСТАНОВИТЬ УКАЗАТЕЛЬ ОТОБРАЖЕНИЯ
        MOV     #2,R4           ;СЧЕТЧИК ОТОБРАЖЕНИЯ
        .WRITE  #\\СОХРАНЕННОЕ_СОДЕРЖИМОЕ_РЕГИСТРОВ:\\
L1:     .WRITE  (R5)+           ;ОТОБРАЗИТЬ СОХРАНЕННОЕ СОДЕРЖИМОЕ
        SOB     R4,L1
M1:     RTRV
RTRN:   RTI
;
;ОСНОВНАЯ ПРОГРАММА
        .MCALL  .WRITE
MAIN:   MOV     #0,R0           ;ЗАПИСЬ В РЕГИСТРЫ ПРОИЗВОЛЬНЫХ ДАННЫХ
        MOV     #1,R1
        MOV     #2,R2
        MOV     #3,R3
        MOV     #4,R4
        MOV     #5,R5
        MOV     #INTRPT,60      ;ИНИЦИАЛИЗАЦИЯ ВЕКТОРА ПРЕРЫВАНИЯ
        MOV     #200,62         ;ПРИОРИТЕТ 4
        MOV     #100,177560     ;РАЗРЕШИТЬ ПРЕРЫВАНИЯ
        .WRITE  #\\НАЖМИТЕ ЛЮБУЮ_КЛАВИШУ_ДЛЯ_ВЫЗОВА_ПРЕРЫВАНИЯ
        WAIT
        .WRITE  #\\СОДЕРЖИМОЕ_РЕГИСТРОВ_ПОСЛЕ_ПРЕРЫВАНИЯ:\\
        .WRITE  R0,R1,R2,R3,R4,R5
        HALT
        .END    MAIN

РЕЗУЛЬТАТ ВЫПОЛНЕНИЯ:   НАЖМИТЕ ЛЮБУЮ КЛАВИШУ ДЛЯ ВЫЗОВА ПРЕРЫВАНИЯ
                        ОБРАБАТЫВАЕТСЯ ПРЕРЫВАНИЕ
                        СОХРАНЕННОЕ СОДЕРЖИМОЕ РЕГИСТРОВ:
                        5 0
                        СОДЕРЖИМОЕ РЕГИСТРОВ ПОСЛЕ ПРЕРЫВАНИЯ:
                        0 1 2 3 4 5
```

г)

Рис. 8.5. Прерывания с сохранением и восстановлением регистров

ПРОГРАММА ВВОДА-ВЫВОДА ДАННЫХ С ПОМОЩЬЮ ОЧЕРЕДИ БУФЕРОВ

В этом примере мы попытаемся проиллюстрировать, как можно воспользоваться всем изученным нами материалом для разработки полезной прикладной программы, предназначенной для сбора данных, и рассмотрим следующие возможности:

1. Использование системных макросов, макросов, определенных пользователем, макросов с автоматическим назначением локальных адресов, подпрограмм и прерываний для выполнения достаточно сложной задачи.
2. Использование программных контрольных точек проверки для отладки.
3. Использование концепции модульности для разбиения сложной программы на модули и разработки их по отдельности с последующей компоновкой в завершенную программу или задачу.

Формулировка задачи

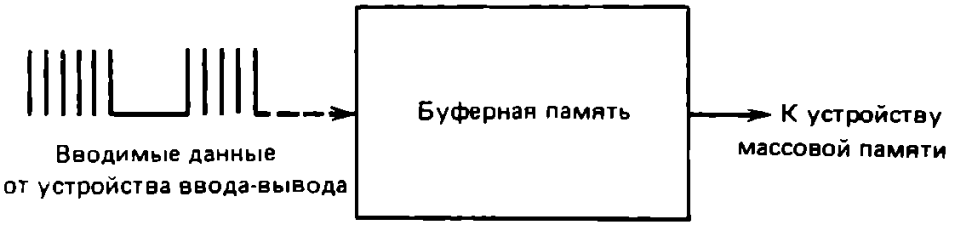
Нередко возникает необходимость собирать данные для последующей обработки, когда они поступают в форме раз за разом возникающих фрагментов или элементов. Во время такой "вспышки" поступления данных нам бы хотелось собирать данные так быстро, как они поступают. В промежутках между "вспышками" мы могли бы обеспечить длительное сохранение собранных данных для обработки в будущем, записав их на внешнее устройство массовой памяти. Очевидно, что в таком приложении для сбора данных во время "вспышки" нам нужна быстрая память, тогда как время доступа к устройству массовой памяти не столь критично при условии, что время между "вспышками" достаточно велико. Однако в некоторых случаях требования к скорости доступа могут быть как раз противоположны только что описанным. То есть входные данные могут поступать нечасто, а моменты их поступления – непредсказуемы. В подобной ситуации нам не удобно привязываться к устройству массовой памяти. Вместо этого мы постараемся отвести небольшой блок оперативной памяти в качестве буфера для сбора данных, поступающих "тонкой струйкой", с последующей транспортировкой собранных в буфере данных на устройство массовой памяти или по другому адресу назначения. Поток данных для описанного процесса показан на рис. 8.6, *а*. Теперь мы покажем, как этот процесс может быть реализован при использовании методов прерываний и подпрограмм. Для буферизации мы воспользуемся очередью буферов.

Структура буфера данных

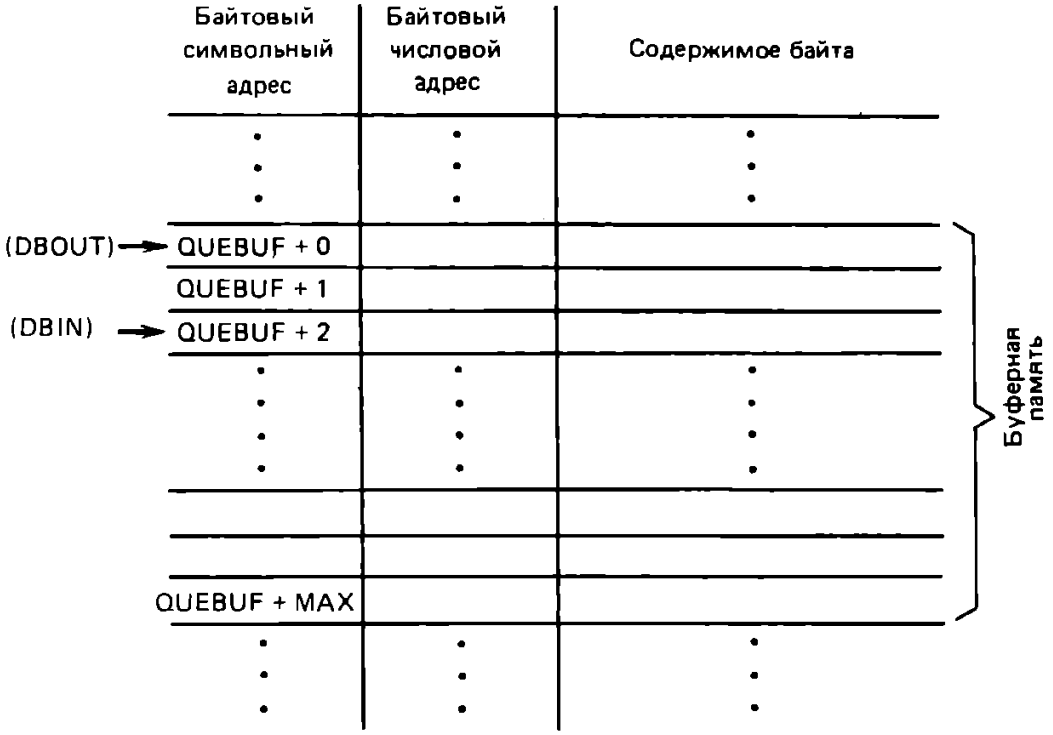
На рис. 8.6, *б* показана карта буферной памяти на байтовой основе. Начальным адресом буфера является метка QUEBUF, а его емкость равна (MAX + 1) байт или 1/2 (MAX + 1) слов. Мы определили также два указателя данных: DBIN – вход в очередь буфера данных; DBOUT – выход из очереди буфера данных. Первый из них всегда указывает на ту ячейку, в которую будут записываться следующие данные, а второй – на ячейку, содержимое которой является следующим кандидатом на вывод из буфера. Интуитивно мы можем чувствовать, что для обслуживания процессов ввода и вывода достаточно лишь одного указателя данных. Например, можно использовать один указатель данных, чтобы сначала "насытить" буфер, а затем сбросить его на начальный адрес буфера и использовать для вывода данных из буфера.

Хотя это просто и понятно, конфигурация с одним указателем имеет серьезный недостаток. Что случится, если в процессе вывода данных из буфера поступят новые данные? Более того, как можно гарантировать, что новые данные не будут записаны поверх уже собранных данных, которые еще не выведены из буфера? Что если для одиночной "вспышки" поступления данных потребуется бóльшая емкость буфера, нежели зарезервированная? Возможно ли выводить данные в период "вспышки" за короткий промежуток между двумя смежными поступлениями данных?

Конфигурацию с двумя указателями можно лучше разъяснить, если представить буфер в циклической форме, как это показано на рис. 8.7. Первоначально оба указате-



а) Схема потока данных



б) Карта буферной памяти

Рис. 8.6. Сбор данных с использованием буферной памяти

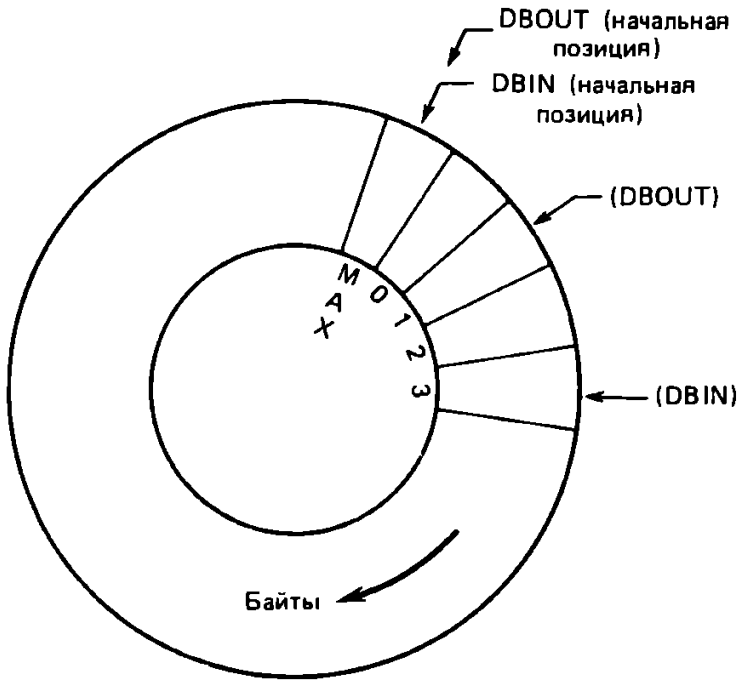


Рис. 8.7. Концепция циклической буферной памяти

ля указывают на максимальный байт MAX. По мере ввода данных в буфер содержимое указателя DBIN увеличивается в направлении по часовой стрелке, в то время как указатель DBOUT все еще остается на месте. Когда данные перестанут поступать, собранные данные, располагающиеся в памяти между указателями, будут выведены из буфера, а указатель DBOUT соответственно увеличен. В результате по мере вывода данных из буфера "разрыв" между двумя указателями будет уменьшаться. Таким образом мы получаем буфер бесконечной емкости, если только разрыв всегда меньше, чем MAX, и больше, чем нуль байт. Мы можем использовать буферное пространство вновь и вновь без необходимости заботиться о том, чтобы не были испорчены действительные данные или были выведены неверные данные. Для поддержания такой идеальной ситуации нам необходимо иметь в виду следующие условия:

1. Совпадение (DBIN) с (DBOUT), т. е. (DBIN) = (DBOUT), означает, что буфер полон.
2. Совпадение (DBOUT) с (DBIN), т. е. (DBOUT) = (DBIN), подразумевает, что буфер пуст.
3. Всякий раз, когда любой из указателей равен MAX, он должен быть сброшен в нулевую позицию.

Алгоритм

Теперь мы разработаем алгоритм, который может быть реализован в лабораторных условиях. Для удобства выделим отдельный блок памяти, который будет имитировать устройство массовой памяти, предназначенное для выходных данных. Функциональная блок-схема системы показана на рис. 8.8. Здесь мы используем метод преры-

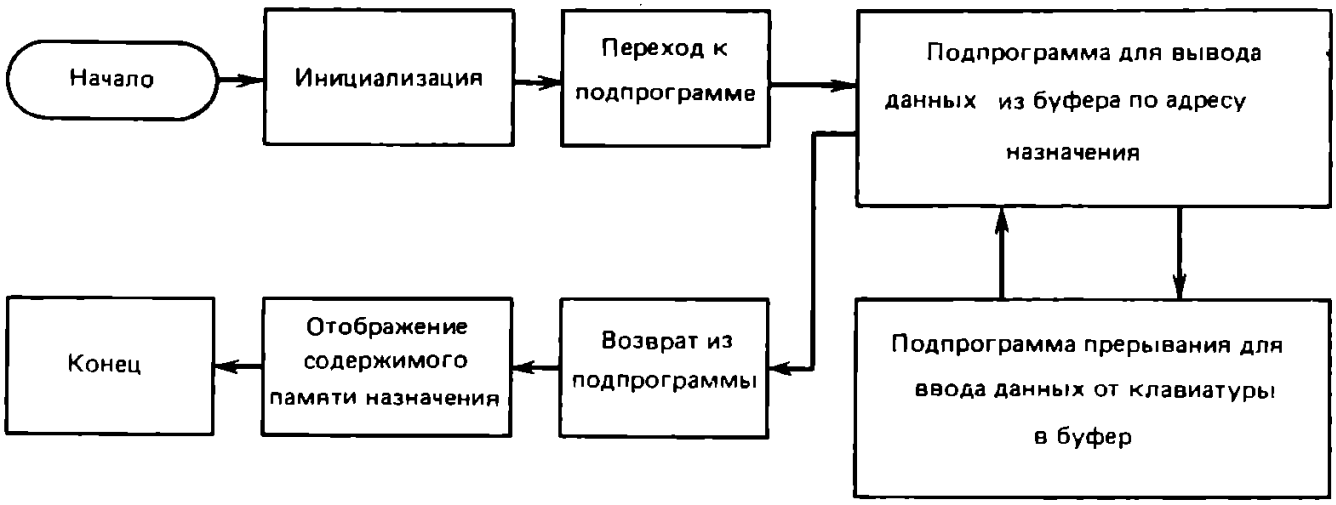


Рис. 8.8. Функциональная блок-схема системы

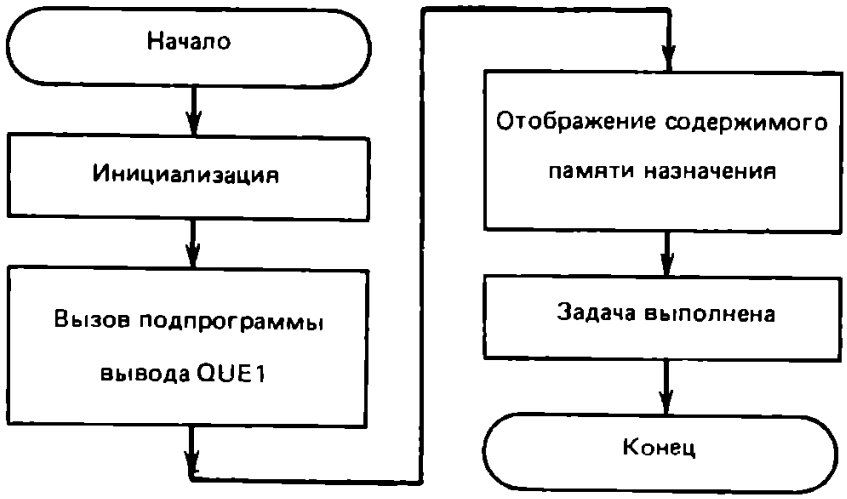


Рис. 8.9. Блок-схема главной программы QUE0

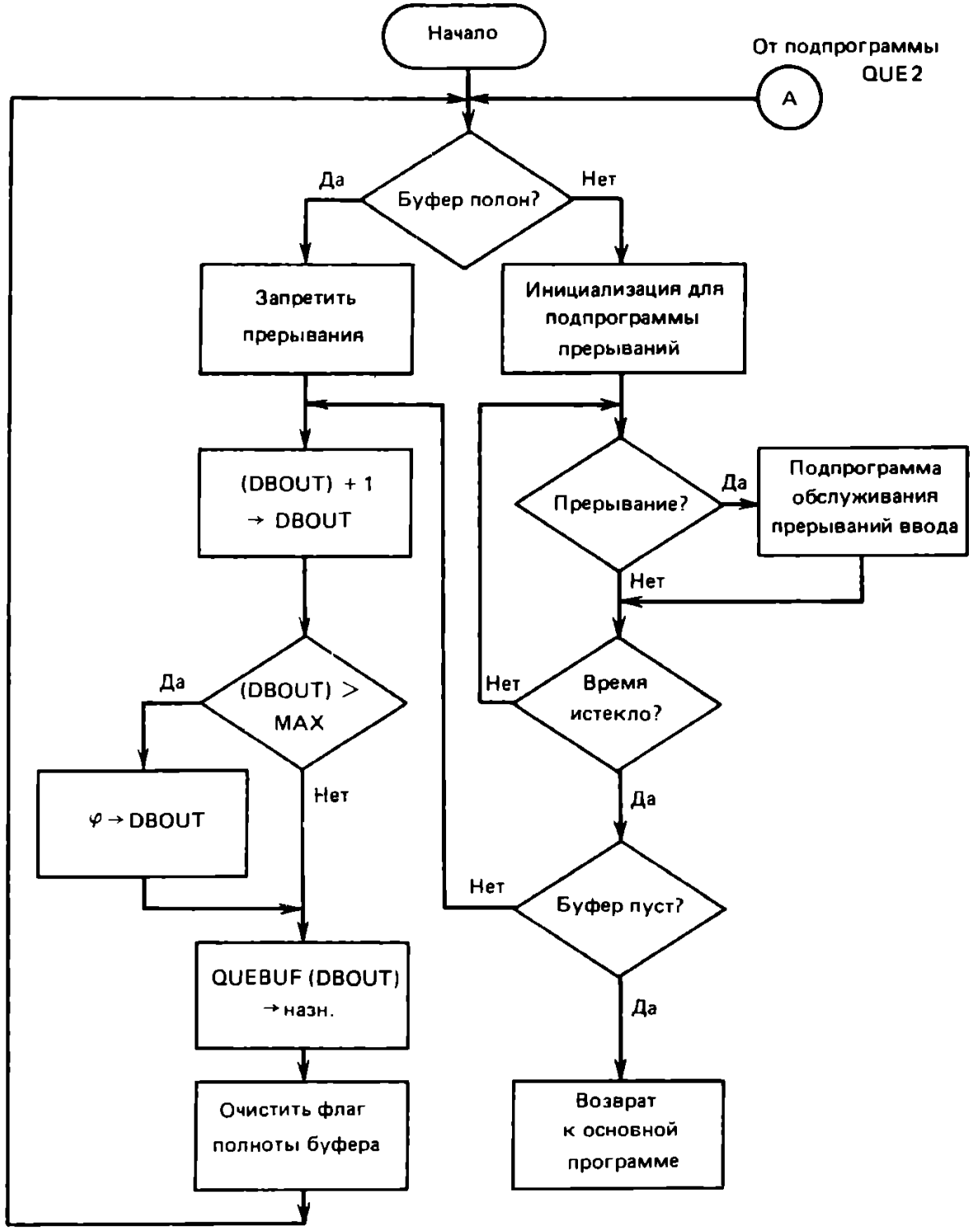


Рис. 8.10. Блок-схема подпрограммы для вывода данных QUE1

ваний для ввода данных в циклический буфер и метод подпрограмм для вывода данных на устройство назначения из циклического буфера плюс главную программу для вызова подпрограммы и отображения результатов.

Подпрограмма завершит свою работу, если: 1) данные из буфера полностью выведены по назначению; 2) входных данных больше не поступает. В соответствии с условием 1) она будет проверять, пуст буфер или нет, и в соответствии с условием 2) — не приходят ли данные в течение определенного промежутка времени.

Процесс прерывания начинается тогда, когда на клавиатуре нажимают какую-либо клавишу. Поскольку среди кодов ASCII есть группа управляющих кодов, восьмеричные значения которых меньше 40 и которые не являются печатаемыми и не могут

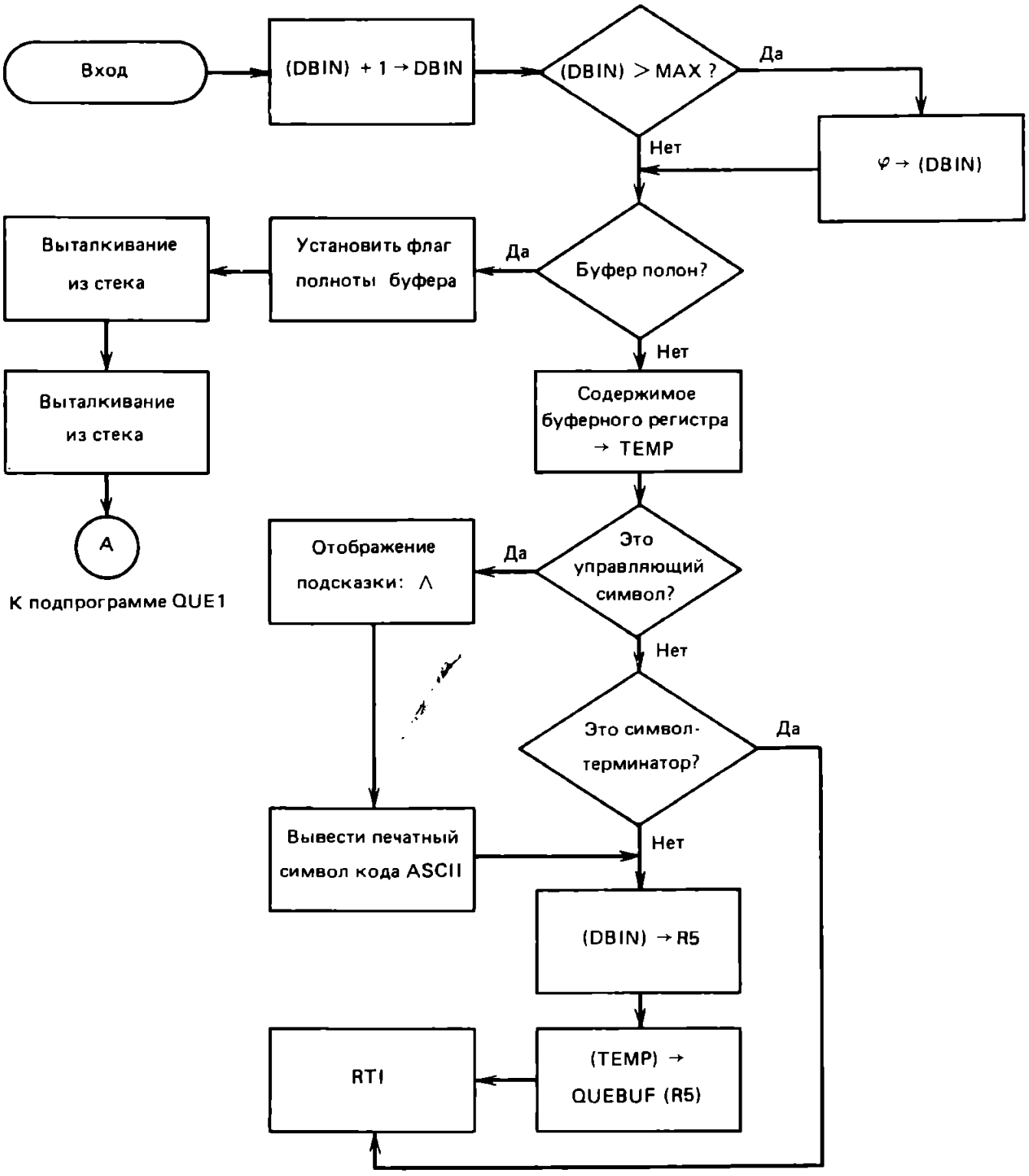


Рис. 8.11. Блок-схема подпрограммы обслуживания прерываний при вводе данных QUE2

быть отображены на ЭЛТ, подпрограмма обслуживания прерываний отображает их в виде двух символов: стрелки вверх ↑ (ее код равен 136) и символа, восьмеричный код которого равен 100 + код управляющей клавиши. Например, если нажимают клавишу DLE, то будет отображена пара ↑ P. Кроме того, подпрограмма обслуживания прерываний завершает свою работу, если нажимают клавишу восклицательного знака или если выполняется условие полноты буфера. Три исходных файла для главной или основной программы, подпрограммы вывода данных и подпрограммы обслуживания прерываний ввода данных мы назвали соответственно QUE0, QUE 1 и QUE 2. Их блок-схемы показаны на рис. 8.9 – 8.11.

Программа

Как видно, программа состоит из трех модулей (QUE1, QUE2 и QUE0); исходные файлы которых представлены на рис. 8.12 – 8.14 соответственно. Обратите внимание, что в качестве "посредника" при взаимодействии пользователя с ЭВМ мы использовали системный макрос .WRITE. Иногда его применяют как своеобразный тестер, чтобы посмотреть, что происходит в программе в процессе выполнения. В нескольких местах модуля прерываний мы вставили инструкцию NOP для демонстрации того, что она может использоваться для установки контрольных точек, удобных при отладке. При работе с программой пользователь легко может заменить эту инструкцию инструкцией HALT, и тогда программа будет останавливаться в этой точке, так что пользователь сможет предпринять поиск возникшей ошибки.

```
        .TITLE  QUE1
        .GLOBL  QUE2,DBIN,DBOUT,BFULL,QUEBUF,MAX,QUE1,DST
        .MCALL  WRITE
        .MACRO  PRINT   R,?B
B:      TSTB    177564
        BPL     B
        MOVB    R,177566
        .ENDM   PRINT
;
QUE1:   TST     BFULL
        BNE     FULL
        MTPS    #0              ;ИНИЦИАЛИЗАЦИЯ PSW
        MOVB    #100,177560     ;РАЗРЕШЕНИЕ ПРЕРЫВАНИЙ
        MOV     #QUE2,60        ;УСТАНОВКА ВЕКТОРА ПРЕРЫВАНИЙ
        MOV     #340,62         ;УСТАНОВКА ВЕКТОРА ПРЕРЫВАНИЙ
        CLR     R3              ;ИНИЦИАЛИЗАЦИЯ ТАЙМЕРА
        CLR     R4
        .WRITE  "\\ОЖИДАНИЕ_ВВОДА_В_ТЕЧЕНИЕ_ОГРАНИЧЕННОГО_ПРЕМЕНИ__
WAIT:   INC     R3
        BNE     WAIT
        INC     R4
        CMP     TLIMIT,R4       ;ВРЕМЯ ИСЧЕРПАНО?
        BPL     WAIT
        .WRITE  "\\СОДЕРЖИМОЕ_DBOUT:__,DBOUT
        .WRITE  "\\СОДЕРЖИМОЕ_DBIN:__,DBIN
        .WRITE  "\\ДАННЫЕ_В_БУФЕРЕ_DST:__,DST
EMPTY:  CMP     DBOUT,DBIN      ;БУФЕР ПУСТ?
        BNE     L1              ;НЕТ, ВСТАВЛЕНИЕ К L1
        RTS     PC              ;ДА, ВОЗВРАТ К ОСНОВНОЙ ПРОГРАММЕ
FULL:   CLR     177560          ;ЗАПРЕТИТЬ ПРЕРЫВАНИЯ
        .WRITE  "\\БУФЕР_ПОЛОН.__ЖДИТЕ!__
L1:     INC     DBOUT
        CMP     DBOUT,MAX       ;DBOUT УКАЗЫВАЕТ НА MAX?
        BLE     XFER
        CLR     DBOUT           ;СБРОСИТЬ DBOUT
XFER:   MOV     DBOUT,R2
        PRINT   QUEBUF(R2)      ;ПРОСМОТР БУФЕРА
        MOVB    QUEBUF(R2),DST(R1)   ;ДАННЫЕ --> ПО НАЗНАЧЕНИЮ
        PRINT   DST(R1)
        INC     R1
        CLR     BFULL           ;СБРОСИТЬ ФЛАГ ПОЛНОТЫ
        BR      QUE1            ;СЛЕДУЮЩИЕ ДАННЫЕ
DST:    .BLKB   20              ;МЕСТО ДЛЯ ДАННЫХ
        .EVEN
TLIMIT: .WORD   7
```

Рис. 8.12. Исходный текст подпрограммы QUE1

```
        .TITLE  QUE2
        .GLOBL  QUE1,QUE2,DBIN,DBOUT,MAX,BFULL,QUEBUF
        .MCALL  .WRITE
        .MACRO  READ    TEMP,?A
A:      TSTB    177560
        BPL     A
        MOVB    177562,TEMP
        CLR     R3
        CLR     R4
        .ENDM   READ
;
        .MACRO  PRINT   TEMP,?B
B:      TSTB    177564
        BPL     B
        MOVB    TEMP,177566
        .ENDM   PRINT
;
QUE2:   INC     DBIN
        .WRITE  "\\СОДЕРЖИМОЕ_DBIN:__,DBIN
        .WRITE  "\\СОДЕРЖИМОЕ_MAX:__,MAX,"\\СОДЕРЖИМОЕ_TEMP:__,TEMP
        CMP     DBIN,MAX        ;DBIN РАВНО ВЕРХНЕМУ ПРЕДЕЛУ?
        BLE     FULCHK
        CLR     DBIN            ;СБРОСИТЬ DBIN
FULCHK: CMP     DBIN,DBOUT      ;БУФЕР ПОЛОН?
        BNE     RD              ;ПЕРЕЙТИ К ЧТЕНИЮ
FULSET: INC     BFULL           ;УСТАНОВИТЬ ФЛАГ ПОЛНОТЫ БУФЕРА
        TST     (SP)+           ;УДАЛИТЬ СОХРАНЕННОЕ (PC)
        TST     (SP)+           ;УДАЛИТЬ СОХРАНЕННОЕ (PSW)
        JMP     QUE1            ;БУФЕР ПОЛОН, ВЫВЕСТИ
RD:     READ    TEMP            ;ОГРАНИЧЕННОЕ ПО ВРЕМЕНИ ЧТЕНИЕ
        CMPB    TEMP,#40        ;НАЖАТА КЛАВИША ПЕЧАТАЕМОГО ЗНАКА?
        BGT     NORM
CNTRL:  PRINT   #136            ;НАПЕЧАТАТЬ СИМВОЛ "^"
        BIS     #100,TEMP       ;TEMP <- (КОД КЛАВИШИ CTRL + 100)
NORM:   PRINT   TEMP
        CMPB    #41,TEMP        ;НАЖАТА КЛАВИША СИМВОЛА ЗАВЕРШЕНИЯ "!"?
        BNE     SKP1
STOP:   RTI                     ;ВОЗВРАТ ИЗ ПРЕРЫВАНИЯ
SKP1:   NOP
        MOV     DBIN,R5
        MOVB    TEMP,QUEBUF(R5)
        NOP                     ;КОНТРОЛЬНАЯ ТОЧКА ДЛЯ ОТЛАДКИ
        BR      STOP
DBIN:   .WORD   17
DBOUT:  .WORD   17
BFULL:  .WORD   0
MAX:    .WORD   17
TEMP:   .BLKB   1
QUEBUF: .BLKB   20
        .EVEN
```

Рис. 8.13. Исходный текст модуля прерываний QUE2

```
;В ЭТОЙ ПРОГРАММЕ ДЛЯ ОБРАЗОВАНИЯ ОЧЕРЕДИ ИСПОЛЬЗУЮТСЯ
;ПОДПРОГРАММА ОБРАБОТКИ ПРЕРЫВАНИЙ ВВОДА В БУФЕР ДАННЫХ (QUE2)
;И ПОДПРОГРАММА ВЫВОДА ДАННЫХ (QUE1).
;ОПРЕДЕЛЕНИЯ:
; QUEBUF -- НАЧАЛЬНЫЙ АДРЕС БУФЕРА, ФИКСИРОВАННОЕ ЗНАЧЕНИЕ
; MAX    -- ПОЛНОЕ ЧИСЛО БАЙТОВ В БУФЕРЕ
; DBOUT  -- УКАЗАТЕЛЬ В БУФЕРЕ ДЛЯ СЛЕДУЮЩИХ ВЫВОДИМЫХ ДАННЫХ
; DBIN   -- УКАЗАТЕЛЬ В БУФЕРЕ ДЛЯ СЛЕДУЮЩИХ ВВОДИМЫХ ДАННЫХ
; DST    -- НАЧАЛЬНЫЙ АДРЕС ОБЛАСТИ НАЗНАЧЕНИЯ ДЛЯ ДАННЫХ
;
;
```

```
        .TITLE  QUE0
        .GLOBL  DST,QUE1
        .MCALL  .WRITE
GO:     CLR     R1
        JSR     PC,QUE1
        .WRITE  #DST
        .WRITE  #"\\ЗАДАЧА_ВЫПОЛНЕНА__
        JMP     165000
        .END    GO
```

Рис. 8.14. Исходный текст главной программы QUE0

8.4. БУДИЛЬНИК

Будильник — это еще один пример устройства, которое можно имитировать на ЭВМ PDP-11, воспользовавшись системными макросами и макросами, определенными пользователями. Этот пример, естественно, приводит к следующему примеру, в котором имитируется секундомер при использовании метода прерываний.

Очевидно, для начала, нам нужен макрос для подсчета интервалов времени. Опираясь на информацию о времени выполнения инструкций, поставляемую производителем ЭВМ (приложение В), мы создаем макрос SEC для подсчета интервала в 1с. Кроме того, нам необходимы следующие макросы:

1. Для включения звонка терминала по достижении предварительно установленного интервала времени.
2. Для проверки, не равно ли текущее время предварительно установленному времени.
3. Для проверки переполнения счетчиков на 60 с, 60 мин и 24 ч.

Блок-схема программы для имитации будильника показана на рис. 8.15. После инициализации системы клиенту или пользователю предлагается ввести текущее время, желаемое время срабатывания звонка и интервал отображения, а также желаемую частоту корректировки текущего времени и отображения его на ЭЛТ. Программа назначает регистр R3 для использования в качестве счетчика интервала отображения единицами по одной секунде. Затем имитируемые часы начинают идти. После каждого односекундного интервала с помощью макроса OVERFLOW осуществляется проверка на переполнение счетчиков секунд, минут и часов. Обратите внимание, что в этом макросе в качестве счетчиков для секунды, минуты и часа назначены соответственно регистры R0, R1 и R2. Затем с помощью макроса ALARMCK мы проверяем текущее время на совпадение с желаемым временем звонка. Как показано в блок-схеме, при любом результате предпринимается соответствующая последовательность действий. Исходный файл этой программы представлен на рис. 8.16.

Приводя этот пример, мы не хотим, конечно, предложить пользоваться ЭВМ PDP-11 вместо будильника; он служит лишь упражнением для получения опыта программирования ввода-вывода.

8.5. СЕКУНДОМЕР

В этом разделе мы покажем, как можно использовать метод прерываний для модификации программы имитации будильника с целью превращения ее в программу секундомера. Но на этот раз мы ограничимся лишь одним системным макросом — .EXIT. Другими словами, чтобы не пользоваться системными макросами .WRITE, .READ и т. п., мы напишем собственные макросы ввода-вывода. Читатель может и не знать, что для отображения числового содержимого регистра, находящегося в двоичном представлении, должен быть написан макрос, преобразующий двоичное число в эквивалентные символы кода ASCII. В примере с секундомером мы проиллюстрируем такой процесс. Блок-схема программы представлена на рис. 8.17.

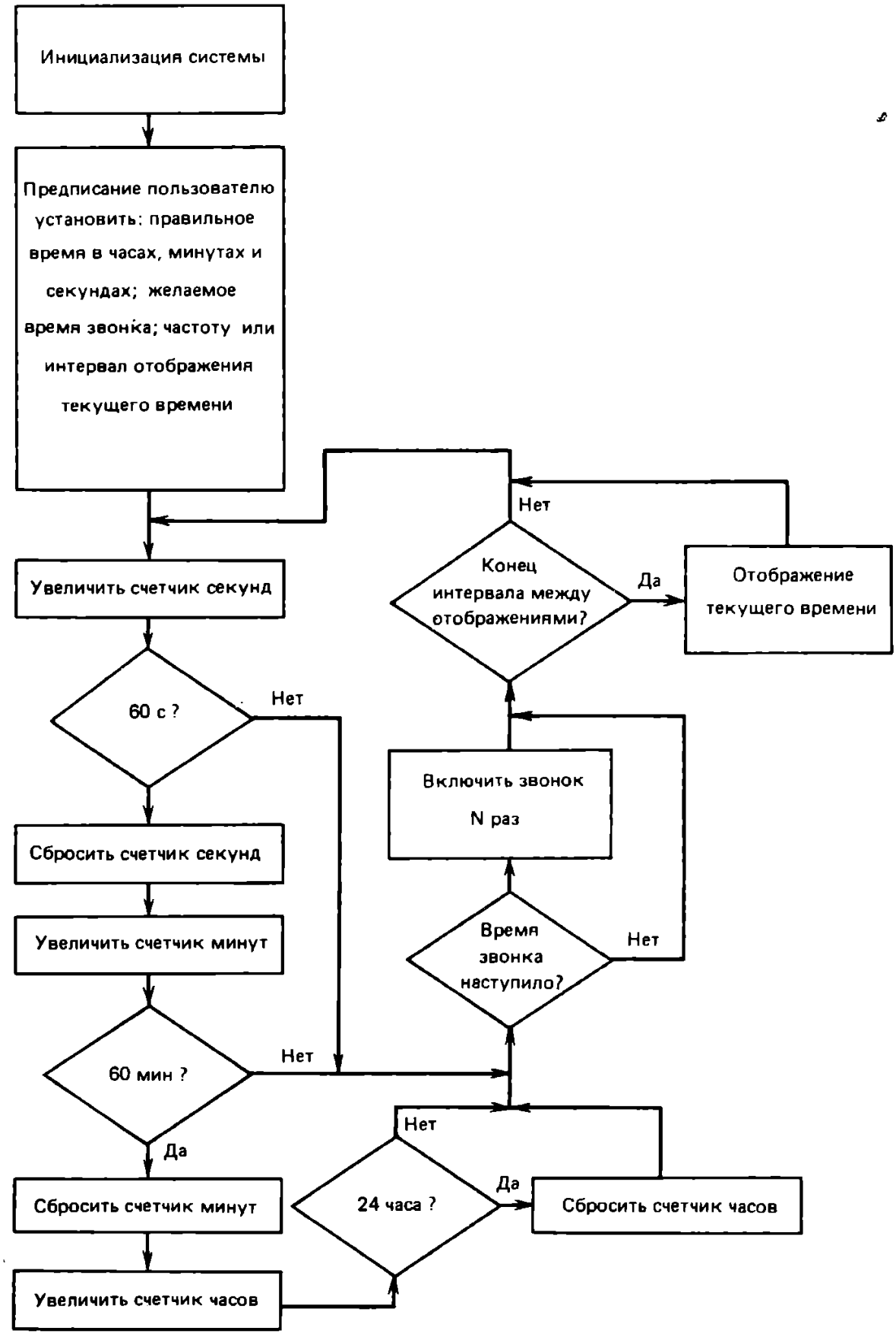


Рис. 8.15. Блок-схема программы для имитации будильника

```
;ЭТА ПРОГРАММА МОДЕЛИРУЕТ БУДИЛЬНИК С ИСПОЛЬЗОВАНИЕМ
;СИСТЕМНЫХ МАКРОСОВ И МАКРОСОВ, ОПРЕДЕЛЕННЫХ ПОЛЬЗОВАТЕЛЕМ
;
        .MCALL  .WRITE,.READ,.RAD,.EXIT
.MACRO  SEC ?L1
        MOV     R5,-(SP)        ;СОХРАНИТЬ СТАРОЕ СОДЕРЖИМОЕ R5
        CLR     R5
L1:     INC     R5
        CMP     #0,R5           ;ОДНОСЕКУНДНЫЙ ЦИКЛ
        BNE     L1
        MOV     (SP)+,R5
.ENDM   SEC
;
.MACRO  RBELL   N,LOOP
        MOV     R5,-(SP)
        MOV     #N,R5           ;ПРОЗВОНИТЬ N РАЗ
LOOP:   .WRITE  "#007           ;ПРОЗВОНИТЬ ОДИН РАЗ
        SEC                     ;ОДНОСЕКУНДНАЯ ЗАДЕРЖКА
        SOB     R5,LOOP
        MOV     (SP)+,R5
.ENDM   RBELL
;
        .MACRO  ALARMCK ALARMH,ALARMM,ALARMS,?NOTCK
        CMP     ALARMS,R0
        BNE     NOTCK
        CMP     ALARMM,R1
        BNE     NOTCK
        CMP     ALARMH,R2
        BNE     NOTCK
        RBELL   5,DLOOP
NOTCK:  NOP
.ENDM   ALARMCK
;
        .MACRO  OVRFLW  ?QUIT
        CMP     #60.,R0
        BGT     QUIT
        CLR     R0
        INC     R1
        CMP     #60.,R1
        BGT     QUIT
        CLR     R1
        INC     R2
        CMP     #24.,R2
        BGT     QUIT
        CLR     R2
QUIT:   NOP
.ENDM   OVRFLW
;
;ОСНОВНАЯ ПРОГРАММА
START:  .WRITE  "\\ВВЕДИТЕ_ОСНОВАНИЕ_И_ФОРМАТ
        .RAD
SETIME: .WRITE  "\\ВВЕДИТЕ_ТЕКУЩЕЕ_ВРЕМЯ_(ЧАС__МИН__СЕК):__
        .READ   R2,R1,R0
        .WRITE  "\\ТЕПЕРЬ_ВРЕМЯ_ТАКОЕ:__,R2,"_:,R1,"_:,R0
SETALRM: .WRITE "\\УСТАНОВИТЕ_ВРЕМЯ_ЗВОНКА_БУДИЛЬНИКА_(ЧАС__МИН__СЕК):__
        .READ   ALARMH,ALARMM,ALARMS
        .WRITE  "\\ВРЕМЯ_ЗВОНКА_БУДИЛЬНИКА:_,ALARMH,"_:,ALARMM,"_:,ALARMS
        .WRITE  "\\УСТАНОВИТЕ_ИНТЕРВАЛ_ЗАДЕРЖКИ_В_СЕКУНДАХ:__
        .REEAD  INTRVL
        MOV     INTRVL,R3
TICK:   SEC
        INC     R0
        OVRFLW
        ALARMCK ALARMH,ALARMM,ALARMS,BELL
```

```
CONT:   DEC     R3
        BEQ     DISPLAY
        JMP     TICK
DISPLAY: .WRITE "\\ТЕПЕРЬ_ВРЕМЯ_ТАКОЕ:__,R2,"_:,R1,"_:,R0
        MOV     INTRVL,R3
        JMP     TICK
INTRVL: .BLKW
ALARMH: .BLKW
ALARMM: .BLKW
ALARMS: .BLKW
        .END    START
```

Рис. 8.16. Исходный текст программы для имитации будильника

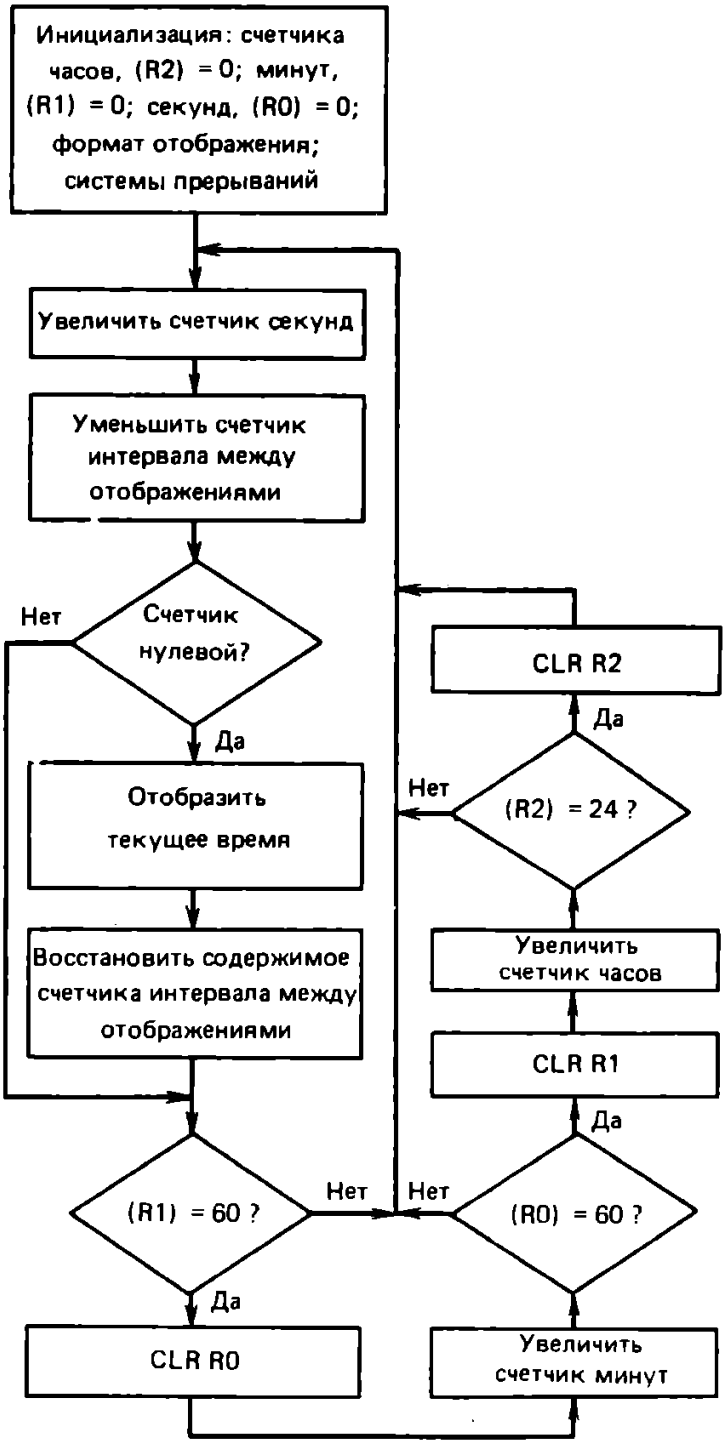


Рис. 8.17, *а*. Блок-схема программы для имитации секундомера

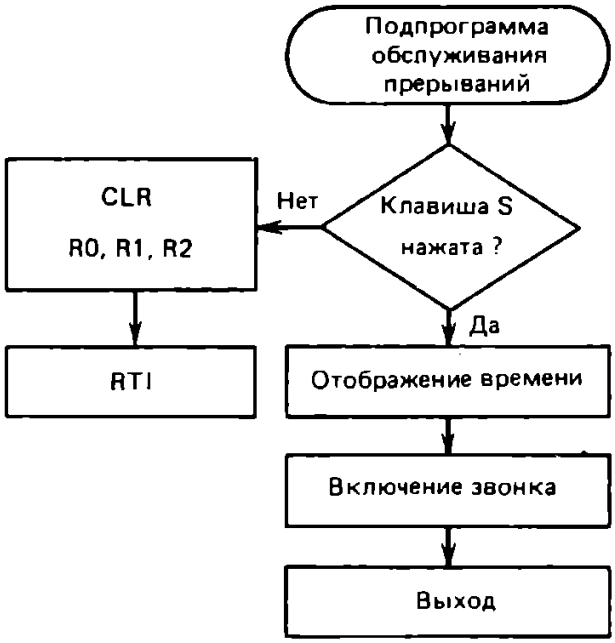


Рис. 8.17, *б*

```
        .TITLE  WATCH
        .MCALL  .EXIT
;ЭТО ПРОГРАММА СЕКУНДОМЕРА, ИСПОЛЬЗУЮЩАЯ ПРЕРЫВАНИЯ,
;НО НЕ ИСПОЛЬЗУЮЩАЯ СИСТЕМНЫХ МАКРОСОВ (ЗА ИСКЛЮЧЕНИЕМ .EXIT)
.MACRO  ASCIOUT R,TEMP1,TEMP2,?CONV,?OK
        MOV     R5,-(SP)        ;ГЕНЕРИРУЕТ ДВУХРАЗРЯДНОЕ ДЕСЯТИЧНОЕ
        MOV     R,TEMP1         ;ЧИСЛО ИЗ ДВОИЧНОГО ЧИСЛА В R
        CLR     TEMP2           ;ДЛЯ ВЫВОДА ДВЕ ЦИФРЫ
CONV:   CMP     TEMP1,#10.      ;ПОМЕЩАЮТСЯ В TEMP2 И TEMP1
        BLT     OK
        SUB     #10.,TEMP1
        INC     TEMP2
        BR      CONV
OK:     ADD     #60,TEMP1
        ADD     #60,TEMP2
        CRTOUT  TEMP2
        CRTOUT  TEMP1
        MOV     (SP)+,R5
.ENDM   ASCIOUT
TEMP1:  .BLKW
TEMP2:  .BLKW
;
.MACRO  SEC ?L1                 ;ЗАДЕРЖКА НА 1 С
        MOV     R5,-(SP)
        CLR     R5
L1:     INC     R5
        TST     R5
        BNE     L1
        MOV     (SP)+,R5
.ENDM   SEC
;
.MACRO  KEYIN   R
        MOVB    177562,R
.ENDM   KEYIN
;
.MACRO  CRTOUT R,?L2
L2:     TSTB    177564
        BPL     L2
        MOVB    R,177566
.ENDM   CRTOUT
;
.MACRO  DISPLAY INFO,?L4
        MOV     R5,-(SP)        ;ОТОБРАЗИТЬ СООБЩЕНИЕ
        MOV     #INFO,R5
L4:     CRTOUT  (R5)+
        TSTB    (R5)
        BNE     L4
        MOV     (SP)+,R5
.ENDM   DISPLAY
;
.MACRO  SHOWTIME        ;ПОКАЗАТЬ ВРЕМЯ ИЗ РЕГИСТРОВ R2,R1,R0
        CRTOUT  CR
        ASCIOUT R2,TEMP1,TEMP2
        CRTOUT  BL
        CRTOUT  BL
        ASCIOUT R1,TEMP1,TEMP2
        CRTOUT  BL
        CRTOUT  BL
        ASCIOUT R0,TEMP1,TEMP2
.ENDM   SHOWTIME
CR:     .WORD   15      ;КОД ASCII ВОЗВРАТА КАРЕТКИ
LF:     .WORD   12      ;КОД ASCII ПЕРЕВОДА СТРОКИ
BELL:   .WORD   7       ;КОД ASCII ЗВОНКА
```

```
BL:     .WORD   40      ;КОД ASCII ПРОБЕЛА
        .EVEN
TRMARK: .ASCIZ  /СЕКУНДОМЕР/
LINE:   .ASCIZ  /----------/
;
;ОСНОВНАЯ ПРОГРАММА
START:  CLR     R0      ;ИНИЦИАЛИЗАЦИЯ
        CLR     R1
        CLR     R2
        CLR     R4
        CRTOUT  LF
        DISPLAY LINE
        CRTOUT  CR
        CRTOUT  LF
        DISPLAY TRMARK
        CRTOUT  CR
        CRTOUT  LF
        DISPLAY LINE
        CRTOUT  CR
        CRTOUT  LF
        MOVB    #0,177560
        SHOWTIME                ;ПОКАЗАТЬ ПЕРВОНАЧАЛЬНОЕ ВРЕМЯ
WAIT1:  TSTB    177560          ;ОЖИДАНИЕ СТАРТОВОГО СИГНАЛА
        BPL     WAIT1
        MOVB    177562,R4
        CMP     #122,R4
        BNE     WAIT1
        CRTOUT  BELL
        MTPS    #0              ;УСТАНОВИТЬ ВЕКТОР ПРЕРЫВАНИЯ
        MOVB    #100,177560
        MOV     #INTRPT,60
        MOV     #140,62
TIME:   NOP                     ;ИМИТИРОВАТЬ ТАЙМЕР
        SEC
        INC     R0
CHK60:  CMP     #60.,R0
        BEQ     J2
        JMP     SHOW
J2:     CLR     R0
        INC     R1
        CMP     #60.,R1
        BEQ     J3
        JMP     SHOW
J3:     CLR     R1
        INC     R2
        CMP     #24.,R2
        BNE     SHOW
        CLR     R2
SHOW:   SHOWTIME
        JMP     TIME
;ПОДПРОГРАММА ОБРАБОТКИ ПРЕРЫВАНИЙ
INTRPT: KEYIN   R4
        MTPS    #340
        CMPB    #123,R4         ;ПРОВЕРКА НА "S" (СТОП)
        BEQ     STOP
        RTI
STOP:   CRTOUT  BELL
        SHOWTIME
WAIT2:  TSTB    177560
        BPL     WAIT2
        MOVB    177560,R4
        CMP     #103,R4         ;ПРОВЕРКА НА "C" (СБРОС)
        BEQ     CLEAR
```

```
        CMP     #122,R4         ;ПРОВЕРКА НА "R" (РЕСТАРТ)
        BNE     WAIT2
        CRTOUT  BELL
RET:    RTI
CLEAR:  CLR     R0
        CLR     R1
        CLR     R2
        CRTOUT  BELL
        SHOWTIME
        JMP     WAIT2
        .END    START
```

РЕЗУЛЬТАТ ВЫПОЛНЕНИЯ:

```
СЕКУНДОМЕР
00   00   00

КОГДА НАЖИМАЕТСЯ "R", ЧАСЫ НАЧИНАЮТ ИДТИ
КОГДА НАЖИМАЕТСЯ "S", ЧАСЫ ОСТАНАВЛИВАЮТСЯ
КОГДА НАЖИМАЕТСЯ "C", ЧАСЫ СБРАСЫВАЮТСЯ
```

Рис. 8.17, *в*. Исходная программа для имитации секундомера

Разработаем сначала макросы, определенные пользователем:

Макрос для формирования односекундного периода времени

```
.MACRO  SEC ?L1
        MOV     R5,-(SP)        ;СОХРАНИТЬ СТАРОЕ (R5)
        CLR     R5
L1:     INC     R5
        CMP     #0,R5
        BNE     L1              ;ОДНОСЕКУНДНЫЙ ЦИКЛ
        MOV     (SP)+,R5        ;ВОССТАНОВИТЬ СТАРОЕ (R5)
.ENDM,  SEC
```

Макрос для приема одного символа с клавиатуры

```
.MACRO  KEYBRDIN R,?L2
L2:     TSTB    177560
        BPL     L2
        MOVB    177562,R
.ENDM   KEYBRDIN
```

Макрос для вывода одного символа на экран ЭЛТ

```
.MACRO  CRTOUT R,?L3
L3:     TSTB    177564
        BPL     L3
        MOVB    R,177566
.ENDM   CRTOUT
```

Макрос для преобразования 6-битового двоичного числа в восьмеричное число из двух цифр в коде ASCII

```
.MACRO  ASCIOUT R,TEMP1,TEMP2,?L5
        MOV     R5,-(SP)
        MOV     #3,R5           ;R5 БУДЕТ СЧЕТЧИКОМ ДЛЯ СДВИГА
        MOV     R,TEMP1         ;TEMP1 - РАБОЧАЯ ЯЧЕЙКА
        BIC     #177770,TEMP1   ;ЗАМАСКИРОВАТЬ СОДЕРЖИМОЕ TEMP1
                                ;КРОМЕ ТРЕХ МЛАДШИХ БИТОВ
        ADD     #60,TEMP1       ;ПРЕОБРАЗОВАТЬ В КОД ASCII
        MOV     R,TEMP2
L5:     ASR     TEMP2           ;ВЫДВИНУТЬ ТРИ МЛАДШИХ БИТА
        DEC     R5
```

```
        BNE     L5
        BIC     #177770,TEMP2
        ADD     #60,TEMP2
        CRTOUT  TEMP2           ;ВЫВЕСТИ НА ДИСПЛЕЙ КОД ASCII
        CRTOUT  TEMP1           ;С ПОМОЩЬЮ МАКРОСА CRTOUT
        MOV     (SP)+,R5
.ENDM   ASCIOUT
TEMP1:  .BLKW   1
TEMP2:  .BLKW   1
```

Обратите внимание, что в этом макросе для использования в качестве рабочих зарезервировано две ячейки памяти. Необходимость резервирования области памяти вне макроса является, конечно, недостатком. Как альтернативу можно предложить "позаимствовать" регистры (как это было сделано с регистром R5). То есть мы можем "позаимствовать" регистры R4 и R3, сохранив (R4) и (R3) следующим образом:

MOV R4, –(SP)
MOV R3, –(SP),

и использовать регистр R3 вместо строчки TEMP1 и регистр R4 вместо строчки TEMP2. Конечно, в конце макроса мы должны восстановить из стека (R4) и (R3):

MOV (SP)+, R3
MOV (SP)+, R4

Макрос для отображения сообщения при использовании директивы .ASCIZ

```
.MACRO  DISPLAY INFO,?L4
        MOV     R5,-(SP)
        MOV     #INFO,R5
L4:     CRTOUT  (R5)+
        TSTB    (R5)
        BNE     L4
        MOV     (SP)+,R5
.ENDM   DISPLAY
```

Макрос для отображения текущего времени

```
.MACRO  SHOWTIME
        CRTOUT  CR              ;ОСУЩЕСТВИТЬ ПРОГРАММНО
                                ;ВОЗВРАТ КАРЕТКИ
        ASCIOUT R2,TEMP1,TEMP2
        CRTOUT  BL              ;ВЫВЕСТИ ПРОБЕЛ
        CRTOUT  BL
        ASCIOUT R1,TEMP1,TEMP2
        CRTOUT  BL
        CRTOUT  BL
        ASCIOUT R0,TEMP1,TEMP2
.ENDM   SHOWTIME
```

Если текущее время, заданное в регистрах R2, R1 и R0, будет равно соответственно 6, 20 и 6, то этот макрос отобразит на ЭЛТ в восьмеричном представлении следующее: 06 24 06.

Определив необходимые макросы, мы переходим к написанию программы. Чтобы продемонстрировать макросы ввода-вывода более полно, усложним эту программу чуть больше, чем требуется для обычного секундомера. Мы хотим, чтобы на экране ЭЛТ отображалось следующее:

```
---------------

СЕКУНДОМЕР

---------------

XX  XX  XX
(ч) (мин) (с)
```

И хотим, чтобы отображение менялось каждые 3 с. При нажатии на клавиатуре клавиши S часы остановятся, будет показано время, когда они остановились, и включен звонок терминала. При нажатии на любую другую клавишу часы сбрасываются в нуль и начинают счет заново. На рис. 8.17, *в* показана исходная программа, реализующая описанную выше спецификацию секундомера.

8.6. ЛОВУШКИ

До сих пор мы рассматривали систему с прерываниями, в которой источниками прерываний являлись внешние устройства. Инициализировав процесс прерывания, мы уже не думали об аппаратной логике прерывания. Центральный процессор может ожидать запроса на прерывание в любой момент, но точное время запроса он знать не может. Этот класс прерываний известен как внешние или аппаратные прерывания, когда идентификация источника прерывания или адрес уникального вектора прерываний обеспечивается аппаратурой ввода-вывода через шину данных.

В системе PDP-11 существует возможность прерываний другого типа – система, кроме аппаратных прерываний, обеспечивает возможность **ловушек**, когда при возникновении аномальной ситуации, такой как недопустимая инструкция, сброс питающего напряжения или обращение к несуществующей ячейке памяти, система попадает в ловушку через **вектор ловушки** с уникальным адресом. Подобно вектору прерываний вектор ловушки состоит из двух слов памяти, содержимым которых являются соответственно стартовый адрес подпрограммы обслуживания ловушки и содержимое PSW для этой подпрограммы. Подпрограмма обслуживания ловушки может содержать всего одну инструкцию, такую как HALT, или последовательность инструкций для вывода сообщения об ошибке для информации пользователя о событии, происшедшем при выполнении программы. Поэтому процесс перехода к ловушке рассматривается как внутреннее или программное прерывание.

Когда происходит переход к ловушке, ЦП (аппаратно) проталкивает в стековую память содержимое слова состояния PSW для главной программы и скорректированное содержимое регистра PC и помещает к регистру PC стартовый адрес подпрограммы обслуживания ловушки, а в слово состояния PSW – ранг приоритета прерывания. В конце подпрограммы обслуживания ловушки инструкция RTI выталкивает сохраненную в стеке информацию назад, что обеспечивает возврат к главной программе.

В противоположность процессу внешнего прерывания адрес вектора ловушки для каждого внутреннего прерывания заранее предопределен, так что пользователь не может изменить его по своему желанию. В системе PDP-11 определены адреса вектора ловушки для различных событий:

Адрес вектора ловушки	Описание
00 00 00	(Зарезервирован)
00 00 04	Ошибки ЦП
00 00 10	Недействительная или зарезервированная инструкция
00 00 14	Ловушка точки разрыва (BPT) или трассировки (бит T = 1 в PSW)
00 00 20	Ловушка ввода-вывода (IOT)
00 00 24	Сброс питания
00 00 30	Ловушка имитатора (EMT)
00 00 34	Инструкция TRAP

Среди указанных пользователю доступны ловушки, соответствующие инструкциям BPT, IOT, EMT и TRAP. Если пользователь хочет разрабатывать или использовать свои собственные подпрограммы обслуживания ловушек, на инструкции EMT и TRAP следует обратить особое внимание. Дело в том, что в векторах 30 и 34 находятся не стартовые адреса подпрограмм обслуживания ловушки, а стартовые адреса подпрограмм

специального назначения, называемых **TRAP-манипуляторами.** Через такие манипуляторы вызываются специализированные подпрограммы обслуживания ловушки.

У инструкций EMT и TRAP имеется операнд, представляющий собой целое число. Ниже показан мнемонический формат этих инструкций.

EMT *n*. В этом случае *n* – целое число, занимающее место младшего байта 16-битового слова ($n = 0, \ldots, 377$). При выполнении этой инструкции в системе происходит прерывание или переход к ловушке, и аппаратурой ЦП выполняется следующая последовательность действий:

текущее содержимое PSW → стек
скорректированное содержимое регистра PC → стек
(30) → PC
(32) → PSW

Но содержимое ячейки памяти 30 не является стартовым адресом подпрограммы обслуживания прерывания, а является стартовым адресом подпрограммы обслуживания, определенной как манипулятор ловушки, с помощью которой система осуществляет переход к *n*-й подпрограмме обслуживания прерываний с последующим возвратом к главной программе с помощью инструкции RTI или RTT.

Инструкция EMT используется обычно в системном программном обеспечении, что позволяет разработать до 377_8 системных подпрограмм обслуживания, и пользователям не рекомендуется ее применять. Для общих целей предназначена инструкция TRAP.

TRAP *n*. Как и в инструкции EMT, *n* – целое число, занимающее младший байт 16-битового слова инструкции ($n = 0, \ldots, 377$). При выполнении этой инструкции аппаратура ЦП осуществляет следующие шаги:

текущее содержимое PSW → стек
скорректированное содержимое регистра PC → стек
(34) → PC
(36) → PSW

Опять содержимое ячейки 34 не определяет подпрограмму обслуживания прерывания, а является стартовым адресом манипулятора ловушки. О написании манипулятора ловушки и инициализации стартового адреса в ячейке 34 и приоритетного ранга в ячейке 36 должен заботиться программист – пользователь системы. Манипулятор ловушки должен выполнять следующие функции:

1. Сохранять и восстанавливать все регистры общего назначения – R0, R1, . . . , R5.
2. Интерпретировать операнд инструкции *n*.
3. Вызывать конкретную, или *n*-ю подпрограмму обслуживания прерываний.
4. Если необходимо, передавать нужные параметры подпрограмме обслуживания прерываний.
5. Осуществлять возврат к главной программе.

Следующие примеры помогут прояснить этот процесс.

МАНИПУЛЯТОР ЛОВУШКИ

Поскольку манипулятор ловушки отвечает за интерпретацию инструкции TRAP и вызов желаемой *n*-й подпрограммы обслуживания прерываний, то он должен иметь программную секцию для интерпретации операнда этой инструкции. К несчастью, к тому моменту, когда ЦП начинает выполнять манипулятор ловушки, сама инструкция TRAP *n* уже "пройдена". То есть информации об операнде *n* уже нет в ЦП. Манипулятору нужно отыскать какой-то способ восстановления этой информации. К счастью, мы знаем, что в стек было протолкнуто скорректированное содержимое регистра PC, а содержимое регистра SP было уменьшено на 2.

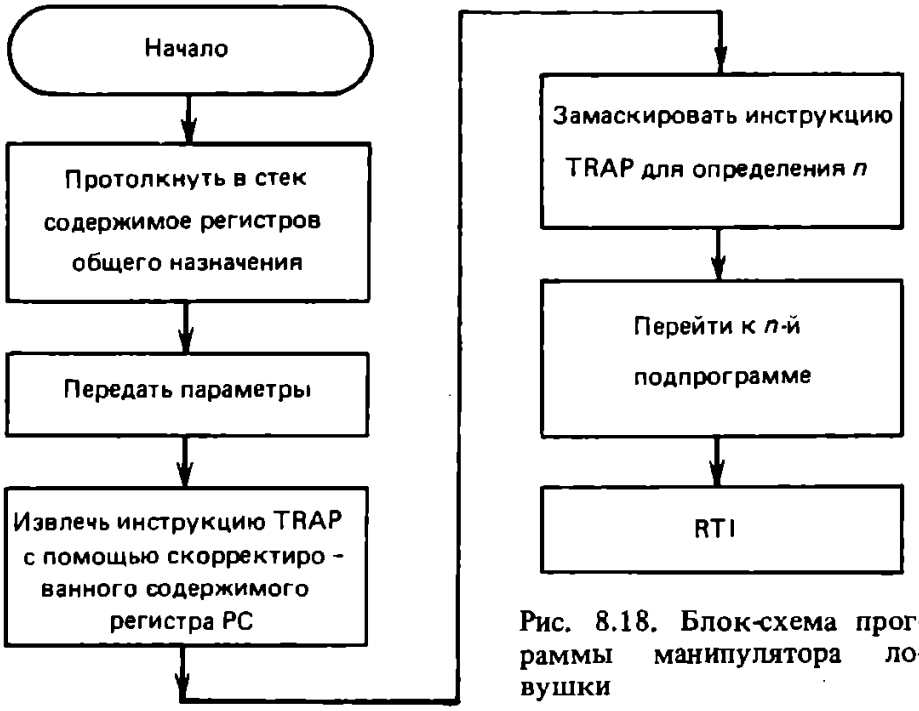


Рис. 8.18. Блок-схема программы манипулятора ловушки

Поэтому мы в состоянии найти ячейку, в которой находится инструкция TRAP *n*, путем уменьшения скорректированного содержимого регистра PC на 2 и извлечь число *n*. Кроме этого в манипуляторе должен быть каталог, в котором перечислены все стартовые адреса используемых подпрограмм обслуживания. На рис. 8.18 показана блок-схема программы манипулятора ловушки. Соответствующая исходная программа:

```
THDLR:  MOV     R0,-(SP)        ;СОХРАНИТЬ РЕГИСТРЫ
        MOV     R1,-(SP)        ;ОБЩЕГО НАЗНАЧЕНИЯ (РОН)
        MOV     R2,-(SP)
        MOV     R3,-(SP)
        MOV     R4,-(SP)
        MOV     R5,-(SP)
        MOV     14(SP),R0       ;ИЗВЛЕЧЬ СКОРРЕКТИРОВАННОЕ (PC)
        MOV     -(R0),R0        ;ИЗВЛЕЧЬ ИНСТРУКЦИЮ TRAP
        BIC     #177400,R0      ;ИЗВЛЕЧЬ N, МАСКИРУЯ R0
        ASL     R0              ;2 * (R0) -> R0
        JMP     @SRVDIR(R0)     ;ПЕРЕЙТИ К ПОДПРОГРАММЕ
                                ;ОБСЛУЖИВАНИЯ ЧЕРЕЗ КАТАЛОГ
RETURN: MOV     (SP)+,R5
        MOV     (SP)+,R4
        MOV     (SP)+,R3
        MOV     (SP)+,R2
        MOV     (SP)+,R1
        MOV     (SP)+,R0
        RTI
SRVDIR: .WORD   SRVR0,...,SRV377
        .END    THDLR
```

ТИПИЧНЫЙ ПРИМЕР ИСПОЛЬЗОВАНИЯ ИНСТРУКЦИИ TRAP

Следующая программа — типичный пример использования инструкции TRAP:

```
START:  MOV     #THDLR,34       ;УСТАНОВИТЬ ВЕКТОР ЛОВУШКИ
        MOV     #200,36         ;УСТ-ТЬ ПРИОРИТЕТНЫЙ РАНГ 4
        TRAP    2               ;ПРОГРАММНОЕ ПРЕРЫВАНИЕ
        HALT
MESSG:  .ASCIZ  /ПРИВЕТ/
        .EVEN
THDLR:  MOV     R0,-(SP)        ;ДЛЯ ПРОСТОТЫ ПОЛОЖИМ, ЧТО
```

```
        MOV     R1,-(SP)        ;СОХРАНИТЬ НУЖНО ТОЛЬКО R0 И R1
        MOV     4(SP),R0        ;СКОРРЕКТИРОВАННОЕ (PC) -> R0
        MOV     -(R0),R0        ;КОД ИНСТРУКЦИИ TRAP 2 -> R0
        BIC     #177400,R0      ;ПОЛУЧИТЬ 2, МАСКИРУЯ R0
        ASL     R0              ;2 * 2 = 4 -> R0
        JMP     @SRVDIR(R0)
RETURN: MOV     (SP)+,R1        ;ВОССТАНОВИТЬ (R1) ИЗ СТЕКА
        MOV     (SP)+,R0        ;ВОССТАНОВИТЬ (R0) ИЗ СТЕКА
        RTI
SRVDIR: .WORD   SRVR0,SRVR1,SRVR2
SRVR0:  JMP     RETURN          ;ФИКТИВНАЯ СЕРВИСНАЯ ПОДПРОГРАММА
SRVR1:  JMP     RETURN          ;ФИКТИВНАЯ СЕРВИСНАЯ ПОДПРОГРАММА
SRVR2:  MOV     #MESSG,R1       ;R1 - УКАЗАТЕЛЬ СООБЩЕНИЯ
L1:     TSTB    177564
        BPL     L1
        MOVB    (R1)+,177566
        TSTB    (R1)
        BNE     L1
        JMP     RETURN
```

Файл листинга этой программы показан на рис. 8.19.

КОММЕНТАРИИ ПО ПОВОДУ ИНСТРУКЦИИ TRAP

Мы уже отмечали, что поскольку в системе PDP-11 инструкция EMT обычно используется системными программистами, она не рекомендуется для общего применения пользователями. Но если мы хотим воспользоваться *n*-й подпрограммой системного сервиса, созданной системным программистом, то для доступа к ней можем воспользоваться инструкцией EMT *n*. Таким образом гарантируется защита системного программного обеспечения, поскольку пользователь может войти в систему только таким способом. Для разработки своих программ мы можем ограничиться инструкцией TRAP.

Обратите внимание, что инструкция TRAP по своим функциям похожа на инструкцию JSR R, SUBR, однако есть и некоторые различия. Первая инструкция занимает только одно слово памяти и вызывает прерывание, обладающее свойством приоритетности. Кроме того, подпрограммы обслуживания определяются аргументом $n (n = 0, \ldots, 377)$. Инструкция JSR занимает два слова памяти, ее адрес назначения — это, как правило, символьная метка, и вызванному ею прерыванию свойство приоритетности не присуще.

8.7. ВЛОЖЕНИЕ ПРЕРЫВАНИЙ И РАЗРЕШЕНИЕ ПРИОРИТЕТОВ

Аналогично макросам и подпрограммам подпрограммы обслуживания прерываний могут вкладываться в другие подпрограммы обслуживания прерываний до тех пор, пока они удовлетворяют условиям приоритета. Вложение прерываний показано на рис. 8.20. Здесь подпрограмма прерывания *n* должна иметь более высокий приоритет, чем подпрограмма прерывания $(n-1)$. Разрешение приоритетов прерываний обычно реализуется аппаратно. Типичные аппаратные реализации показаны на рис. 8.21.

На рис. 8.21, *а* представлена схема простого цепочечно-группового метода. Линии запроса прерывания здесь собираются логически по операции ИЛИ. Когда одно или несколько устройств выставляют сигнал запроса прерывания, ЦП отвечает на это сигналом подтверждения прерывания. Сигнал подтверждения последовательно проходит через все устройства, но устройство 1 получает сигнал первым. Если устройство 1 не запрашивало прерывания, оно передает сигнал подтверждения дальше, к следующему устройству и т. д. Если устройство 1 запрашивало прерывание, оно не будет передавать сигнал подтверждения к следующему устройству, а перешлет к ЦП по шине данных свой идентификационный номер, представляющий собой не что иное, как адрес век-

```
 1                                          ;ТИПИЧНЫЙ ПРИМЕР ИСПОЛЬЗОВАНИЯ ИНСТРУКЦИИ TRAP
 2                                          ;
 3 000000  012767  000020' 000034' START:   MOV     #THDLR,34       ;УСТАНОВИТЬ ВЕКТОР ЛОВУШКИ
 4 000006  012767  000200  000036'          MOV     #200,36         ;УСТАНОВИТЬ ПРИОРИТЕТНЫЙ РАНГ 4
 5 000014  104402                           TRAP    2               ;ПРОГРАММНОЕ ПРЕРЫВАНИЕ
 6 000016  000000                           HALT
 7                                          ;
 8 000020  010046                THDLR:     MOV     R0,-(SP)        ;ДЛЯ ПРОСТОТЫ ПОЛОЖИМ, ЧТО
 9 000022  010146                           MOV     R1,-(SP)        ;СОХРАНИТЬ НУЖНО ТОЛЬКО (R0) И (R1)
10 000024  016600  000004                   MOV     4(SP),R0        ;СКОРРЕКТИРОВАННОЕ (PC) --> R0
11 000030  014000                           MOV     -(R0),R0        ;КОД ИНСТРУКЦИИ TRAP 2 --> R0
12 000032  042700  177400                   BIC     #177400,R0      ;ПОЛУЧИТЬ 2, МАСКИРУЯ R0
13 000036  006300                           ASL     R0              ;2 * 2 = 4 --> R0
14 000040  000170  000052'                  JMP     @SRVDIR(R0)     ;ПЕРЕХОД К СООТВЕТСТВУЮЩЕЙ ПОДПРОГРАММЕ
15 000044  012601                RETURN:    MOV     (SP)+,R1        ;ВОССТАНОВИТЬ (R1) ИЗ СТЕКА
16 000046  012600                           MOV     (SP)+,R0        ;ВОССТАНОВИТЬ (R0) ИЗ СТЕКА
17 000050  000002                           RTI
18                                          ;
19 000052  000060' 000064' 000070' SRVDIR:  .WORD   SRVR0,SRVR1,SRVR2
20 000060  000167  177760          SRVR0:   JMP     RETURN          ;ФИКТИВНАЯ СЕРВИСНАЯ ПОДПРОГРАММА
21 000064  000167  177754          SRVR1:   JMP     RETURN
22 000070  012701  000116'         SRVR2:   MOV     #MESSG,R1       ;R1 - УКАЗАТЕЛЬ СООБЩЕНИЯ
23 000074  105767  177564'         L1:      TSTB    177564
24 000100  100375                           BPL     L1
25 000102  112167  177566'                  MOVB    (R1)+,177566
26 000106  105711                           TSTB    (R1)
27 000110  001371                           BNE     L1
28 000112  000167  177726                   JMP     RETURN
29 000116     160     162     151 MESSG:    .ASCIZ  /ПРИВЕТ/
   000121     167     145     164
   000124     000
30                                          .EVEN
31         000000'                          .END    START
```

Рис. 8.19. Файл листинга для примера использования инструкции TRAP

тора прерываний. Это приведет к выполнению соответствующего процесса прерывания. Заметим, что при такой конфигурации устройство 1 имеет наивысший приоритет.

На рис. 8.21, *б* показана другая типичная схема разрешения приоритетов прерываний, которая может представлять собой интерфейсную плату или микросхему. Основной принцип работы почти такой же, как у схемы на рис. 8.21, *а*, за исключением того, что текущий приоритетный статус ЦП пересылается в схему разрешения приоритетов

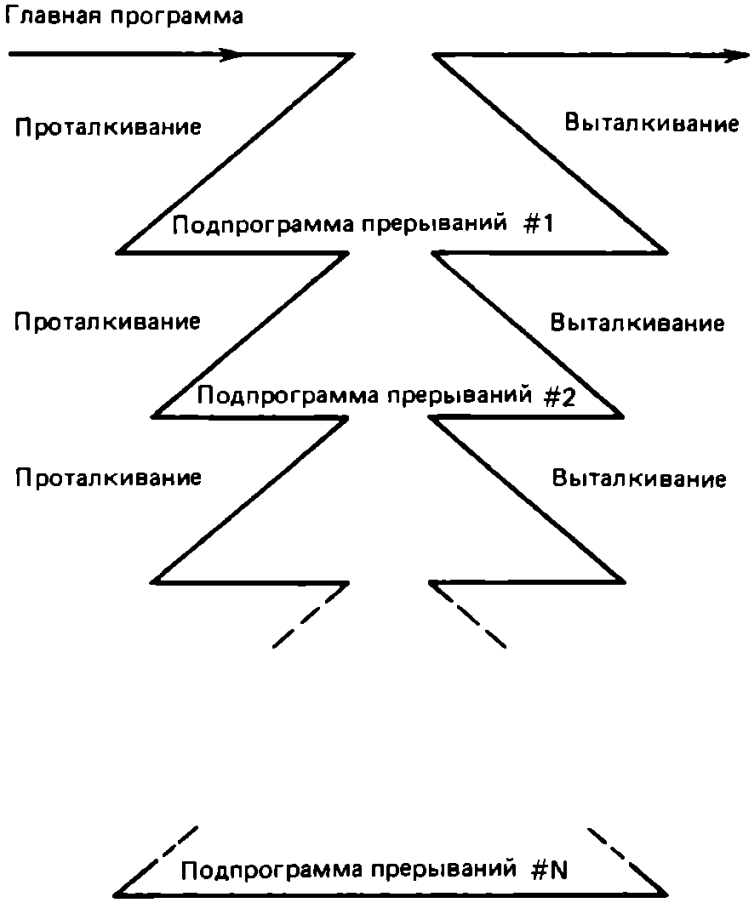
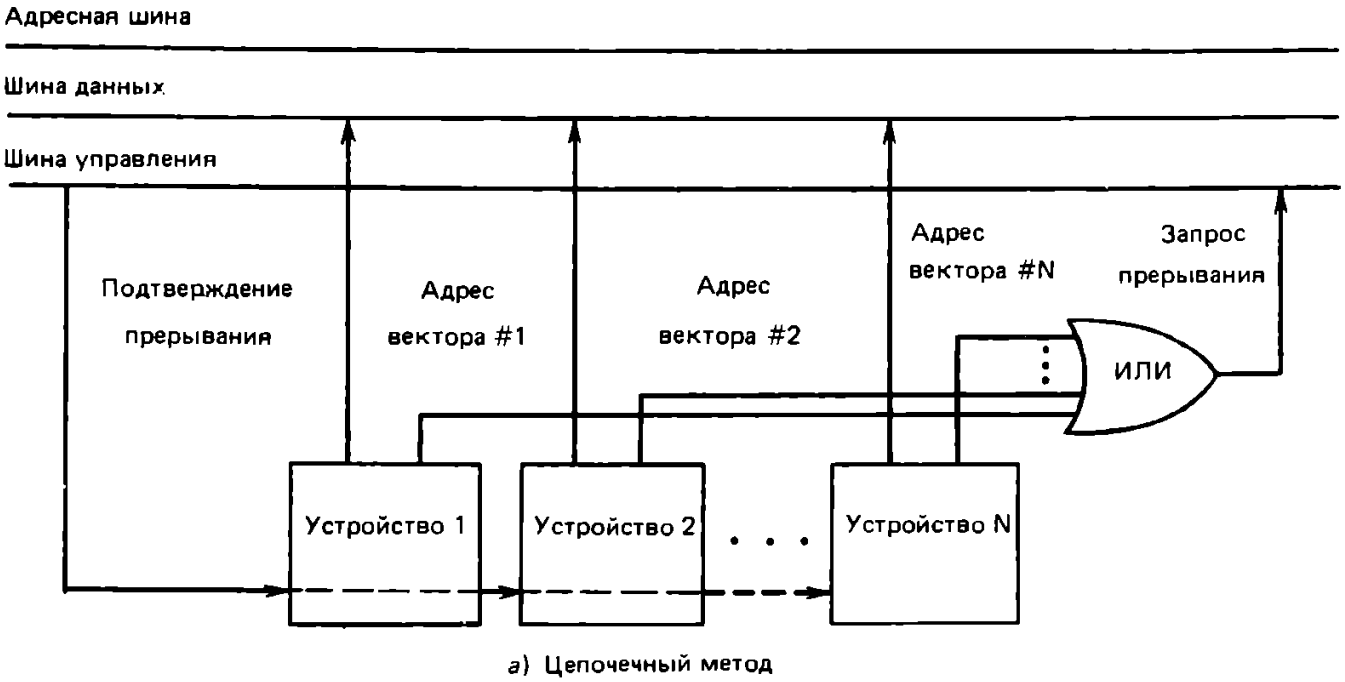
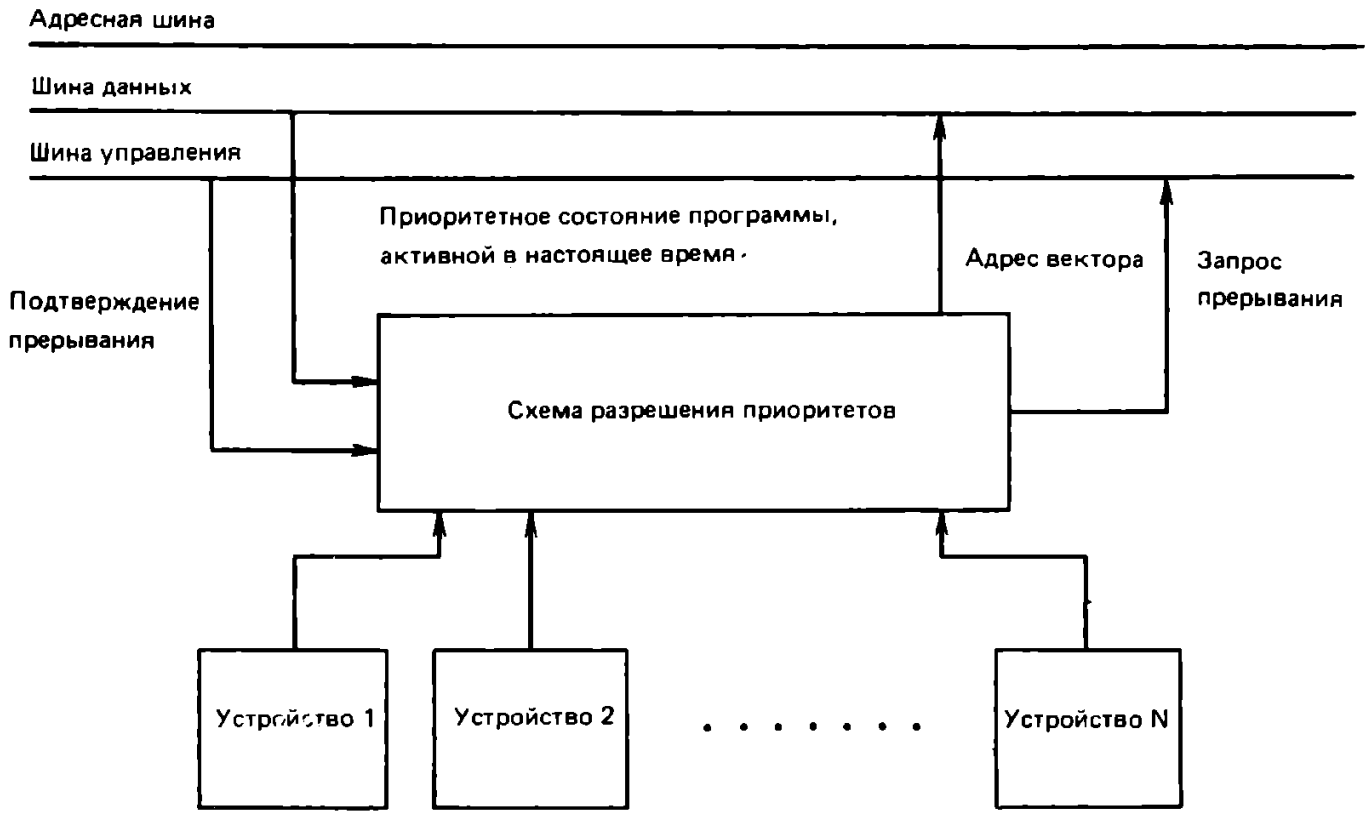


Рис. 8.20. Вложение прерываний



а) Цепочечный метод

Рис. 8.21, *а*. Типичная схема разрешения приоритетов прерываний



б) Микросхема или плата разрешения приоритетов

Рис. 8.21, *б*. Типичная схема разрешения приоритетов прерываний

и затем сравнивается с приоритетными статусами устройств, запросивших прерывание. Любое устройство с наивысшим приоритетом останавливает эту операцию и посылает свой адрес вектора прерываний. Кроме того, в этом случае пользователь может управлять схемой разрешения приоритетов прерываний с помощью набора инструкций, распознаваемых аппаратно, поэтому такая схема является программируемой и, следовательно, обеспечивает гибкость.

8.8. УПРАЖНЕНИЯ

1. Кратко обсудите различие между методами опроса и прерывания для устройств ввода-вывода.
2. Объясните, почему во многих случаях важно, чтобы в системе, работающей с прерываниями, были средства для сохранения и восстановления содержимого регистров общего назначения.
3. Используя метод прерывания, напишите программу, которая будет отображать на ЭЛТ сообщение "Я МОГУ БЫТЬ ВАМ ПОЛЕЗНОЙ" всякий раз, когда на клавиатуре будет нажиматься любая клавиша. Вы можете положить адресную информацию порта ввода-вывода терминала следующей:

   Ввод: регистр управления и состояния содержит 177560; буферный регистр данных – 177562.

   Вывод: регистр управления и состояния содержит 177564; буферный регистр данных – 177566;

   Вектор прерываний: 60.

4. Разработайте полную, но простую автоматизированную систему управления делопроизводством бухгалтерии на базе ЭВМ PDP-11. Бухгалтер обычно работает на терминале с ЭЛТ, подключенном к системе, и обрабатывает счета клиентов. Очевидно, что содержимое регистров общего назначения важно в любой момент времени. Главному бухгалтеру нужен другой терминал, подключенный к другому порту ввода-вывода системы, чтобы он имел возможность прерывать работу бухгалтера и требовать обслуживания со стороны системы. Предполагается, что им потребуется "переговариваться" через терминалы. Вы должны спланировать, какого типа программное

обеспечение должно быть создано, а также определить адресную информацию для обоих терминалов. Разработайте сначала подробную блок-схему, затем напишите соответствующую программу на языке ассемблера. Всегда полезно, если вы разбиваете свою программу на модули или подпрограммы.

ГЛАВА 9
ПРЯМОЙ ДОСТУП В ПАМЯТЬ (ПДП)

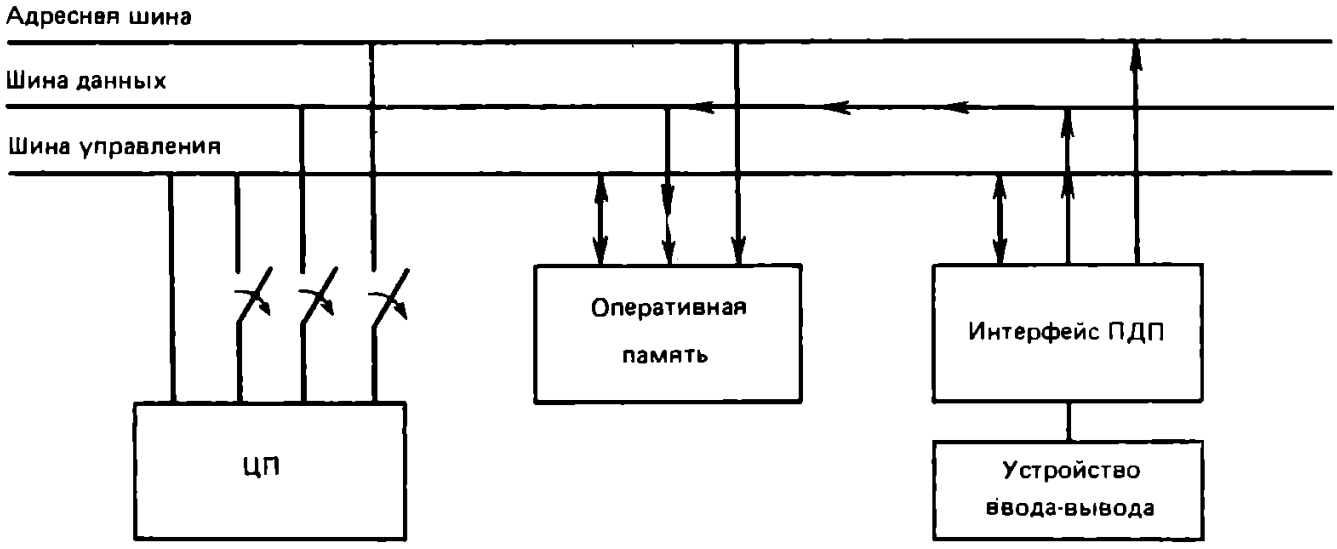
9.1. ВВЕДЕНИЕ

До сих пор мы описывали аппаратуру и программное обеспечение, необходимые для программирования интерфейсов ввода-вывода и процессов прерывания. Обратите внимание, что все эти процессы или операции осуществляются с участием ЦП. Именно ЦП управляет "рукопожатием" и транспортировкой данных; и в ЦП по счетчику команд PC пересылаются те адреса, которые подлежат чтению или записи в памяти. Очевидно, что ЦП является в системе главным; в любое время он управляет шинами. Но на практике нам часто нужно лишь собирать данные и записывать их в память без какой-либо обработки. Аналогично во многих случаях нам хочется лишь выводить данные из памяти на периферийные устройства в том виде, в каком они находятся в памяти. Очевидно, что вовлечение ЦП в операции такого типа приводит к напрасной трате его "способностей" или "времени", поскольку никаких манипуляций данными не требуется.

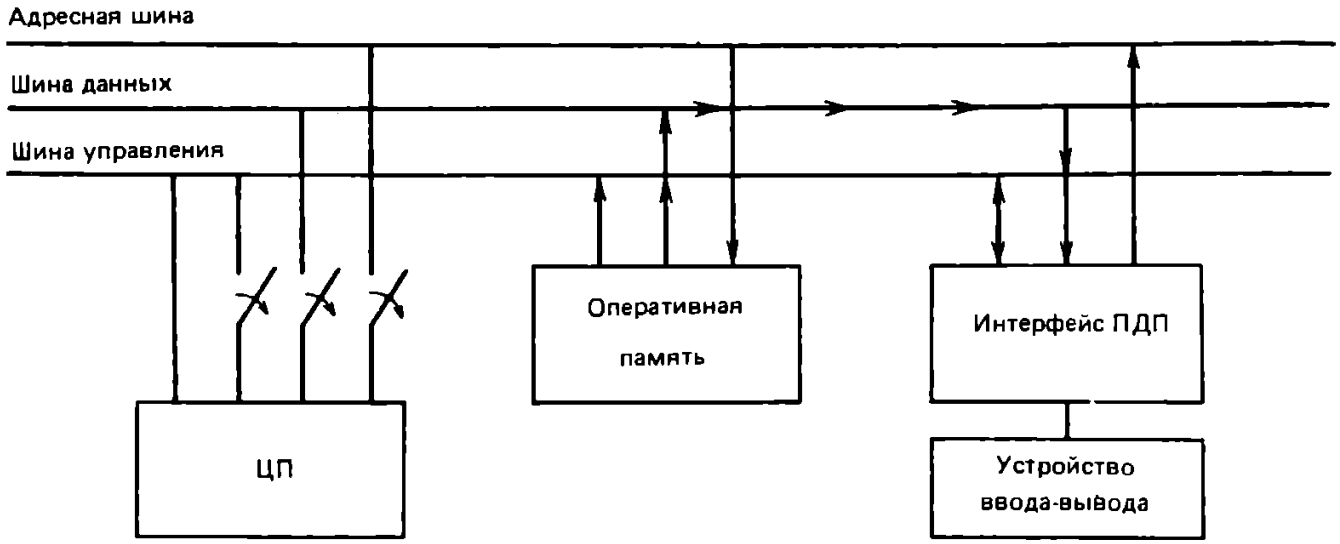
Чтобы улучшить эффективность использования ЦП и увеличить скорость транспортировки данных непосредственно в память от устройств ввода-вывода и обратно, желательно располагать некоторым методом, позволяющим устранить вовлечение ЦП в транспортировки такого типа. Именно это осуществляется с помощью метода **прямого доступа в память (ПДП)**. Ясно, что при пересылке одного или двух элементов данных между устройством ввода-вывода и памятью существенной экономии машинного времени получить невозможно. Но если мы хотим пересылать блок данных, то с помощью метода ПДП можем сэкономить довольно много машинного времени. Поэтому мы будем рассматривать применение метода ПДП, только если необходимо транспортировать данные в виде блоков. В противном случае будем использовать обычную пересылку данных ввода-вывода.

Как и при любом другом процессе ввода-вывода, для выполнения ПДП необходим определенный аппаратно-программный интерфейс. Чтобы устранить вторжение ЦП в процесс ПДП, нам нужно определить, какую функцию или функции, выполняемые ЦП, нам надо или не надо имитировать. Очевидно, что нам необходим регистр, эквивалентный регистру PC, для определения того, в каком месте памяти мы хотим осуществлять транспортировку данных, и сигналы чтения и записи для задания направления транспортировки: в память или из памяти. Затем мы должны определить, сколько слов в блоке данных мы хотим транспортировать. Следовательно, нам необходим какой-то счетчик, позволяющий выяснить, все ли данные блока уже транспортированы. Кроме того, мы должны располагать механизмом для выполнения "рукопожатия", чтобы интерфейсная плата ПДП могла "выставить" ЦП из операционного цикла в период транспортировки данных и вернуть управление назад ЦП после завершения задания.

Процесс ПДП изображен на рис. 9.1. Сначала ЦП инициализирует интерфейсную плату ПДП, а когда данные готовы к транспортировке, интерфейсная плата посылает сигнал запроса ПДП к ЦП. ЦП отвечает сигналом согласия с запросом ПДП и отсоединяется от шин. В результате плата ПДП становится главной на шинах и выполняет задачу транспортировки данных. В зависимости от конкретной конфигурации плата



а) ПДП для ввода данных в память



б) ПДП для вывода данных из памяти

Рис. 9.1. Функциональная схема прямого доступа в память

ПДП может удержать управление шинами в течение одного или нескольких циклов транспортировки данных за одну передачу. Однако после завершения транспортировки блока данных плата ПДП инициирует сигнал запроса прерывания, чтобы проинформировать ЦП о завершении передачи, и ЦП может возобновить управление шинами. На рис. 9.1, *а* и *б* показаны соответственно транспортировка данных в оперативную память и из нее.

9.2. АППАРАТНАЯ ОРГАНИЗАЦИЯ И ПРИНЦИПЫ РАБОТЫ

Давайте теперь рассмотрим аппаратную конфигурацию базовой интерфейсной платы ПДП. В общем случае она состоит из пяти регистров плюс идентификационный номер прерывания (вектор прерывания):

1. *Регистр счетчика слов (РСС).* Этот регистр позволяет выяснить, полностью ли уже переслан целый блок данных.

2. *Регистр адреса на шине (РАШ).* Этот регистр функционирует как регистр РС в ЦП. Обычно он инициализируется ЦП значением стартового адреса блока памяти, из которого или в который должна производиться транспортировка данных.

3. *Регистр управления и состояния (РУС).* Так же как и на любой другой плате ввода-вывода, этот регистр служит хранилищем команд к этой плате и статусной информации от нее. Он имеет стандартное определение бит:

b_7 – бит готовности или завершения;
b_6 – бит разрешения прерываний;
b_4, b_5 – биты расширения адреса;
b_1, b_2, b_3 – биты выбора функции;
b_0 – бит разрешения транспортировки по ПДП.
Кроме того, используются такие типичные определения:
b_{15} – бит ошибки;
b_{14} – бит обращения к несуществующей памяти;
b_{13} – бит внимания;
b_{12} – эксплуатационный бит;
b_{11}, b_{10}, b_9 – биты состояния устройства;
b_8 – цикл подготовки шинного цикла ПДП.

4. *Буферный регистр данных входа (БРДВХ).* Этот регистр сохраняет данные, которые должны быть прочитаны в память.

5. *Буферный регистр данных выхода (БРДВЫХ).* Этот регистр принимает данные из памяти.

6. *Регистр установки адреса вектора прерываний.* Этот регистр (на рис. 9.2 он не показан) обеспечивает идентификационный номер платы для процессов прерывания. Его содержимое может устанавливаться вручную с помощью 8-битового двухрядного переключателя.

Рассмотрим теперь операцию ПДП по порядку. Обратимся к рис. 9.2. Программисту следует инициализировать плату, для чего необходимо: 1) загрузить регистр счетчика слов значением, равным числу подлежащих транспортировке слов, но взятым с отрицательным знаком и представленным в виде дополнения до двух; 2) загрузить регистр адреса на шине начальным адресом блока памяти, в который или из которого будут пересылаться данные; 3) загрузить регистр управления и состояния соответствующим битовым образом.

Битовый образ регистра РУС у различных ЭВМ может различаться, но принцип действия остается одним и тем же. Существуют биты, которые функционируют как управляющие или командные биты для платы, и есть биты, которые показывают состояние платы или подключенного к ней устройства ввода-вывода. Первые обычно программируются и могут по желанию устанавливаться или сбрасываться с помощью инструкций. Вторые, как правило, устанавливаются или сбрасываются аппаратной логикой на плате, показывающей состояние устройства ввода-вывода. Как и у регистра РУС обычной платы ввода-вывода, бит 0 служит для разрешения работы устройства, он активизирует процесс ПДП, бит b_6 является битом разрешения прерываний, а бит b_7 – битом завершения или готовности. Бит b_7 в данном случае применяется для индикации того, что содержимое счетчика слов РСС стало равным нулю; если бит b_6 установлен в логическую единицу, то переход бита b_7 в логическую единицу вызывает прерывание. Поэтому инициализирующая часть программы должна сбрасывать бит b_7 в нуль и устанавливать бит b_6 в единицу.

После инициализации интерфейсная плата ПДП препринимает следующие шаги для выполнения операции транспортировки данных: 1) выдает сигнал запроса ПДП, когда данные в буферном регистре готовы к транспортировке; 2) становится главной на шине, поскольку ЦП в этом месте от шины отсоединяется. Затем выполняется процесс транспортировки данных. При этом РАШ функционирует как РАП для памяти, и в зависимости от конкретного режима работы будет происходить или пересылка

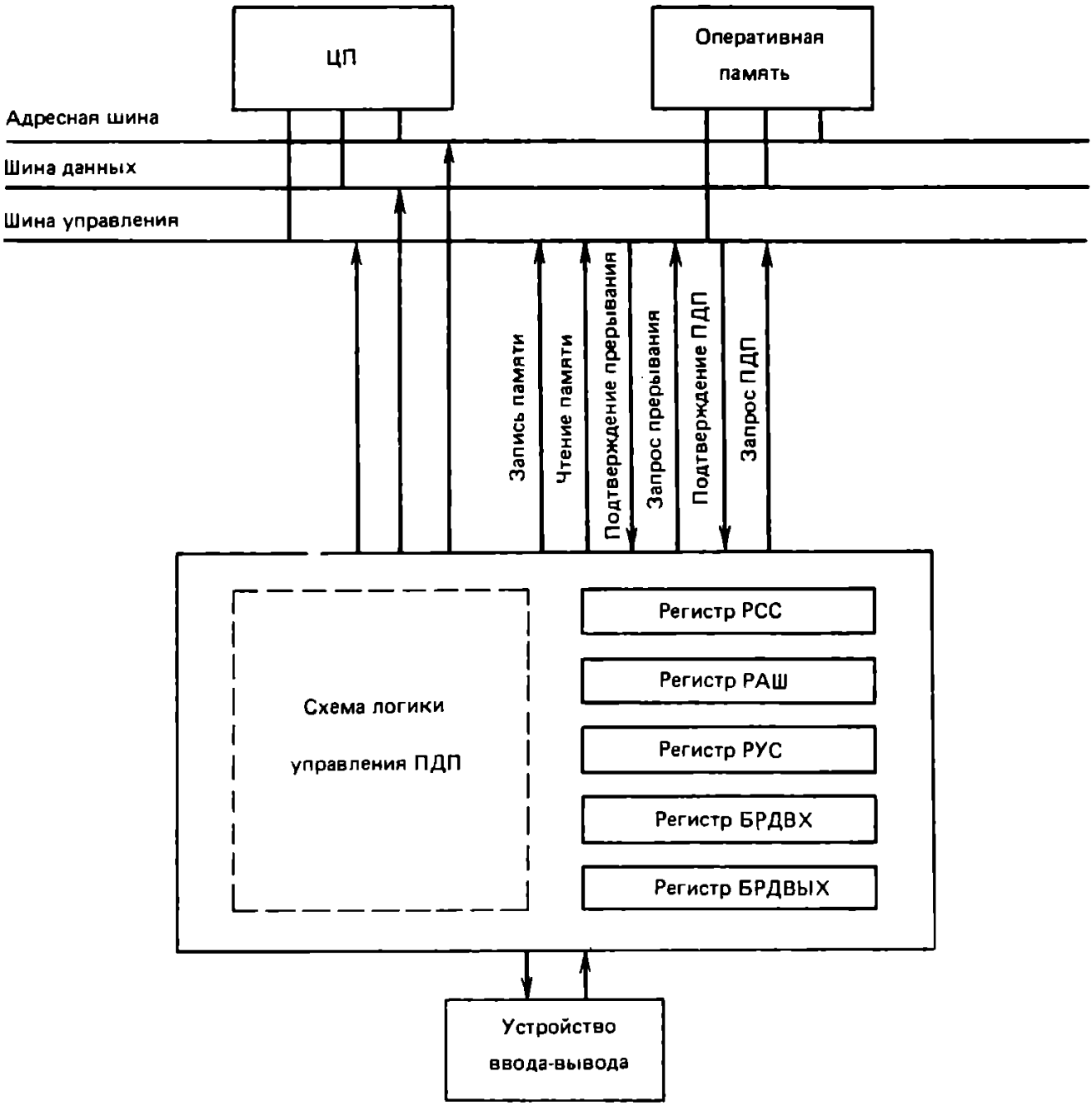


Рис. 9.2. Схема работы прямого доступа в память

(БРДВХ) → @ (РАШ), т. е. содержимое БРДВХ будет копироваться в ячейку памяти, указанную регистром РАШ, или @ (РАШ) → (БРДВЫХ).

После этого плата ПДП возвращает управление ЦП, уменьшает регистр РСС и увеличивает регистр РАШ. Этот цикл повторяется до тех пор, пока содержимое регистра РСС не станет равным нулю. В этот момент устанавливается бит готовности или завершения, в результате чего инициируется процесс прерывания и адрес вектора платы помещается на шину данных. Затем ЦП переходит к подпрограмме обслуживания прерываний и выполняет ее.

Ответственность за написание подпрограммы обслуживания прерываний таким образом, чтобы она удовлетворяла конкретному приложению, лежит на программисте. Обратите внимание, что плата выдает запрос прерывания путем установки в единицу бита b_8 или бита b_{15}. В первом случае подразумевается, что завершился процесс транспортировки данных; второй случай является индикацией того, что в схеме на плате допущена ошибка. Поэтому важно, чтобы подпрограмма обслуживания прерываний

могла распознавать, какое прерывание произошло: обычное или по ошибке, для чего ей надо сначала проверять бит b_{15}, а затем предпринимать соответствующие действия.

Метод транспортировки данных с использованием ПДП известен как метод "кражи циклов". Поскольку управление шиной плата ПДП получает только на время транспортировки данных, то считается, что происходит "кража" цикла у ЦП. Если такая "кража" будет синхронизирована с циклами извлечения и выполнения инструкций ЦП, то процесс ПДП не окажет влияния на производительность системы, поскольку во время цикла выполнения ЦП, как правило, управлять внешней шиной не требуется.

9.3. ТИПИЧНЫЙ ПРИМЕР

Предположим, что нам надо собрать набор из 200 элементов данных, полученных от аналого-цифрового преобразователя (АЦП) с помощью метода ПДП. Упрощенная блок-схема системы показана на рис. 9.3. Система действует следующим образом:

1. Бит 0 регистра РУС, установленный ЦП, разрешает работу местного тактового генератора, который порождает импульсную последовательность, управляющую преобразованием.

2. Импульс завершения преобразования от АЦП сигнализирует, что цифровые данные готовы в БРДВХ, и приводит к тому, что плата генерирует сигнал запроса ПДП.

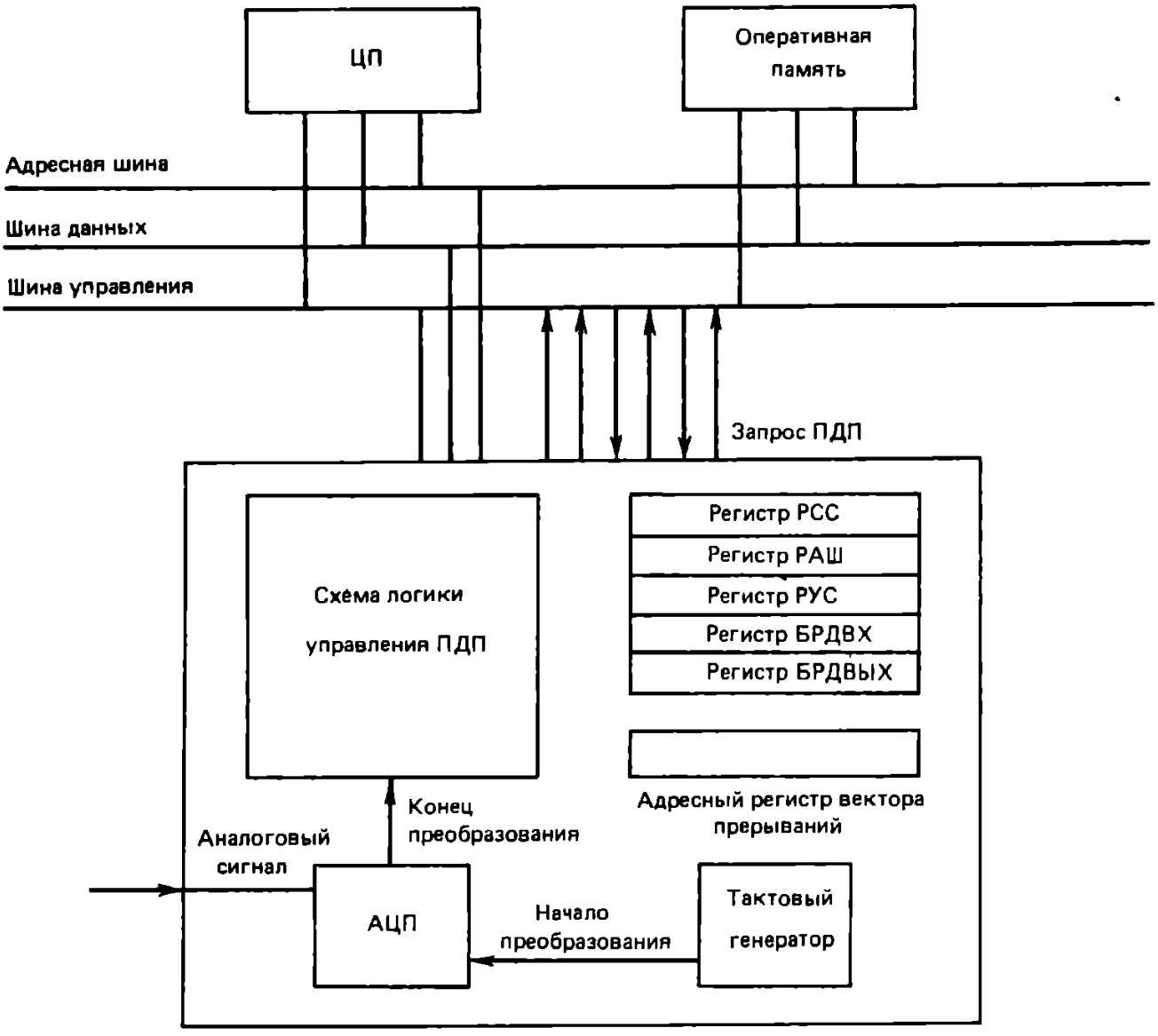


Рис. 9.3. Упрощенная блок-схема накопления данных от АЦП с помощью метода ПДП

3. Поступает сигнал согласия на прерывание ПДП, вызывающий транспортировку данных из БРДВХ в память со стартовой адресной меткой ADCDATA.

4. Увеличивается содержимое регистра РАШ.

5. Увеличивается содержимое регистра РСС, чтобы уменьшить счет слов, поскольку этот регистр был загружен отрицательным значением полного числа элементов данных в виде дополнения до двух. В этом примере загружаемое в этот регистр значение равно -200 или 177600_8.

6. Схема на плате проверяет, не произошло ли ошибки, т. е. выполняется ли условие (РСС) = 0. Если выполняется, то устанавливается бит b_7 в регистре РУС и генерируется сигнал запроса прерывания. Если же произошла ошибка, то вызывается подпрограмма обслуживания прерывания по ошибке.

7. После получения сигнала согласия на прерывание плата помещает адрес вектора прерываний на шину данных.

8. ЦП переходит к подпрограмме обслуживания прерываний, которая проверяет биты b_{15} и b_7.

9. ЦП возобновляет выполнение главной программы.

Исходная программа для этой последовательности действий:

```
.MACRO  CRTOUT R,?L1
L1:     TSTB    177564
        BPL     L1
        MOVB    R,177566
.ENDM   CRTOUT
.MACRO  DISPLAY INFO,?L2
        MOV     R5,-(SP)
        MOV     #INFO,R5
L2:     CRTOUT  (R5)+
        TSTB    (R5)
        BNE     L2
.ENDM   DISPLAY
START:  .                       ;ГЛАВНАЯ ПРОГРАММА
        .
        .
        MOV     #INTRPT,VECTOR
        MOV     #ADCDATA,BAR    ;ИНИЦИАЛИЗИРОВАТЬ BAR
        MOV     #177600,WCR     ;ЗАГРУЗИТЬ -200 В WCR
        MOV     #101,CSR        ;РАЗРЕШИТЬ ПРЕРЫВАНИЯ
                                ;И НАЧАЛО ПРЕОБРАЗОВАНИЯ.
        .                       ;ВЫПОЛНЯЕТСЯ ГЛАВНАЯ ПРОГРАММА,
        .                       ;ПОКА НЕ ПРОИЗОЙДЕТ ПРЕРЫВАНИЕ.
        .                       ;В ЭТО ЖЕ ВРЕМЯ КОНТРОЛЛЕР ПДП
        .                       ;"КРАДЕТ" ЦИКЛЫ ДЛЯ ПЕРЕСЫЛКИ ДАННЫХ
        .                       ;МЕЖДУ ПАМЯТЬЮ И ПЛАТОЙ ПДП
        .
ADCDATA: .BLKW  200             ;БЛОК ПАМЯТИ ПОД ДАННЫЕ
INTRPT: TST     CSR             ;ПОДПРОГРАММА ОБРАБОТКИ ПРЕРЫВАНИЙ
                                ;НАЧИНАЕТ С ПРОВЕРКИ ТОГО, ЧТО БИТ 15 РАВЕН 1
        BPL     RETURN          ;БИТ 7 РАВЕН 1, ДАННЫЕ СОБРАНЫ НОРМАЛЬНО
        DISPLAY MESSG           ;БИТ 15 РАВЕН 1, БЫЛА ОШИБКА.
                                ;ВЫВЕСТИ СООБЩЕНИЕ ОБ ОШИБКЕ
        HALT
MESSG:  .ASCIZ  /ОШИБКА ПРИ ПДП/
RETURN: RTI                     ;ВОЗВРАТ К ГЛАВНОЙ ПРОГРАММЕ
        .END    START
```

Обратите внимание на то, что процесс транспортировки данных по ПДП в главной программе не показан. Управление шинами в процессе ПДП осуществляется платой ПДП, она же выполняет и реальную транспортировку данных. То есть аппаратура платы ПДП выдает сигнал запроса ПДП всякий раз, когда генерируется импульс завершения

преобразования АЦП. В результате ЦП "отсоединяет себя" от шины, т. е. временно отдает управление шиной, и выдает сигнал согласия на прерывание по ПДП в плату ПДП.

Получая сигнал согласия на прерывание, плата ПДП берет на себя управление шиной и пересылает элемент данных из ячейки памяти, определенной регистром РАШ, или в нее, увеличивает содержимое регистра РСС на 1 и возвращает управление ЦП. Этот процесс повторяется до тех пор, пока не будет выполнено условие (РСС) = 0, после чего плата ПДП инициирует процесс прерывания, чтобы сигнализировать процессору об окончании процесса ПДП. В этот момент ЦП отвечает на запрос прерывания от платы ПДП и переходит к подпрограмме обслуживания прерываний.

9.4. УПРАЖНЕНИЯ

1. Кратко объясните преимущества и недостатки (если они есть) вычислительной системы с возможностями ПДП.
2. Предположите, что цифровая вычислительная система работает в следующей последовательности:
   а) извлекает инструкцию из памяти;
   б) проверяет, не имеет ли сигнал запроса ПДП логического значения "истина";
   в) если имеет, ЦП посылает сигнал подтверждения ПДП и передает право занять ведущее положение на шине тому устройству, которое запросило операцию ПДП; иначе происходит переход к п. г);
   г) ЦП выполняет инструкцию;
   д) ЦП опять проверяет, не имеет ли сигнал запроса ПДП все еще значения "истина";
   е) если да, ЦП будет работать на холостом ходу до тех пор, пока сигнал запроса ПДП не примет логическое значение "ложь";
   ж) система возвращается к шагу а).

Теперь нарисуйте два графика – один для операции прямого асинхронного ввода-вывода (не ПДП), второй – для операции ПДП в двумерном изображении, когда ось X представляет число элементов данных, пересланных из памяти или в нее (MAX = 100), а ось Y представляет скорость выполнения в микросекундах, необходимую для транспортировки определенного числа элементов данных. Вы можете положить, что время, требуемое для извлечения одного слова памяти, равно 0,5 мкс и время транспортировки одного слова по ПДП также равно 0,5 мкс. Следует помнить, что операция ПДП выполняется аппаратно платой ПДП, для чего могут потребоваться "накладные расходы" в 200 нс/слово.

ГЛАВА 10
ВВЕДЕНИЕ В СИСТЕМУ VAX-11

10.1. ВВЕДЕНИЕ

Предыдущие главы были посвящены рассмотрению структуры ЭВМ и программированию на языке ассемблера для ЭВМ PDP-11. Опыт показывает, что студентам, освоившим этот материал, легко работать и на других ЭВМ. Например, у наших студентов не возникало никаких трудностей при переходе к системе на основе микропроцессора 8085. Возможно, это было потому, что структура микропроцессора 8085 и набор его инструкций намного проще, чем у ЭВМ PDP-11. Для начала можно отметить, что 8085 является 8-битовой машиной, тогда как PDP-11 машина 16-битовая. Режимы адресации микропроцессора 8085 просты, у него нет таких мощных режимов, как режимы адресации с автоувеличением или автоуменьшением. Поэтому при использовании микропроцессора 8085 студенты и не испытывают трудностей, хотя чувствуют его ограничения по производительности. Можно считать, что разработка систем на базе 8085 вместо PDP-11 сопряжена лишь с примитивизацией доступных разработчику средств, и не ожидать больших трудностей. Чего, однако, нам следует ожидать, если придется иметь дело с более

развитой машиной? Очевидной ЭВМ, которую можно выбрать для этого случая, является VAX-11.

Далее мы будем иметь дело с этой мощной машиной. Концепции и методы, рассмотренные в предыдущих главах, применимы и к этой ЭВМ. Цель данной главы состоит в том, чтобы помочь читателю "плавно перейти" от использования ЭВМ PDP-11 к использованию ЭВМ VAX-11.

VAX – это первые буквы английских слов, означающих "виртуальное адресное расширение". Как и ЭВМ PDP-11, ЭВМ VAX-11 производится фирмой Digital Equpment Corporation. Это 32-битовая машина, которая может работать в двух режимах: естественном и совместимом. В совместимом режиме ЭВМ VAX-11 может выполнять все программы, разработанные для ЭВМ PDP-11, если их немного модифицировать; иначе машина работает со всеми своими возможностями в естественном режиме. Поскольку ЭВМ VAX-11 – 32-битовая машина, у нее возможно адресное пространство свыше 4 Гбайт. В естественном режиме ЭВМ VAX-11 может выполнять большой набор инструкций (включающий 248 основных инструкций) переменной длины. Некоторые инструкции могут иметь до шести **операндных спецификаторов**. Под операндным спецификатором мы понимаем ту часть инструкции, которая определяет адресную информацию для извлечения исходных операндов или операндов назначения. В ЭВМ PDP-11, например, инструкция MOV A, B имеет два операндных спецификатора – A и B. Помимо этого набор инструкций ЭВМ VAX-11 может манипулировать и транспортировать целый ряд различных типов данных, включая числа с плавающей точкой. При установленной операционной системе ЭВМ VAX-11 может быть мультипрограммной многопользовательской цифровой вычислительной системой реального времени. Краткое описание уникальных особенностей ЭВМ VAX-11, отличающих ее от ЭВМ PDP-11, дается в последующих разделах.

10.2. СТРУКТУРА ЭВМ

СИСТЕМА VAX-11

Типичная система VAX-11 показана на рис. 10.1. Обратите внимание, что у VAX-11 есть два шинных адаптера: адаптер общей шины и адаптер массовой шины. При такой конфигурации все совместимые с общей шиной периферийные устройства могут быть прямо подключены к общей шине, тогда как высокоскоростные устройства массовой памяти, такие как дисковые приводы или магнитофоны, могут подключаться к мас-

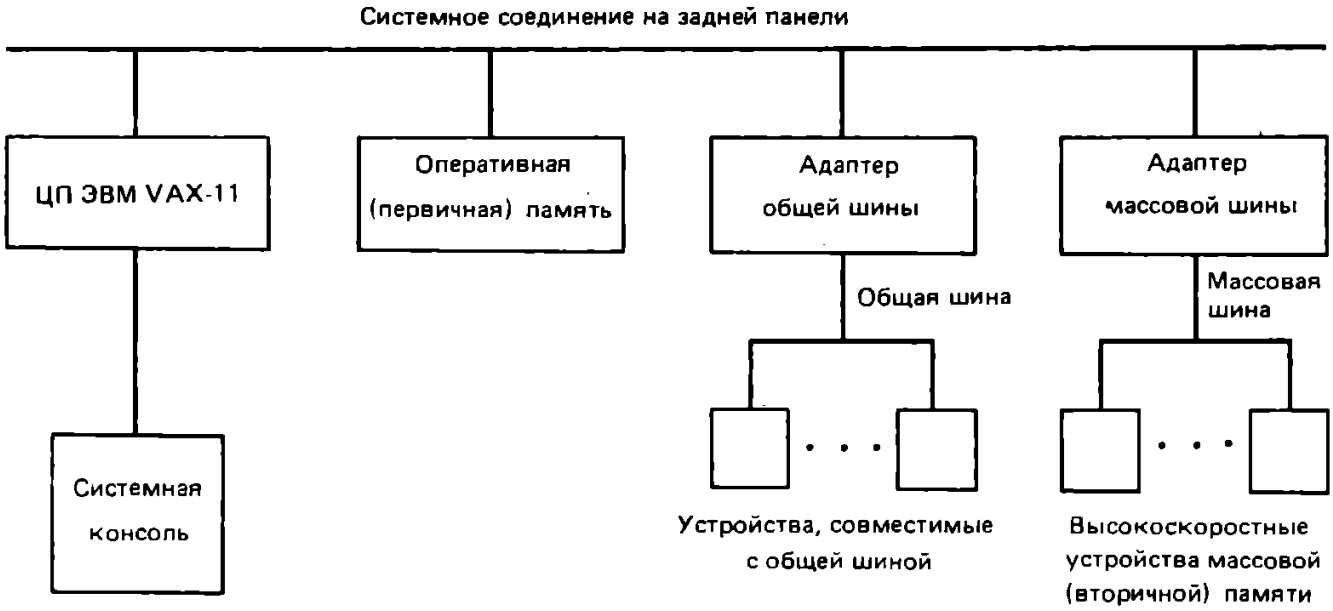


Рис. 10.1. Системная блок-схема типичной ЭВМ VAX-11

совой шине через собственные контроллеры. В функции шинных адаптеров входят: преобразование данных общей шины или массовой шины к формату, определяемому асинхронным соединителем на задней панели; связь и управление взаимодействием между ЦП, памятью и устройствами на общей шине и массовой шине. С помощью дисковой памяти (вторичной), работающей под воздействием устройства управления памятью совместно с полупроводниковой (первичной) памятью, в системе VAX-11 осуществляется механизм виртуальной памяти. Сравнивая структуру простой ЭВМ PDP-11 со структурой ЭВМ VAX-11, можно перечислить некоторые уникальные особенности последней. Прежде всего VAX-11 – 32-битовая машина. В ней применяется механизм виртуальной памяти. Наконец, это мультипрограммная система.

Система виртуальной памяти – это система, обеспечивающая пользователя гигантским адресным пространством памяти, называемым виртуальным адресным пространством, и при этом требующая адресного пространства физической памяти меньшего объема. Другими словами, при поддержке устройства управления памятью система виртуальной памяти с небольшим адресным пространством физической памяти может "питать" пользователей таким образом, что они смогут создавать программы так, как будто располагают адресным пространством, которое может быть адресовано регистром программного счетчика PC. Например, ЭВМ VAX-11/750 с физической памятью 8 Мбайт может предоставить пользователю два миллиарда байт адресного пространства для ассемблированных программ. Для удобства ссылок это гигантское адресное пространство (определяемое регистром PC) называют **виртуальным** или **логическим адресным пространством**, а действительное адресное пространство памяти называют **физическим адресным пространством**. Концепция **мультипрограммирования** означает, что с помощью разделения времени ЦП и разделения пространства физической памяти вычислительная система может выполнять целый набор программ в режиме, подобном мультиплексированию. Поскольку скорость работы современного ЦП измеряется порядком долей микросекунды на инструкцию, то пользователям будет казаться, что все программы выполняются одновременно.

ПРИНЦИП ВИРТУАЛЬНОЙ ПАМЯТИ

Функциональная блок-схема системы виртуальной памяти представлена на рис. 10.2. Для простоты положим, что это небольшая система, подобная PDP-11, в которой у нас есть только физическая или полупроводниковая память объемом 4 Кбайта, используемая как первичная, но функционально мы преобразовали ее, как это показано, в систему виртуальной памяти. Поскольку регистр PC (или R7) в системе PDP-11 имеет длину 16 бит, с его помощью можно адресовать 64 Кбайта адресного пространства памяти. Следовательно, в этой системе мы имеем виртуальное адресное пространство объемом 64 Кбайт и физическое адресное пространство объемом 4 Кбайта. Предположим, что физическая память организована так, что адресное пространство от адреса 0 до адреса 1777_8 зарезервировано для системного программного обеспечения, а адресное пространство от адреса 2000_8 до адреса 3777_8 – для программ пользователей, и

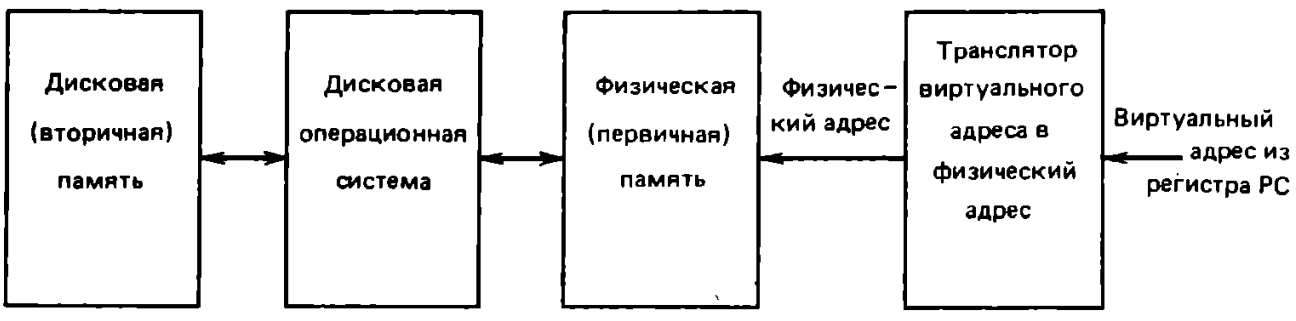


Рис. 10.2. Функциональная блок-схема системы виртуальной памяти

что на диске у нас есть пользовательская программа (в объектном коде), для которой требуется 6 Кбайт памяти.

Очевидно, что для всей программы имеющейся физической памяти недостаточно, поэтому нам остается надеяться лишь на то, что операционная система сделает что-нибудь "умное", чтобы справиться с этой проблемой. Операционная система может сегментировать программу пользователя на шесть неразрывных секций, например 0, 1, ... , 5, по 1000_8 байт каждая. Давайте также сегментируем область программы пользователя в физической памяти (от адреса 2000 до адреса 3777) на два неразрывных сегмента по 1000 байт каждый (положим для удобства, что мы используем восьмеричную числовую систему без указания индекса 8). Таким образом, и секция, и сегмент занимают 1000 байт. Если предположить, что в программе пользователя нет инструкции, которая осуществляла бы переход "дальше", чем на 1000 байт, то полная программа может быть выполнена в ограниченном пространстве физической памяти следующим образом.

Сначала операционная система загрузит секции 0 и 1 программы пользователя соответственно в сегменты 0 и 1 физической памяти. Для выполнения программы ЦП начнет выполнять инструкции, заданные с адреса 2000, и будет продолжать выполнение по ходу программы. Когда ЦП завершит выполнение сегмента 0 и приступит к сегменту 1, операционная система загрузит секцию 2 программы пользователя в сегмент 0. Поэтому, когда ЦП выполнит последнюю инструкцию сегмента 1, он сможет продолжить выполнение в сегменте 0 и т. д. Эта операция похожа на операцию в примере с буфером данных в виде очереди из гл. 8. Поскольку физическая память загружается последовательно секциями 0, 1, . . . , 5 программы пользователя, то в конце концов ЦП выполнит всю программу.

Попытаемся теперь понять необходимость блока преобразования адресов, показанного на рис. 10.2. Сначала надо исследовать механизм адресации системы. Вспоминаем, что объектный файл программы всегда начинается с нулевого адреса. Следовательно, объектный код программы пользователя в нашем примере должен начинаться с нуля и заканчиваться на адресе 5777. Здесь мы имеем виртуальное или логическое адресное пространство от 0 до 5777, сегментированное на шесть секций, но физическое адресное пространство при этом занимает адреса от 2000 до 3777. Как мы помним, первые две секции программы пользователя загружались в физическую память, начиная с адреса 2000. И здесь мы имеем взаимно однозначное соответствие между адресами виртуальной памяти и физическими адресами. То есть первоначально секции 0 и 1 оказываются в сегментах 0 и 1; следовательно:

0 (виртуальный) = 2000 (физический);

1777 (виртуальный) = 3777 (физический).

Однако при втором круге секция 2 оказывается в сегменте 0, а секция 1 все еще в сегменте 1; и для секции 2 мы имеем:

2000 (виртуальный) = 2000 (физический);

2777 (виртуальный) = 2777 (физический).

Обратите внимание, что преобразование адреса является динамическим, и поэтому, как показано на рис. 10.2, нужна схема динамического преобразования адресов. При наличии встроенного адресного преобразователя программа пользователя по-прежнему может ссылаться на виртуальные адреса, вовсе не зная о том, каким образом в действительности ЦП выполняет программу в ограниченном пространстве физической памяти. Периодическая "подкачка" пользовательской программы в физическую память и автоматическое преобразование адресов виртуальной памяти в адреса физической памяти – задача системного программного обеспечения, осуществляющего управление памятью. Более подробно с системой виртуальной памяти ЭВМ VAX-11 мы познакомимся в разделе по управлению памятью.

ПРИНЦИП МУЛЬТИПРОГРАММИРОВАНИЯ

Основная цель мультипрограммирования состоит в обеспечении эффективного использования ресурсов вычислительной системы. Очевидно, что среди пользовательских программ одни могут быть ориентированы на операции с числами (такие занимают ЦП в существенной степени); другие могут в основном иметь дело с вводом-выводом, а это означает, что большую часть времени ЦП будет работать на холостом ходу, ожидая ответа от периферийных устройств; иные программы могут быть короткими, тогда для них требуется немного физической памяти. В системе мультипрограммирования, пока идет обработка программ, ориентированных на ввод-вывод, ЦП может выполнять программу вычислительного характера. Кроме того, в физической (первичной) памяти в одно и то же время могут быть резидентными все или какая-то часть множества пользовательских программ, так что в любой момент времени незанятой памяти либо окажется очень мало, либо не будет совсем. Эффективность использования памяти, тем самым, увеличивается. Принцип мультипрограммирования изображен на рис. 10.3. Обратите внимание, что время ЦП сегментируется квантами равной длительности и таким образом осуществляется мультиплексирование времени для обслуживания программ 1 и 2. Поскольку скорость работы ЦП высока, то создается впечатление, что пользовательские программы обслуживаются одновременно.

Чтобы внести ясность и пояснить принцип мультипрограммирования, воспользуемся аналогией с ситуацией, когда несколько семей снимают частные квартиры в большом доме. Здание оборудовано плавательным байссейном, прачечной, источником минеральной воды, теннисными кортами и т. п., а каждая квартира состоит из одной или нескольких спален, столовой, кухни и ванной комнаты. Есть и контора, в которой располагается управляющий зданием. Мы можем вообразить себе, что семья, живущая в квартире, эквивалентна программе пользователя, а квартира – это физическая память для пользовательской программы; что управляющий – это операционная система, резидентная в физической памяти (конторе); что плавательный бассейн, теннисные корты и т. п. – это ЦП, периферийные устройства и т. д. и что семья состоит из родителей и детей, что эквивалентно пользовательским программам.

В семье родители обычно имеют наивысший приоритет при использовании имеющихся средств и ресурсов; аналогично некоторые подпрограммы в программах пользователей могут быть более привилегированными, чем другие. Кроме того, операционная система (управляющий) несет ответственность за планировку работы пользовательских программ (семей или членов семей, живущих в здании) с целью разделения времени имеющихся в системе ресурсов (ЦП, печатающих устройств и т. п.). Можно сказать, что в широком смысле принцип мультипрограммирования почти идентичен принципу, положенному в основу использования здания со сдаваемыми внаем квартирами.

Операционная система, реализующая принцип мультипрограммирования, осуществляет планировку назначения временных сегментов использования ЦП для выполнения

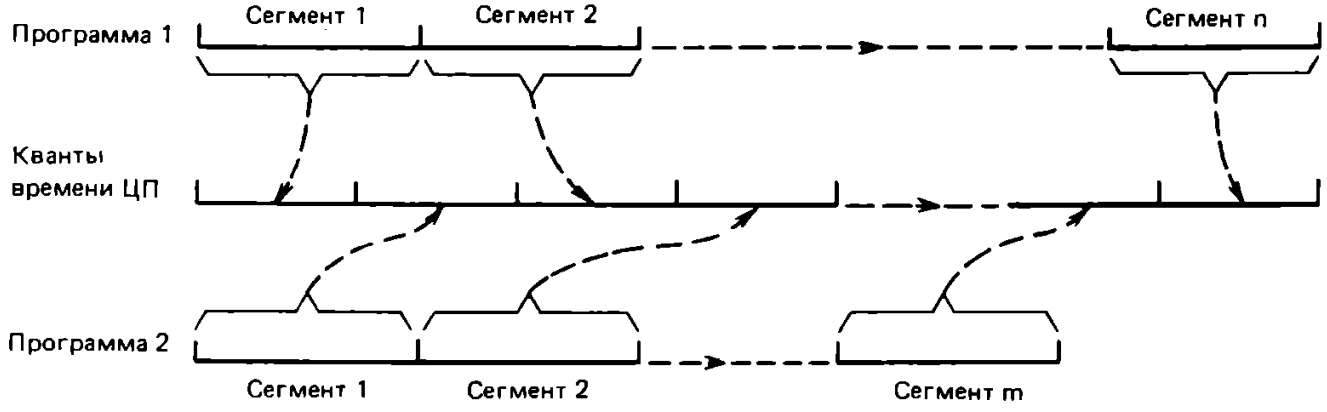


Рис. 10.3. Принцип мультипрограммирования

программ, а также планировку использования периферийных устройств с целью повышения эффективности использования системы. Предположим, например, что мы хотим выполнить набор пользовательских программ в системе мультипрограммирования. Во-первых, программы должны быть загружены с диска в оперативную память. Затем операционная система должна спланировать, как программы будут выполняться ЦП в режиме разделения времени. То есть каждая программа получит свой квант времени использования ЦП (рис. 10.3). Более короткие программы могут закончить выполнение за меньшее число квантов времени, чем более длинные. Поскольку скорость операций ЦП весьма велика, все выполняемые программы могут "думать", что только они используют ЦП. Это похоже на то, как управляющий зданием каждой семье планирует некоторый отрезок времени для пользования, например, минеральным источником, имеющимся в здании. Если спланированное время является в точности таким, какое хотелось бы иметь каждой семье, то члены каждой семьи будут чувствовать, что это их собственный источник.

Однако при исследовании разделения времени ЦП выявляется одна проблема: если мы недостаточно внимательны, может быть нарушена конфиденциальность программы со стороны другой программы. Например, содержимое регистров общего назначения для программы 1, показанной на рис. 10.3, может быть испорчено, если его не сохранить на время выполнения программы 2 в период законного владения ею ЦП. Чтобы предотвратить влияние одних программ на другие, каждая программа должна сохранять содержимое своих ключевых регистров, в том числе скорректированное содержимое регистра PC и содержимое слова состояния PSW, в конце каждого кванта времени выполнения и начинать каждый квант (когда она вновь получает ЦП в свое распоряжение) с восстановления содержимого этих регистров.

Вспоминаем, что в системе, использующей прерывания, которая была описана в предыдущей главе, содержимое слова состояния PSW и скорректированное содержимое регистра PC должны проталкиваться в стек, в то время как в ЦП должны быть перенесены стартовый адрес и слово состояния PSW для программы обслуживания прерываний, чтобы началось выполнение этой подпрограммы. Аналогично (но на более высоком уровне) в мультипрограммной системе каждая программа должна обладать собственной, "частной" областью памяти (похожей на область вектора прерывания) для сохранения содержимого своих ключевых регистров и другой важной информации.

В начале выделенного программе временного кванта сохраненное или инициализированное содержимое ключевых регистров, в том числе скорректированное содержимое регистра PC и слова состояния PSW для этой программы, должно быть перенесено в ЦП и загружено в соответствующие аппаратные регистры (PC, PSW, R0, R1 и т. д.) для выполнения программы. В конце временного кванта текущее содержимое ключевых регистров опять должно быть сохранено в этой специфической области памяти, зарезервированной для данной программы, до тех пор, пока опять не наступит очередь этой программы использовать ЦП. Содержимое ключевых регистров и другую важную информацию, сохраняемую в памяти и переносимую обратно в ЦП, – в совокупности называют **информацией состояния процесса** или **контекстом** процесса. Операция занесения контекста в ЦП и сохранения его в памяти называется **переключением контекста процесса**. Теперь дадим определение термина "процесс".

Обеспечиваемый операционной системой **процесс** – это среда, в которой пользователь может разрабатывать, запускать и отлаживать программы. Процесс определяется его контекстом. В противоположность системе прерываний простой ЭВМ PDP-11, где подпрограмма обслуживания прерываний могла определяться лишь вектором из двух слов памяти, в мультипрограммной системе контекст процесса – это нечто большее, чем содержимое регистров общего назначения. Контекст также должен содержать информацию о месте расположения инструкций и данных соответствующей задачи в

физической памяти. Очевидно, что для сохранения конфиденциальности каждый процесс должен также иметь свою стековую область памяти; в противном случае его информация легко может быть разрушена другими процессами, разделяющими с ним один и тот же ЦП. Поэтому в контекст процесса должна также входить информация, определяющая местоположение стека в физической памяти. Контекст и операция переключения контекста обусловливают возможность построения мультипрограммной системы.

Суммируя сказанное, можно представить себе процесс как некую виртуальную машину, определяемую контекстом процесса, в которой может быть разработана и выполнена некоторая задача. Для каждой задачи существует одна виртуальная машина, и виртуальные машины реализуются на одной физической машине на основе разделения времени. Все процессы совместно используют (разделяют) все ресурсы системы, но каждый процесс все-таки сохраняет свою конфиденциальность. Опираясь на эти сведения, мы можем рассмотреть систему VAX-11 несколько более детально.

ЦЕНТРАЛЬНЫЙ ПРОЦЕССОР (ЦП)

Основные элементы центрального процессора ЭВМ VAX-11 показаны на рис. 10.4. В противоположность ЭВМ PDP-11 в этой машине мы располагаем шестнадцатью 32-битовыми регистрами и 32-битовым длинным словом состояния процессора. По соглашению, принятому для системы VAX-11, двоичные представления 8-, 16-, 32-, 64- и 128-битовой длины определяются соответственно как **байт, слово, длинное слово, четырехкратное слово** и **восьмикратное слово.**

Младшее слово $b_0, \ldots, b_{15}$ длинного слова состояния процессора (PSL) называется словом состояния процессора PSW со следующим определением бит, которое несколько отличается от случая PDP-11:

b_0 – код условия переноса (заема);
b_1 – код условия переполнения;
b_2 – бит нулевого условия;
b_3 – бит условия отрицательного числа;
b_4 – бит ошибки трассировки;
b_5 – бит разрешения действия по переполнению целого числа;
b_6 – бит ловушки по ошибке исчерпания диапазона чисел с плавающей точкой;
b_7 – бит разрешения действия по переполнению;
$b_8, \ldots, b_{15}$ – эти биты не используются.

Старшее слово $b_{16}, \ldots, b_{31}$ PSL определяется так:

$b_{16}, \ldots, b_{20}$ – уровень приоритета прерываний;
b_{21} – этот бит не используется;
b_{22}, b_{23} – предыдущий режим доступа;
b_{24}, b_{25} – текущий режим доступа;
b_{26} – действия со стеком прерываний;
b_{27} – первая часть инструкции завершена;
b_{28}, b_{29} – биты не используются;
b_{30} – бит ожидания решения трассировки;
b_{31} – бит совместимого режима.

Регистры R0, . . ., R11 используются в качестве регистров общего значения. Регистры R12 и R13 используются в процессе процедуры: регистр R12 служит указателем аргументов (УА); регистр R13 – указателем кадра (УК). Процесс процедуры – это более строгий процесс подпрограммы, т. е. для процедуры процесс вызова и возврата является более строгим, чем для подпрограммы. В общем случае указатель кадра используется как указатель для сохранения содержимого слова состояния PSW и регистров общего назначения главной программы, а указатель аргументов служит указателем списка

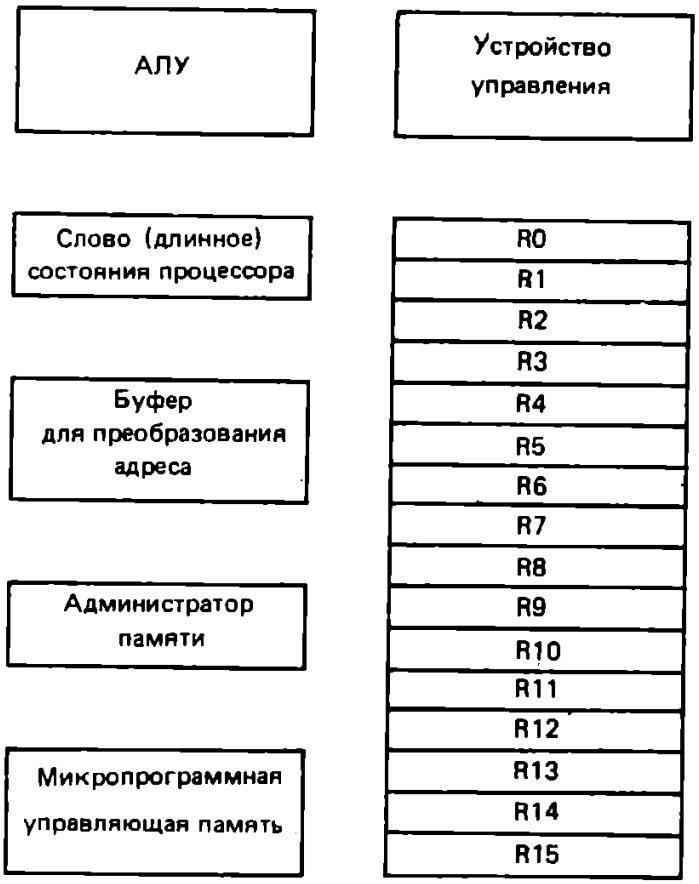


Рис. 10.4. Центральный процессор ЭВМ VAX-11

аргументов или массива, предназначенного для передачи аргументов или параметров вызываемой подпрограмме и получения от нее результатов. Различие между процедурой и подпрограммой мы рассмотрим далее. Регистр R14 – это общесистемный указатель стека (SP), а регистр R15 – программный счетчик (PC), он функционирует как регистр R7 в системе PDP-11.

Обратите внимание, что по сравнению с ЭВМ PDP-11 в схеме ЦП, изображенной на рис. 10.4, есть три новых блока: буфер для преобразования адреса, администратор памяти и необязательная микропрограммная память управления. Кратко опишем функции этих блоков. Напоминаем, что VAX-11 – 32-битовая машина. Все регистры в ЦП имеют длину 32 бита. Это означает, что у нас есть регистр PC, способный адресовать 2^{32} или более 4 миллиардов ячеек памяти. Другими словами, в терминах описанной выше системы виртуальной памяти вы располагаем виртуальным или логическим адресным пространством памяти в 4 миллиарда байт. Хотя в будущем нам может потребоваться такая гигантская физическая память и эта потребность может быть удовлетворена, пока мы вполне удовлетворимся физической памятью объемом в два миллиона байт. В таком случае необходим механизм преобразования адресов, который будет транслировать виртуальный адрес, определяемый инструкцией, в реальный адрес физической памяти. Блок буфера преобразования адресов является компонентом системы виртуальной памяти ЭВМ VAX-11.

Блок администратора памяти образуется из аппаратных и программных модулей. Его основные функции: 1) отображение или трансляция адреса виртуальной памяти в физический адрес и обеспечение того, чтобы для каждой пользовательской программы это было "прозрачно", чтобы каждая программа "чувствовала", что она работает не в виртуальном адресном пространстве, а в гигантской физической памяти с непрерывными адресами, начинающимися с нуля; 2) распределение свободных сегментов физической памяти между программами на основе равноправия; 3) обеспечение защиты конфиденциальности каждого процесса. (Более подробно управление памятью будет рассмотрено далее.)

Давайте обратимся теперь к третьему специальному блоку – **микропрограммной управляющей памяти**. В публикациях фирмы Digital Equipment Corporation этот блок называется управляющей памятью пользователя (УПП). Это высокоскоростная читаемая и записываемая память прямого доступа (ППД) объемом 1024 X 80. Если она установлена в системе, пользователи получают набор элементарных инструкций, с помощью которого они могут создать или изобрести свой собственный набор инструкций и записать его в УПП. Вновь созданный набор инструкций может совершенно отличаться от набора инструкций ЭВМ VAX-11. Такой процесс называется **микропрограммированием**. Концепция микропрограммирования очень близка к концепции макросов, определенных пользователем. Вспоминаем, что макросы могут создаваться пользователями на основе набора инструкций, поставляемого производителем ЭВМ. Например,

для макросов ввода-вывода, определенных нами в гл. 6, мы воспользовались форматом .MACRO . . . .ENDM, чтобы создать такие макросы, как READ, PRINT и DISPLAY. Для создания этих макроинструкций мы выбрали из набора инструкций ЭВМ PDP-11 такие инструкции, как TSTB, BPL и MOVB.

Очевидно, что созданные таким образом макросы имеют ограниченную эффективность и гибкость, поскольку мы лимитированы данным набором инструкций ЭВМ PDP-11. В противоположность этому (хотя для процесса микропрограммирования пользователю также дан некоторый набор инструкций, на основании которого создаются новые инструкции) этот набор инструкций более элементарен и очень гибок, его называют **набором микроинструкций**. Располагая таким набором, пользователь может изобрести практически любую нужную ему инструкцию. Следовательно, при наличии необязательной мультипрограммной управляющей памяти пользователи ЭВМ VAX-11 имеют возможность создать целое семейство специальных наборов инструкций для конкретных приложений. Кроме того, они могут написать полный набор инструкций для эмуляции такой ЭВМ, которая совершенно отлична от ЭВМ VAX-11.

ПАМЯТЬ

Поскольку устройство памяти, реализованное для ЭВМ VAX-11, представляет собой систему виртуальной памяти, необходимо изучить его более детально.

Организация физической (первичной) памяти

Хотя устройство физической памяти ЭВМ VAX-11 все так же основывается на традиционной концепции "адрес-содержимое", оно позволяет нам извлекать из памяти инструкции или данные в виде базовых элементов: байта, слова, длинного слова, четырехкратного и восьмикратного слова. Соответственно спецификации инструкции ЦП извлекает и обрабатывает операнд или операнды разного размера в базовых единицах. Фактически это просто расширение операций ЭВМ PDP-11. Например, в ЭВМ PDP-11 есть такие инструкции

```
MOVB  A, B
MOV   A, B
```

В первом случае операндами являются младшие байты ячеек A и B соответственно. Но во втором случае операнды – это целое слово из ячеек A и B. ЭВМ VAX-11 имеет следующие аналогичные инструкции:

```
MOVB    A,B     ;НА БАЙТОВОЙ ОСНОВЕ
MOVW    A,B     ;НА СЛОВНОЙ ОСНОВЕ
MOVL    A,B     ;НА ОСНОВЕ ДЛИННОГО СЛОВА
MOVQ    A,B     ;НА ОСНОВЕ ЧЕТЫРЕХКРАТНОГО СЛОВА
```

На рис. 10.5 показаны структуры адресуемых базовых элементов памяти с различным числом байт. Рисунок построен по степени увеличения числа байт, где A – это стартовый или базовый символьный адрес базового элемента памяти. В результате такой структуры памяти при корректировке содержимого регистра PC прибавляемое значение варьируется от одного байта до значительного числа байт в зависимости от размера операнда, над которым выполняется инструкция.

Кроме того, ЭВМ VAX-11 обеспечивает набор специальных инструкций, которые могут оперировать битовым полем переменной длины. То есть эти инструкции могут оперировать конкретной группой смежных бит, определяемых спецификатором операнда. Эта группа адресуется с помощью трех переменных: позиции, выраженной числом смещения в битах от адресной базы; размером группы, выраженной числом бит; базовым адресом. Например, если мы хотим адресовать операнд 3-битового размера с 10-битовым смещением от ячейки A, то мы специфицируем адрес следующим образом: позиция – 10; размер – 3; база – A. То есть эта инструкция будет оперировать битами b_{12}, b_{11} и b_{10} длинного слова, размещенного по адресу A.

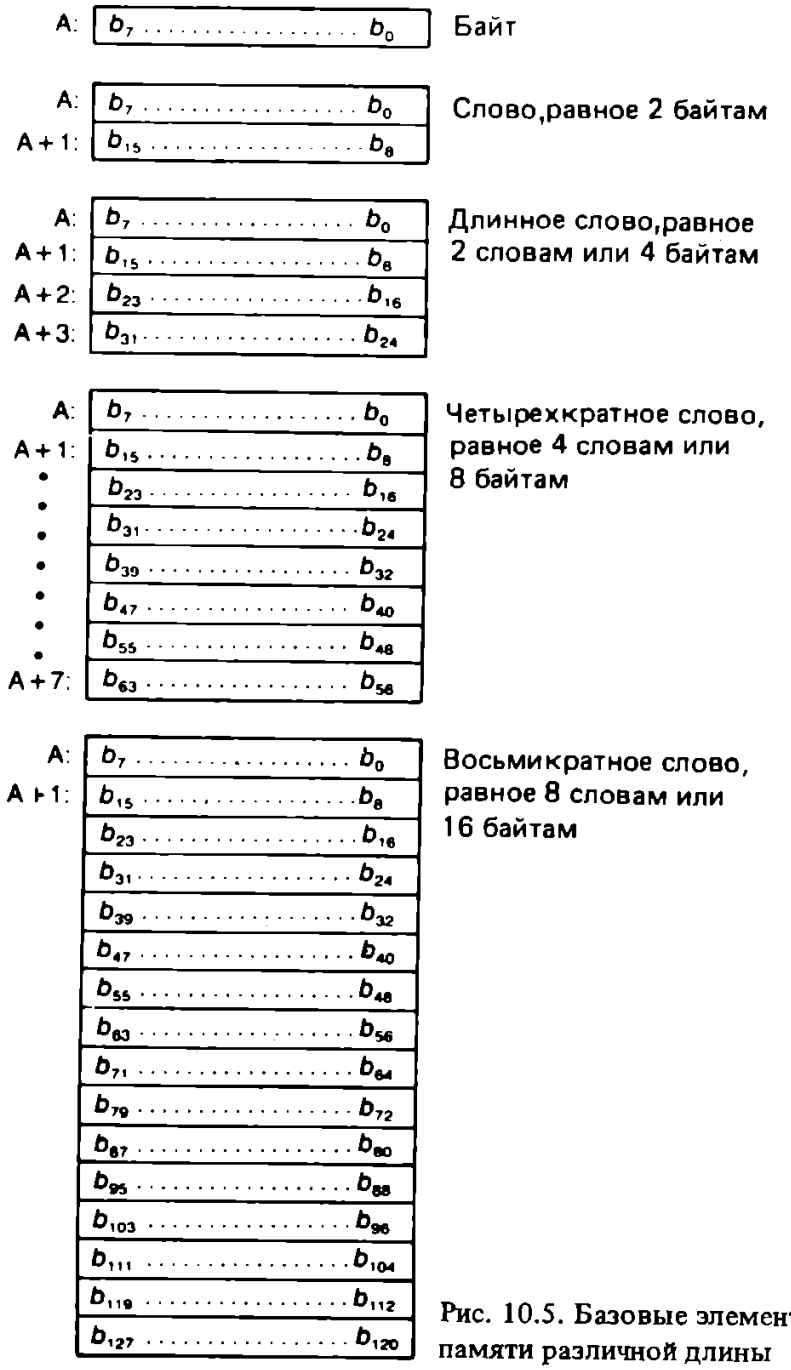


Рис. 10.5. Базовые элементы памяти различной длины

Нужно, наконец, здесь упомянуть, что для ЭВМ VAX-11 объем физической памяти может достигать 2^{30} байт, которые адресуются с помощью адресных бит $b_{29}, \ldots, b_0$. В настоящее время это, конечно, впечатляющий размер устройства физической памяти.

Организация виртуальной памяти

Как мы уже упоминали, 32-битовый регистр R15 ЭВМ VAX-11 служит программным счетчиком PC. Следовательно, с помощью регистра PC можно адресовать виртуальное адресное пространство размером 2^{32} (свыше 4 миллиардов) байт. Структурно виртуальное адресное пространство делится на четыре региона, определяемые битами b_{31} и b_{30}. На рис. 10.6 показано определение этих четырех регионов в виртуальном адресном пространстве. Обратите внимание, что каждому региону соответствует следующий диапазон адресов (в шестнадцатеричном представлении):

0000 0000 → 3FFF FFFF ≙ регион P0
4000 0000 → 7FFF FFFF ≙ регион P1
8000 0000 → BFFF FFFF ≙ системный регион
C000 0000 → FFFF FFFF ≙ резервный регион

Регион P0 предназначается для программы, у которой значение адреса увеличивается с ячейки 0000 0000 и растет в сторону более высоких значений.

Регион P1 отводится для функций управления, у которых значение адреса растет вниз. Например, адрес стековой памяти обычно растет в сторону меньших адресных значений.

Системный регион предназначается для программ операционной системы.

Четвертый регион зарезервирован для использования в будущем, т. е. младшие 2 миллиарда байт предназначены для пользовательских программ и определяются как пространство для процесса, тогда как старшие 2 миллиарда байт — это системное пространство. Обратите внимание, что до сих пор для описания объема памяти в качестве единицы мы использовали бит и байт. Однако нередко в схеме адресации памяти применяется другой термин — "страница". Одна страница эквивалентна 512 байтам. Для системы управления памятью транспортировка информации между физической и дисковой памятью осуществляется страницами, а не байтами или битами.

Адрес (в шестнадцатеричном представлении)	Содержимое
0000 0000 ⋮ 3FFF FFFF	Регион P0 (программа)
4000 0000 ⋮ 7FFF FFFF	Регион P1 (управление)
8000 0000 ⋮ BFFF FFFF	Системный регион
C000 0000 ⋮ FFFF FFFF	Зарезервировано

Рис. 10.6. Адресное пространство виртуальной памяти

Вспоминаем, что в разделе "Принцип виртуальной памяти" мы ввели сегментирование пользовательской программы, располагающейся постоянной на диске, на секции, а физической памяти – на сегменты. После сегментирования секции программы загружаются в сегменты памяти по одной секции каждый раз. Секция в области пользовательской программы и сегмент в области физической памяти однозначно определяются двумя аргументами: базовым или стартовым адресом и длиной секции (сегмента), выраженной числом байт. Например, в вышеприведенном примере мы можем определить секцию 1 по базовому адресу 1000 и длине секции 1000 байт, а сегмент 0 – по базовому адресу 2000 и длине сегмента 1000 байт.

Длину в байтах в данном случае не обязательно задавать константой, она может иметь любое значение. Однако в системе, использующей страницы, такой как система памяти для ЭВМ VAX-11, мы называем секцию или сегмент страницей, имеющей постоянную длину 512 байт. Поскольку длина страницы постоянна, страница может однозначно определяться одним значением базового адреса. Соответственно адрес памяти в ЭВМ VAX-11 может быть задан номером страницы и числом байтового смещения от базового адреса этой страницы.

Для простоты представим себе систему страничной организацией памяти, в которой используется 16-битовый регистр PC. Определим биты $b_8, b_7, \ldots, b_0$ регистра PC как поле для задания байтового смещения, а биты $b_{15}, b_{14}, \ldots, b_9$ как поле для задания страницы. Тогда, например, для битовой конфигурации регистра PC

Поле номера страницы	Поле байтового смещения
0 000 010	000 000 111

можно сказать, что регистр PC в настоящее время указывает на адрес седьмого байта на странице 2. Следует иметь в виду, что для 16-битовой машины с 16-битовой длиной регистра PC мы можем располагать адресным пространством в 256 страниц. Но давайте теперь вернемся к системе VAX-11.

Поскольку ЭВМ VAX-11 – 32-битовая ЭВМ, поле страницы у нее длиной 21 бит.

Но так как в ЭВМ VAX-11 применяется система виртуальной памяти, у нее есть виртуальное адресное пространство и физическое адресное пространство. На рис. 10.7, *а* и *б* показаны формат адресного указателя и организация адресного пространства соответственно для схем виртуальной и физической памяти. Важно помнить, что виртуальная память — это просто абстрактное адресное пространство, обеспечиваемое банком дисковой памяти, имеющим гигантский объем памяти, адресное пространство которого организуется с помощью номера привода, номера дорожки, номера сектора и номера байта. За систематизированное сохранение на диске наших программных файлов отвечает операционная система. Следовательно, программисту нужно только вообразить себе, что он располагает неограниченным числом пространств виртуальной памяти, по одному на каждый процесс, организованных в виде непрерывных последовательностей байт и страниц с адресами, начинающимися с ячейки нуль.

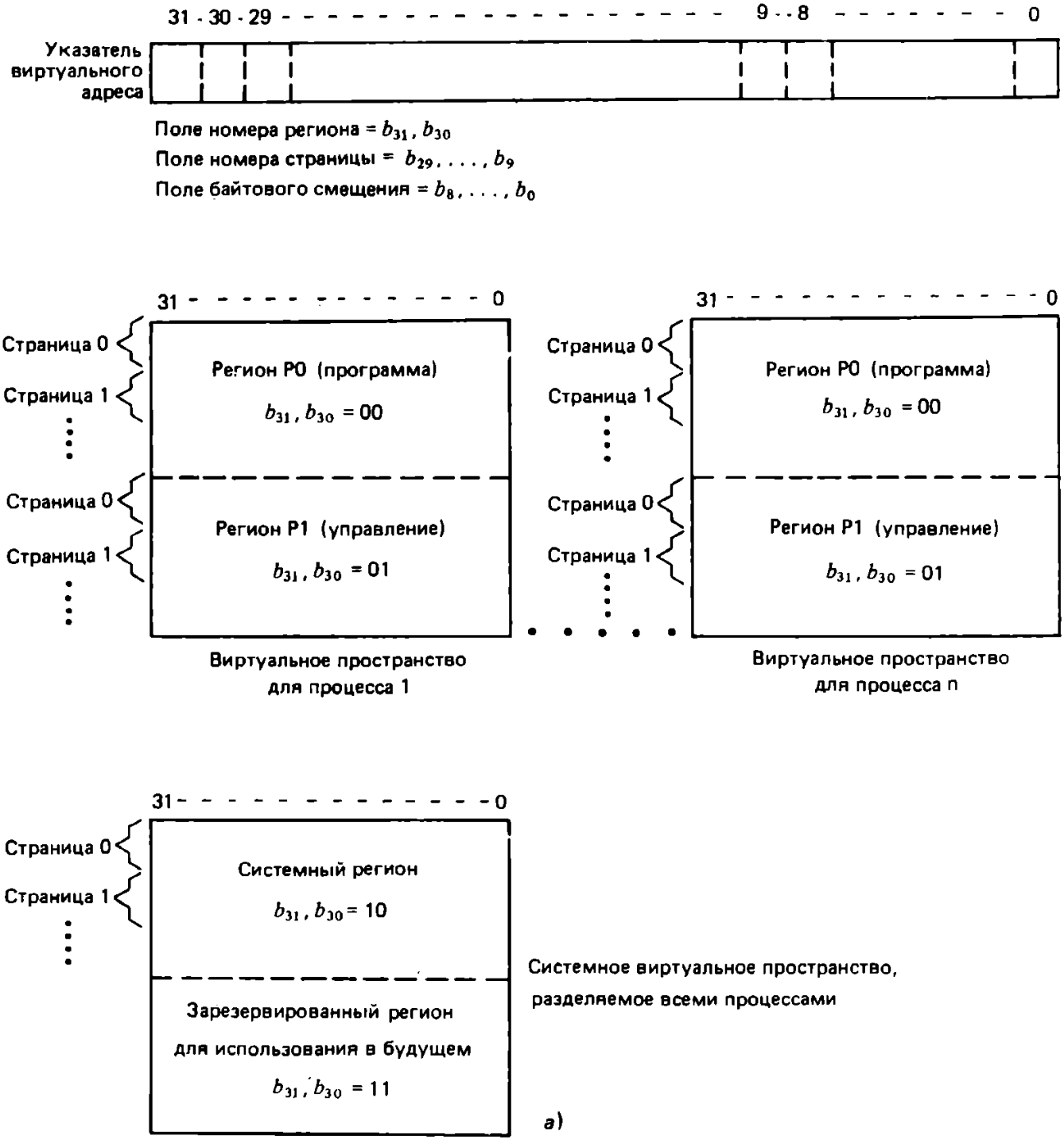


Рис. 10.7, *а*. Схема пространства виртуальной памяти

Предположим, например, что есть три задачи[1], которые мы хотим реализовать на ЭВМ VAX-11. Вероятнее всего, для каждой задачи у нас будет главная программа, подпрограммы и макросы, которые следует отредактировать, ассемблировать, скомпоновать, отладить и запустить в среде (или процессе) на ЭВМ VAX-11. Для каждой задачи нам потребуется вызывать системные программы, такие как **редактор, ассемблер, отладчик** и т. п., которые помогут в разработке программы нашей задачи. Поскольку эти системные программы постоянно располагаются в системном регионе (b_{31}, $b_{30} = 10$) и они одни и те же для любой задачи, все задачи могут разделять (совместно использовать) один и тот же системный регион виртуального пространства.

Следовательно, для каждого процесса, в котором может разрабатываться и выполняться задача в виртуальном пространстве, мы должны иметь регионы P0 и P1 (см. рис. 10.7, *а*). Однако, как показано, у нас только один системный регион. Следовательно, мы имеем три пары регионов P0 и P1 в виртуальной памяти, но только один системный. Давайте вернемся к нашему примеру. Поскольку у нас три задачи (A, B и C), то потребуется три процесса, например, с номерами 1, 2 и 3, в которых можно разрабатывать и запускать эти задачи. Каждый процесс обладает собственным контекстом. Помимо этого задаче A может потребоваться 100 или 200 страниц виртуальной памяти, тогда как другим задачам может потребоваться большее или меньшее число страниц. Однако у них будут собственные регионы P0 и P1, причем программа начинается с нулевой страницы и байта нуль в регионе P0 и простирается в сторону больших адресов непрерывно, а структуры управления, такие как стековая память, в регионе P1 начинаются с верхнего предельного адреса и непрерывно распространяются в сторону меньших адресов. В результате каждая задача для разработки и запуска своих программ реализуется как бы собственной ЭВМ VAX-11.

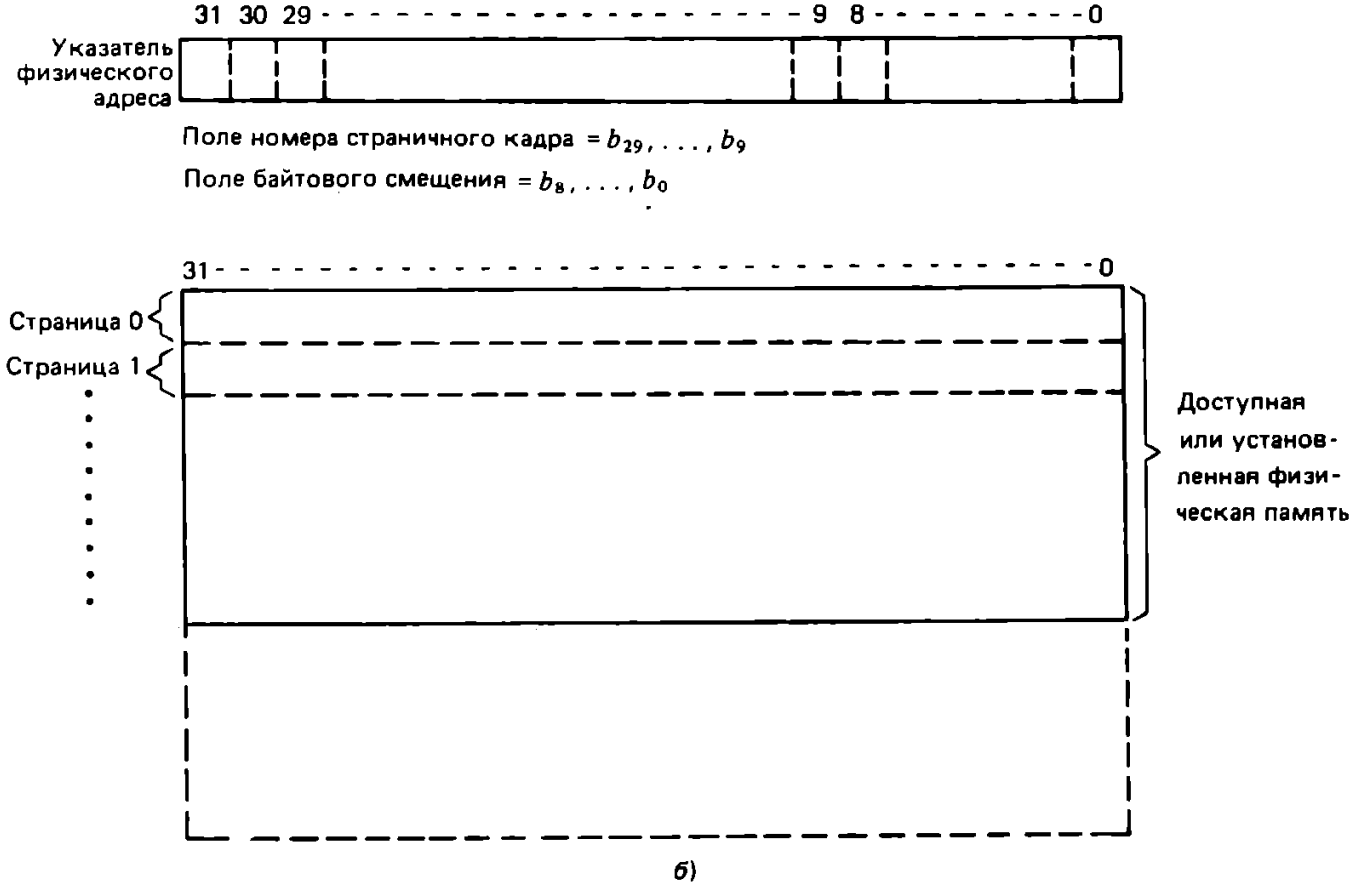


Рис. 10.7, *б*. Схема пространства физической памяти

[1] Под задачей здесь и далее понимается подготовленная к выполнению программа. – *Прим. перев.*

Преобразование (трансляция) адресов

В силу концепции виртуальной памяти каждая задача может иметь пространство памяти объемом 3 миллиарда байт, в котором располагаются три региона (P0, P1 и системный). Если мы хотим одновременно разрабатывать и запускать три задачи, нам потребуется 7 миллиардов байт физической памяти (напоминаем, что они разделяют один и тот же системный регион). Очевидно, что в настоящее время непрактично требовать такой гигантской физической памяти. К счастью, поскольку в каждой программе в один и тот же момент времени выполняется только одна инструкция, нет необходимости хранить все программы в физической памяти одновременно. Три задачи могут разделять одно и то же устройство физической памяти и один ЦП.

Для простоты предположим, что все три задачи уже разработаны, скомпонованы и готовы к выполнению и операционная система загрузила три задачи A, B и C в две, три и четыре страницы соответственно в физическую память так, как это показано на рис. 10.8. По мере выполнения оставшаяся часть программы для каждой задачи будет постоянно заноситься в физическую память для замены старых страниц. Тем временем задачи A, B и C будут выполняться ЦП в режиме временного мультиплексирования. Обратите внимание, что стартовый адрес в пространстве физической памяти

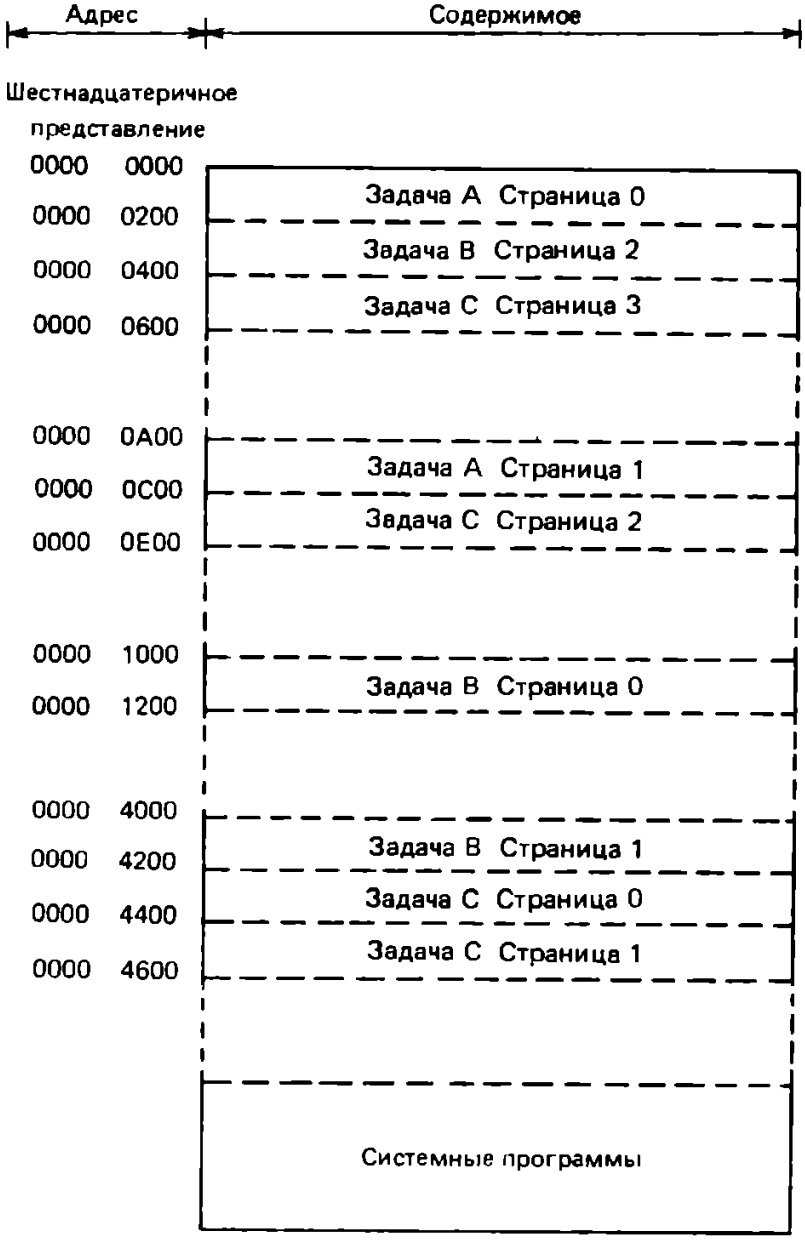


Рис. 10.8. Распределение физической памяти операционной системой

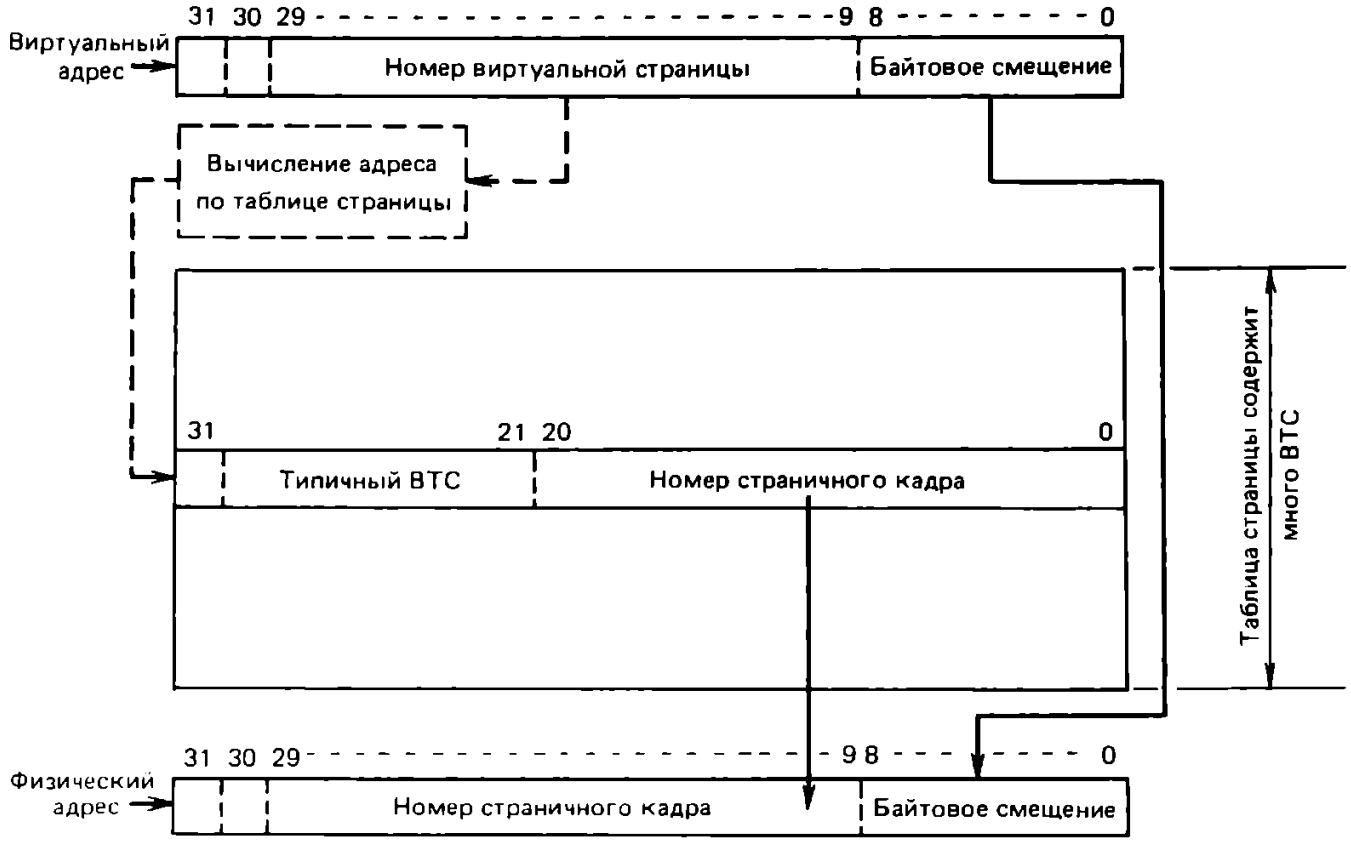


Рис. 10.9. Преобразования адреса

для задачи B – страница 0 – будет, как показано на рис. 10.8, равен 0000 1000, тогда как в виртуальном пространстве задачи B этот адрес равен 0000 0000. Поэтому при выполнении задачи B инструкция будет просто ссылаться на адрес виртуального пространства.

Однако перед извлечением операнда инструкции необходимо выполнить трансляцию виртуального адреса в физический адрес, так как во время выполнения программа находится в физической памяти. Но каким образом транслятор адреса узнает физическое местоположение рассматриваемой программы? В ЭВМ VAX-11 это делается с помощью таблицы уникального преобразования адресов, называемой **таблицей страниц** и показанной на рис. 10.9. Обратите внимание, что любая инструкция программы является первоисточником виртуального адреса. В виртуальном адресе значение байтового смещения является тем же самым, что и в физическом адресе, но информация о номере виртуальной страницы ($b_{29}, \ldots, b_9$) не дает действительного номера страницы в физическом адресе, а используется для вычисления адреса конкретного места в таблице страниц, где находится информация о соответствующем номере физической страницы. Поэтому содержимое элемента таблицы страниц называют **входом таблицы страниц** (ВТС), ведь оно указывает на вход в нужную физическую страницу.

Снова обратимся к рис. 10.8. Если, например, инструкция задачи C ссылается на виртуальный адрес восьмого байта страницы 0, то соответствующий физический адрес будет 0000 4208, тогда как виртуальный адрес будет 0000 0008. К счастью, процесс преобразования адресов выполняется системой управления памятью (администратором памяти), так что пользователю нужно заботиться только о виртуальном адресе. Однако мы должны понимать, что как только содержимое виртуальных страниц задачи будет загружено в страницы физической памяти, должна быть скорректирована таблица страниц, т. е. содержимое соответствующих ВТС. Эта корректировка также выполняется администратором памяти. Очевидно, что полное число табличных входов определяется размером задачи, выраженным числом страниц. То есть для каждой страницы требуется один вход таблицы страниц.

Как упоминалось выше, блок управления памятью (администратор памяти) выполняет следующие основные функции: 1) преобразование (трансляция) адреса виртуальной памяти в адрес физической памяти; 2) назначение пространства физической памяти для каждого процесса; 3) предохранение конфиденциальности каждого процесса. В этом разделе мы рассмотрим вместе все элементы памяти, такие как контекст, таблица страниц, вход таблицы страниц и т. п., чтобы показать, как осуществляются функции управления памятью.

Нам надо, чтобы несколько задач могли быть запущены одновременно в вычислительной системе, которая располагает центральным процессором, способным работать с высокой скоростью, и высокоскоростной первичной (полупроводниковой) памятью, поддержанной высокоскоростной массовой (вторичной) памятью, такой как система дисковой памяти. В системе VAX-11 есть ЦП и физическая память довольно большого объема. Все задачи разделяют время ЦП и пространство физической памяти таким образом, что каждая задача выполняется как бы на собственной ЭВМ. Планировщик времени ЦП операционной системы назначает кванты времени ЦП для каждой задачи, а администратор памяти, также являющийся частью операционной системы, управляет процессом разделения пространства памяти для каждой задачи. Адресуемыми единицами физической памяти являются следующие: 8 бит (байт); 2 байта (слово); 4 байта (длинное слово); 8 байт (четырехкратное слово); 16 байт (восьмикратное слово); 512 байт (страница).

Инструкции обращаются к байту, слову, длинному слову, четырех- или восьмикратному слову, а администратор памяти пересылает информацию между вторичной (дисковой) памятью и первичной (полупроводниковой) памятью в виде страниц. Задача, которая может состоять из подпрограмм, макросов, данных и т. п., всегда располагается в последовательных байтах виртуального адресного пространства. Однако в доступную физическую (первичную) память она загружается страница за страницей. Следовательно, как показано на рис. 10.8, в физической памяти номера страниц задачи могут быть непоследовательными, но номера байт внутри страницы должны быть последовательными. К счастью, после загрузки виртуальной страницы в физическую память появляется назначенный этой виртуальной странице уникальный физический начальный адрес. Поэтому имеет место взаимно однозначное соответствие между адресными значениями виртуальных и физических адресов. Как показано на рис. 10.8, если инструкция задачи С ссылается на ячейку в виртуальном адресном пространстве (а следует заметить, что инструкция всегда ссылается на виртуальный адрес), например на пятый байт страницы 1 или на виртуальный адрес

```
   31 30 29 . . . . . 9 8 . . . . . 0
= | 0  0  0  0 . . . . . 0 0 1 0 0 0 0 0 0 1 0 1 |
```

= 00000205 (шестнадцатеричное число)

то соответствующий ему физический адрес будет:

```
   31 30 29 . . . . . 9 8 . . . . . 0
= | 0  0    0 . . . 0 1 0 0 0 1 0 0 0 0 0 0 0 1 0 1 |
```

= 00004405 (шестнадцатеричное число)

Следовательно, преобразование адреса сводится только к табличному поиску. Поскольку номера байт внутри страницы следуют непрерывно, то для осуществления ссыл-

ки на байтовый адрес необходима информация о начальном адресе каждой физической страницы. Другими словами, необходима таблица, преобразующая номера виртуальных страниц в номера физических страниц. Она называется таблицей страниц. Очевидно, что как только задача загружена (или частично загружено) в физическую память, для нее должна существовать уникальная таблица страниц. Однако, как показано на рис. 10.7, *а*, пространство виртуальной памяти разделяется на регион P0, регион P1 и системный регион, т. е. задача может иметь свою программу в регионе P0, свою стековую память в регионе P1 и может вызывать системные программы из системного региона. Значит, для каждой задачи должно быть три таблицы страниц. Но поскольку системный регион является общим для всех задач, то для каждой задачи существует две таблицы страниц – одна для региона P0 и одна для региона P1. Кроме того, все задачи разделяют одну и ту же таблицу страниц для системного региона.

Познакомимся теперь детально со структурой таблицы страниц. Таблица страниц представляет собой блок последовательно расположенных длинных слов, записанных в область памяти. Она определяется своим начальным адресом и длиной блока. Адрес каждого длинного слова таблицы страниц является функцией номера виртуальной страницы, задаваемого в виртуальном адресе, как это показано на рис. 10.9. Содержимое каждого длинного слова таблицы страниц называется входом таблицы страниц (ВТС). ВТС обеспечивает следующую информацию: 1) режим доступа к этой странице программы для целей обеспечения конфиденциальности; 2) загружена ли эта виртуальная страница в страницу физической памяти; 3) номер кадра (базы) соответствующей физической страницы.

Каждая таблица страниц определяется регистром базы и регистром длины. Например, системная таблица страниц (СТС) определяется системным регистром базы (СРБ), содержимое которого есть адрес СТС, и системным регистром длины (СРД). Аналогично регистр базы региона P0 (РБР0) и регистр длины региона P0 (РДР0) определяют таблицу страниц региона P0, а регистры РБР1 и РДР1 – таблицу страниц региона P1.

В качестве примера рассмотрим инструкцию, в которой есть ссылка на определенную системную страницу в спецификации виртуального адреса в спецификаторе операнда, как показано на рис. 10.9. Администратор памяти оставит прежним байтовое смещение, но извлечет номер виртуальной страницы и вычислит адрес ВТС в системной таблице страниц с помощью регистров СРБ и СРД. Когда определено местоположение ВТС, находится номер физической страницы, который затем присоединяется к байтовому смещению для образования физического адреса. Этот процесс напоминает одноуровневую косвенную адресацию, описанную при рассмотрении режимов адресации системы PDP-11. Но ВТС для расположений регионов P0 и P1 вычисляются с помощью системной таблицы страниц, поэтому этот процесс аналогичен двухуровневой косвенной адресации.

Формат ВТС определяется следующим образом:
номер кадра страницы – биты $b_{20}, \ldots, b_0$;
поле, зарезервированное для операционной системы, – биты $b_{25}, \ldots, b_{21}$;
бит модификации – бит b_{26};
защита – биты $b_{30}, \ldots, b_{27}$;
бит действительности – бит b_{31}.

Прочитав вход таблицы страниц, ЦП или администратор памяти проверяет код защиты на возможность доступа к этой странице. В ЭВМ VAX-11 каждой инструкции назначается один из четырех режимов[1] доступа: ядра, исполнителя, супервизора и пользователя; причем привилегии доступа они имеют в порядке уменьшения. Другими словами, режим ядра имеет наивысшую привилегию. То есть "вызывающая" страница в режиме ядра может осуществлять доступ к любой странице, поскольку три другие

[1] Имеются в виду режимы работы центрального процессора. – *Прим. перев.*

имеют меньший ранг привилегии. Если эта страница проходит тест защиты, то ЦП проверяет бит действительности. Если бит действительности равен 1, значит, данная страница была загружена в физическую память и номер кадра страницы $b_{20}, \ldots, b_0$ в ВТС является номером страницы физической памяти, тем самым преобразование адреса виртуальной памяти в адрес физической памяти завершено. Коды защиты ВТС показаны в табл. 10.1. Если, например, текущая инструкция находится в режиме ядра и код защиты в ВТС – 0011, то страница рассматриваемого входа таблицы страниц может быть только прочитана этой инструкцией, но не записана.

Таблица 10.1. **Коды защиты ВТС**

Двоичный код				Текущий режим доступа				Комментарий
b_{30}	b_{29}	b_{28}	b_{27}	Я	И	С	П	
0	0	0	0	–	–	–	–	Нет доступа
0	0	0	1	Непредсказуемый				Зарезервирован
0	0	1	0	ЧЗ	–	–	–	
0	0	1	1	Ч	–	–	–	
0	1	0	0	ЧЗ	ЧЗ	ЧЗ	ЧЗ	Любой доступ
0	1	0	1	ЧЗ	ЧЗ	–	–	
0	1	1	0	ЧЗ	Ч	–	–	
0	1	1	1	Ч	Ч	–	–	
1	0	0	0	ЧЗ	ЧЗ	ЧЗ	–	
1	0	1	0	ЧЗ	Ч	Ч	–	
1	0	1	1	Ч	Ч	Ч	–	
1	1	0	0	ЧЗ	ЧЗ	ЧЗ	Ч	
1	1	0	1	ЧЗ	ЧЗ	Ч	Ч	
1	1	1	0	ЧЗ	Ч	Ч	Ч	
1	1	1	1	Ч	Ч	Ч	Ч	

Обозначения в таблице: – – нет доступа; Ч – только чтение; ЧЗ – чтение и запись; Я – ядро; И – исполнитель; С – супервизор; П – пользователь.

Подводя итоги, можно сказать, что администратор памяти, основываясь на виртуальном адресе, извлекает правильный вход таблицы страниц; основываясь на информации из ВТС, он проверяет режим доступа и преобразует адрес виртуальной памяти в адрес физической памяти.

МУЛЬТИПРОГРАММИРОВАНИЕ

Теперь мы можем рассмотреть работу системы мультипрограммирования в ЭВМ VAX-11 в целом. Пусть на ЭВМ VAX-11 предстоит выполнение некоторой группы задач. Операционная система загрузит несколько страниц каждой задачи в доступную физическую память и скорректирует входы таблицы страниц каждой задачи. Затем операционная система будет мультиплексировать время ЦП для выполнения этих задач. Для каждой задачи операционная система обеспечивает среду, называемую процессом, в которой задача выполняется. Всякий раз, когда какой-либо процесс переключается для выполнения его центральным процессором, он должен привносить в систему свое собственное начальное строение или информацию о регистрах PC, R0, R1, . . . , спецификациях таблиц страниц (регистре базы и регистре длины), состоянии процессора и т. д. После исчерпания выделенного кванта времени процесс должен быть отсоединен от ЦП, а информация о его состоянии должна быть где-то записана для

будущей работы. Информация о состоянии процесса – это контекст, а процесс переключения называется переключением контекста. Блок памяти, в котором записывается информация о состоянии, называется управляющим блоком процесса (УБП). Содержимое УБП показано на рис. 10.10. В ЦП для переключения контекста существует аппаратный управляющий блок процесса.

Указатель стека ядра
Указатель стека исполнителя
Указатель стека супервизора
Указатель стека пользователя
R0
R1
R2
R3
R4
R5
R6
R7
R8
R9
R10
R11
R12 – указатель аргументов
R13 – указатель кадра
PC
Длинное слово состояния процессора
P0 – регистр базы
P0 – регистр длины
P1 – регистр базы
P1 – регистр длины

Рис. 10.10. Управляющий блок процесса

10.3. ИНСТРУКЦИИ И РЕЖИМЫ АДРЕСАЦИИ

Изучив основы структуры ЭВМ, системы виртуальной памяти и мультипрограммирования для ЭВМ VAX-11, теперь можно программировать на языке ассемблера. Основное внимание уделим тем особенностям, которые отличают ЭВМ VAX-11 от ЭВМ PDP-11. Рассмотрим сначала отличия типов данных, формата инструкций и режимов адресации.

ТИПЫ ДАННЫХ

Как уже отмечалось, извлечение из первичной памяти ЭВМ VAX-11 производятся байтами, словами, длинными словами, четырех- и восьмикратными словами в отличие от ЭВМ PDP-11, в которой используются только байты и слова. Таким образом, целые числа могут размещаться во всех этих единицах памяти. Как и в ЭВМ PDP-11, целые числа могут быть представлены со знаком и без знака, причем для целых чисел со знаком используется дополнение до двух. Рассмотрим теперь отдельно представление чисел с плавающей точкой для ЭВМ VAX-11, которое является для нее стандартным.

Ниже показано число с плавающей точкой, представленное 32-битовым длинным словом:

31 . . . . . 16 15 14 . . . . . 7 6 0

Дробная часть (младшая часть)	Знак	Порядок	Дробная часть (старшая часть)

Обратите внимание, что число с плавающей точкой представляется битом знака b_{15}, порядком $b_{14}, \ldots, b_7$ и дробной частью (старшая часть которой располагается в битах $b_6, \ldots, b_0$, а младшая часть – в битах $b_{31}, \ldots, b_{16}$, причем бит b_{31} сцепляется с битом b_0). Поскольку после нормализации самый старший бит дробной части всегда равен 1, для экономии бит в представлении этот бит всегда опускается.

Например, если дробная часть есть .1010 0000 1100 0000 0000 1111, то в поле дроби она будет запакована так:

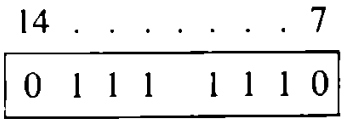
6 0

0 1 0 0 0 0 0

31 16

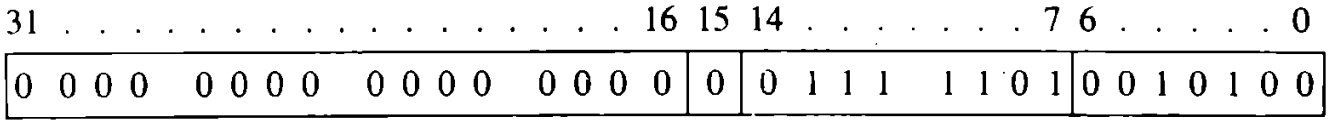
1 1 0 0 0 0 0 0 0 0 0 0 1 1 1 1

Для представления порядка сначала фактическое значение порядка складывается с константой, равной 128, а затем помещается в поле порядка. Например, если фактический порядок есть –2, то в поле порядка мы найдем –2 + 128 = 126, что может быть показано так:



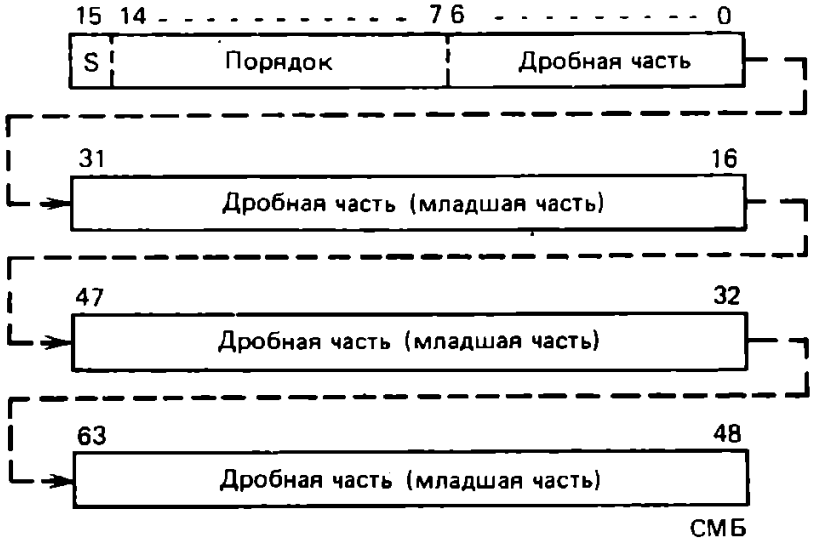
При таком представлении порядок всегда будет положительным двоичным числом, но фактический порядок = число в поле порядка – 128

В качестве другого примера покажем представление числа 0.100101×2^{-3};



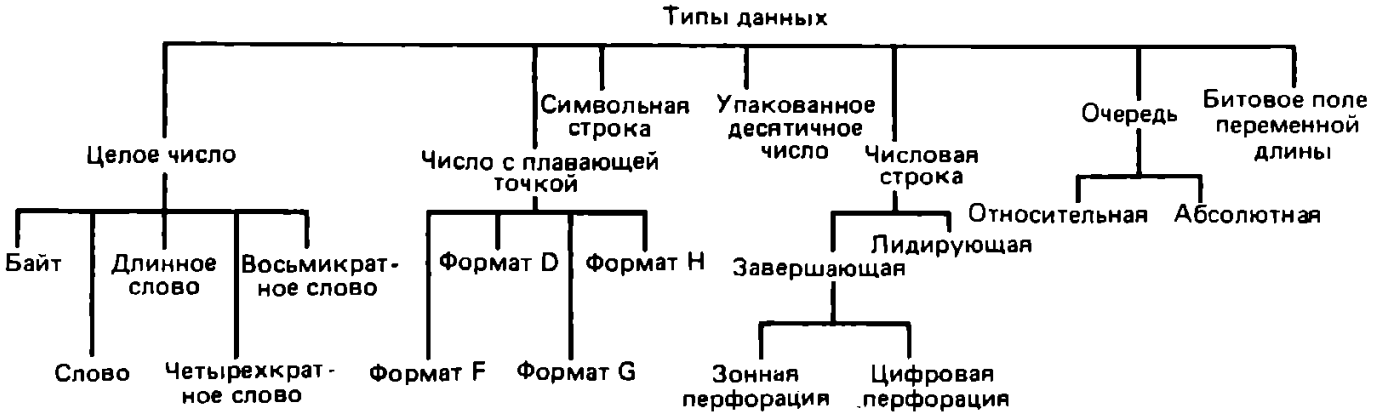
Здесь фактический порядок равен –3, а значение в поле порядка равно –3 + 128 = 125 = $= 0111\ 1101_2$.

Описанное выше представление – это представление чисел с плавающей точкой одинарной точности. В ЭВМ VAX-11 пользователи всегда предпочитают использовать представление с плавающей точкой двойной точности, формат которого определяется следующим образом:



В дополнение к целым числам, числам с плавающей точкой одинарной и двойной точности ЭВМ VAX-11 предлагает пользователю и другие типы данных. Полный набор типов данных системы VAX-11 показан в табл. 10.2.

Таблица 10.2. **Типы данных ЭВМ VAX-11**



ИНСТРУКЦИИ

ЭВМ VAX-11 располагает мощным набором инструкций для разработки как простых, так и очень сложных программ. В дополнение к гигантскому виртуальному адресному пространству эта ЭВМ может обрабатывать различные типы данных, и существуют такие инструкции, которые могут иметь до шести спецификаторов операндов. Например, инструкция ЭВМ PDP-11

```
ADD  A, B
```

имеет два спецификатора операндов. В следующем подразделе мы познакомимся с общим форматом набора инструкций ЭВМ VAX-11.

В базовой системе PDP-11 мы имели только два типа данных для целых чисел: байт и слово. Когда инструкция выполняет операцию с байтом, то к мнемонике инструкции добавляется буква B. Например, инструкция

```
MOVB  A, B
```

предписывает ЦП скопировать содержимое младшего байта ячейки A в младший байт ячейки B. Если буква B не добавляется, это означает, что инструкция работает со словом. Инструкции ЭВМ VAX-11 дополняются буквами B, W, L, Q, F и т. п., где B означает байт, W – слово, L – длинное слово, Q – четырехкратное слово, F – число с плавающей точкой и т. д. Кроме того, где нужно, за спецификацией типа данных следует число, показывающее число спецификаторов операндов. Например, инструкция

```
ADDB2  A, B
```

работает с байтовыми данными и имеет два спецификатора операндов. А инструкция

```
ADDL3  A, B, C
```

работает с данными в виде длинного слова (32-битового) и имеет три спецификатора операндов. То есть в данном случае инструкция предписывает процессору сложить длинные слова данных из ячеек A и B и поместить результат в ячейку C. Данные в ячейках A и B остаются без изменения.

Помимо добавления информации о типе данных и числе операндных спецификаторов программисту разрешается для указания того, что операция ссылается на адрес, а не на содержимое адреса, к мнемонике инструкции добавлять букву A. Например, инструкция

```
MOVAL  A, B
```

означает, что в ячейку B копируется адресное значение метки A (т. е. #A, а не содержимое ячейки A).

Набор инструкций для ЭВМ VAX-11 приведен в приложении E. В общем случае формат предложения инструкции в ЭВМ VAX-11 соответствует обычному соглашению, когда инструкция имеет четыре поля: адресную метку; мнемонический код операции, спецификаторы операндов, комментарии.
Они разделяются обычным образом:

```
АДРЕСНАЯ_МЕТКА: КОД_ОПЕРАЦИИ   СПЕЦИФ1,СПЕЦИФ2,...   ;КОММЕНТАРИИ
```

Отметим, что в противоположность системе PDP-11 из выпускаемой фирмой документации исчезло восьмеричное представление чисел. В системе VAX-11 используются только двоичное, десятичное и шестнадцатеричное представления.

Поскольку инструкции VAX-11 имеют переменную длину, число занимаемых байт памяти варьируется в зависимости от типа операционного кода и режимов адресации, используемых для спецификаторов операндов. Общий формат инструкции VAX-11 показан на рис. 10.11. Обратите внимание, что код операции может занимать один или два байта памяти, каждый спецификатор операндов – один или несколько байт.

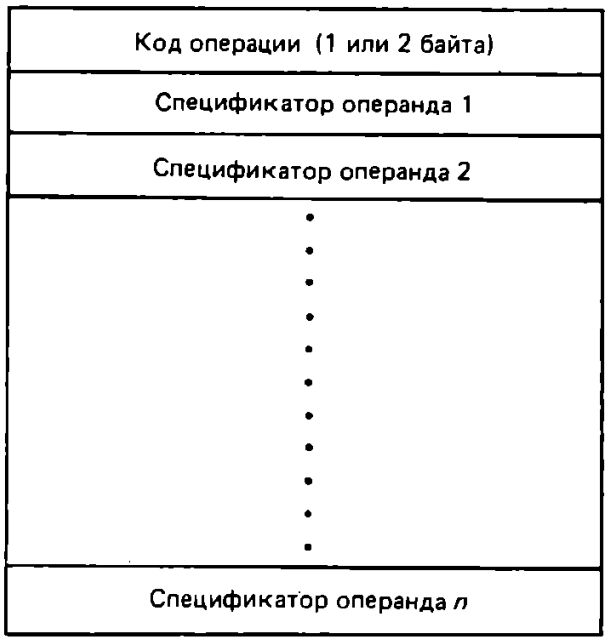


Рис. 10.11. Общий формат инструкций ЭВМ VAX-11

РЕЖИМЫ АДРЕСАЦИИ

Рассмотрим детально режимы адресации для инструкций ЭВМ VAX-11, разделив их на две группы: 1) режимы адресации, вовлекающие регистры общего назначения, R0, R1, . . . , R11, исключая регистр PC, R15; 2) режимы адресации, вовлекающие регистр PC.

РЕЖИМЫ АДРЕСАЦИИ БЕЗ РЕГИСТРА PC

Регистровый режим. Этот режим аналогичен режиму прямой адресации в инструкции ЭВМ PDP-11, за исключением того, что содержимое регистра 32-битовое, а не 16-битовое, как в ЭВМ PDP-11. Пример:

```
MOVW R1, R2  ;  (R1) → (R2)
```

Здесь копируется только младшее слово регистра R1 в младшее слово регистра R2. Если перед выполнением

(R1) = A09A779B
(R2) = 00000000

то после выполнения

(R1) = A090779B
(R2) = 0000779B

Напоминаем, что вместо восьмеричного представления используется шестнадцатеричное представление. Следующий пример:

```
CLRL  (R2)  ;  (R2) – содержимое регистра R2
```

Положим, что перед выполнением

(R2) = 00002120

и содержимое ячейки памяти 0000 212 равно AD9CF712. Тогда после выполнения в ячейке 0000 2120 будет содержаться 0000 0000. Воспользуемся картой памяти

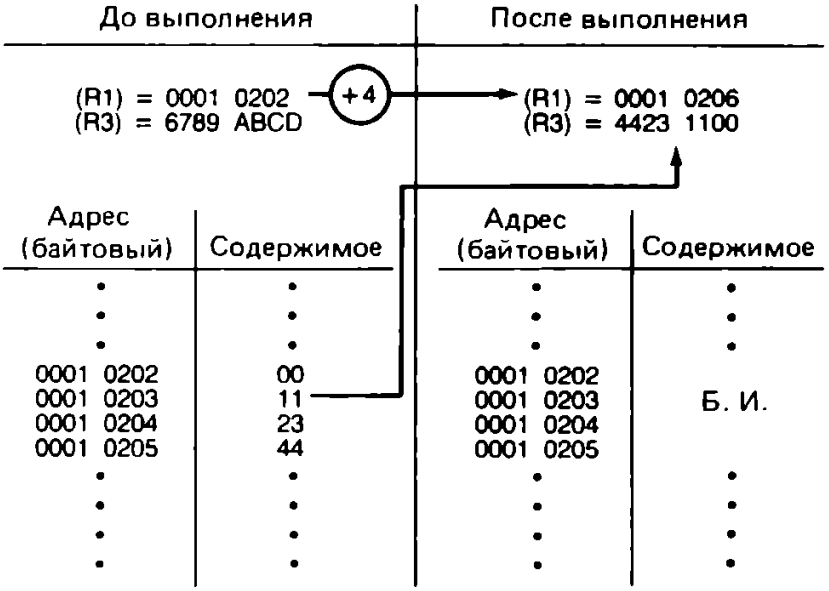
До выполнения		После выполнения	
(R2) = 0000 2120		(R2) = Б. И.*	
Адрес (байтовый)	Содержимое	Адрес (байтовый)	Содержимое
0000 2120	12	0000 2120	00
0000 2121	F7	0000 2121	00
0000 2122	9C	0000 2122	00
0000 2123	AD	0000 2123	00
•	•	•	•
•	•	•	•
•	•	•	•
•	•	•	•
•	•	•	•

*Б. И. – без изменения

Режим автоувеличения. *Пример:*

MOVL (R1) +, R3 ; (R1) + 4 → R1

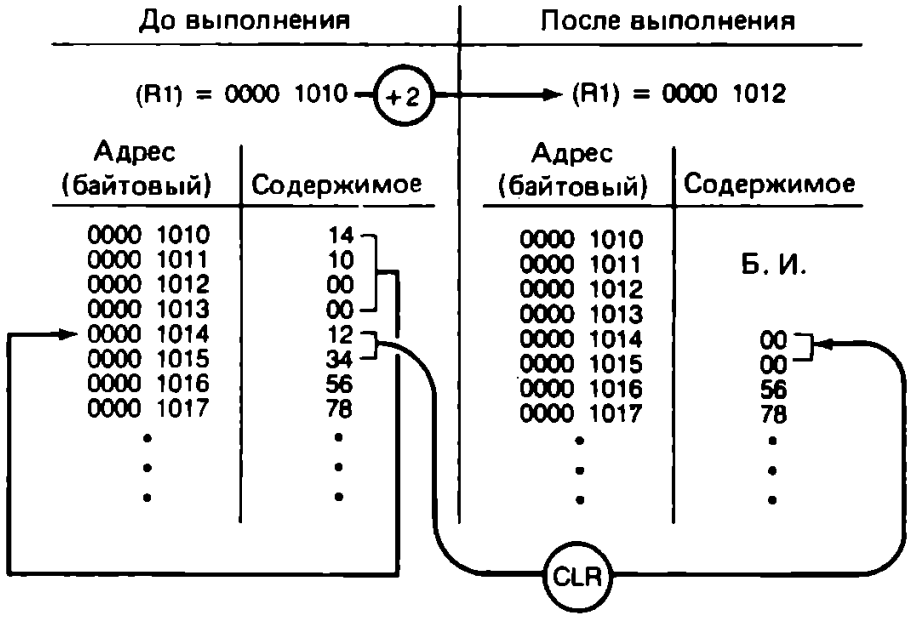
Увеличение содержимого регистра R1 на 4 происходит из-за того, что инструкция выполняет операцию с длинным словом. Карта памяти для этого примера:



Косвенный режим автоувеличения. *Пример:*

CLRW @(R1) + ; (R1) + 2 → R1

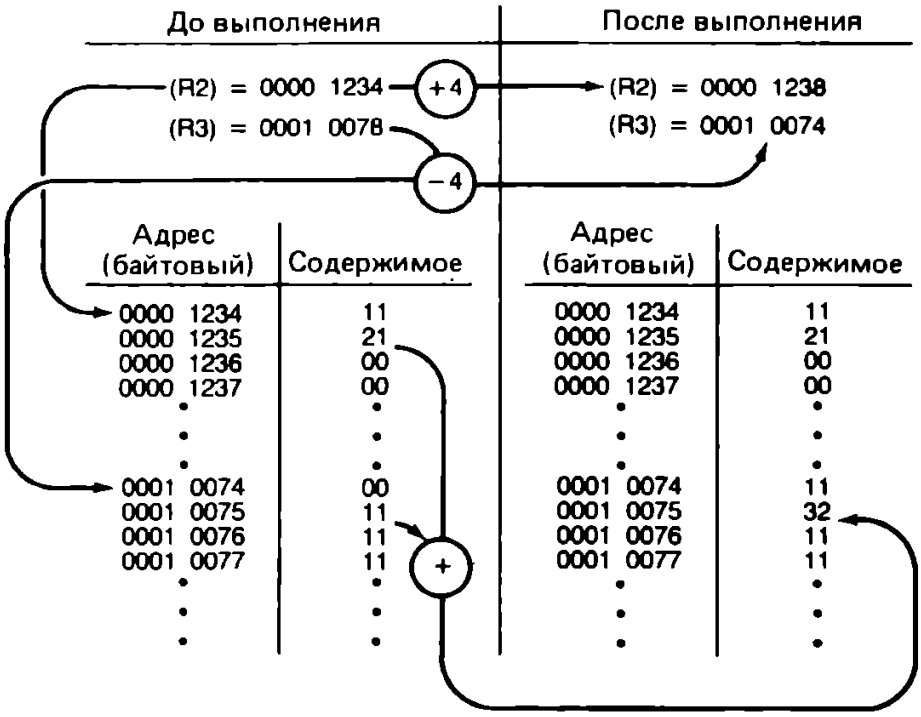
Происходит увеличение на 2, так как инструкция выполняет операцию со словом. Карта памяти:



Режим автоуменьшения. *Пример:*

ADDL (R2) +, –(R3) ; (R2) + 4 → R2, (R3) – 4 → R3

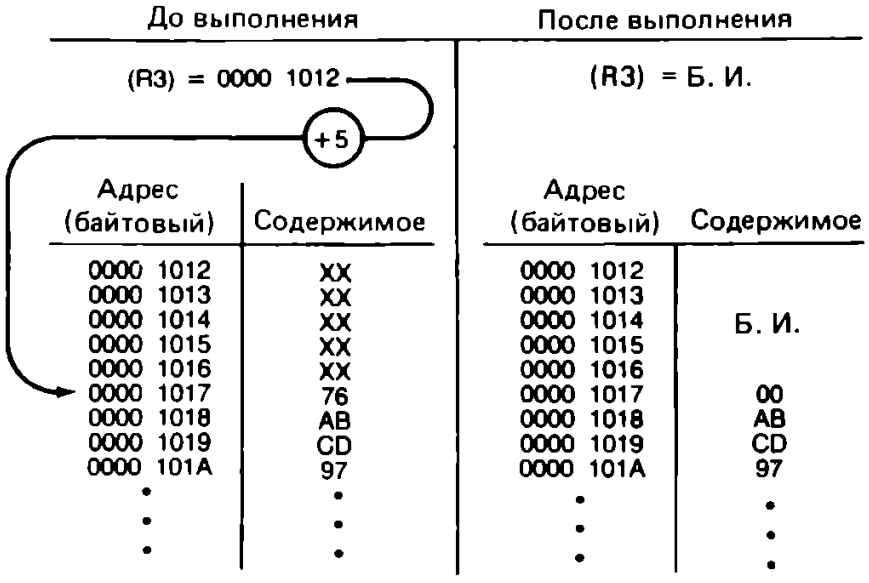
Карта памяти:



Режим сдвига. Этот режим адресации аналогичен индексному режиму в системе PDP-11. *Пример:*

CLRB 5(R3)

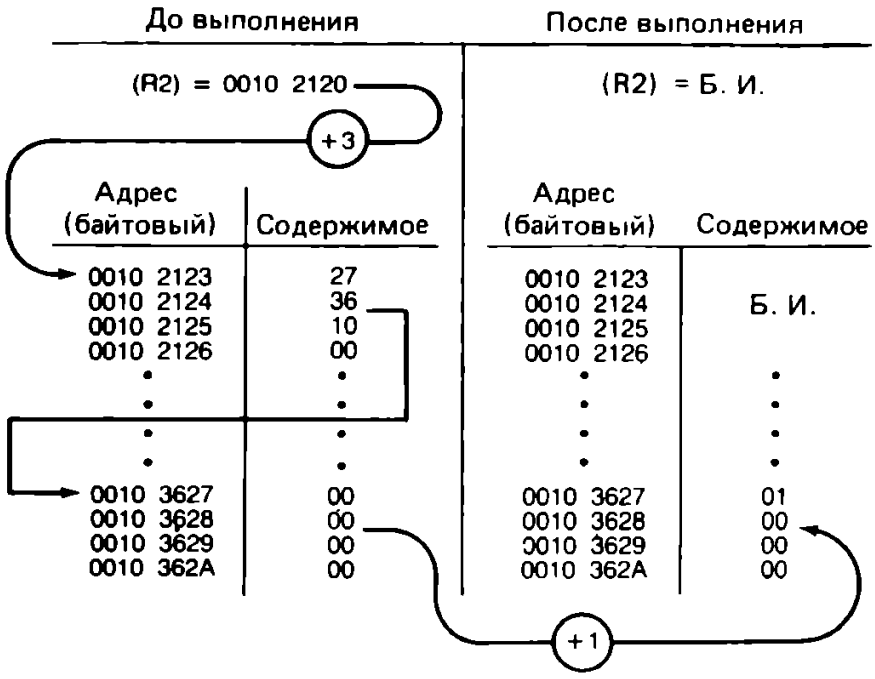
Карта памяти:



Косвенный режим сдвига. Этот режим аналогичен косвенному индексному режиму ЭВМ в PDP-11. *Пример:*

INCW @3(R2)

Карта памяти:



Замечание. В двух последних режимах программист имеет возможность использовать специальные символы, чтобы информировать ассемблер, что значение сдвига должно помещаться в байт, слово или длинное слово:

B ↑ D(R*n*) ; приводит к сдвигу в байте
W ↑ D(R*n*) ; приводит к сдвигу в слове
L ↑ D(R*n*) ; приводит к сдвигу в длинном слове

Индексный режим. К сожалению, название этого режима вносит путаницу, поскольку он отличается от индексного режима, определенного в системе PDP-11. Допустимые форматы для этого режима описаны в табл. 10.3.

Таблица 10.3. **Адресация индексного режима**[1]

Режим	Нотация ассемблера
Регистровый косвенный индексный	(R*n*) [R*x*]
Индексный с автоувеличением	(R*n*) + [R*x*]
Непосредственный индексный	# константа [R*x*], что распознается ассемблером, но не полезно в общем случае. Адрес операнда не зависит от значения константы
Косвенный индексный с автоувеличением	@(R*n*) + [R*x*]
Абсолютный индексный	@# адрес [R*x*]
Индексный с автоуменьшением	--(R*n*) [R*x*]
Индексный со смещением в байте, слове или длинном слове	B ↑ D(R*n*) [R*x*]
	W ↑ D(R*n*) [R*x*]
	L ↑ D(R*n*) [R*x*]
Косвенный индексный со смещением в байте, слове или длинном слове	@B ↑ D(R*n*) [R*x*]
	@W ↑ D(R*n*) [R*x*]
	@L ↑ D(R*n*) [R*x*]

[1] Таблица перепечатана с любезного разрешения фирмы DEC.

Пример 1:

```
CLRW   (R2) [R5]
```

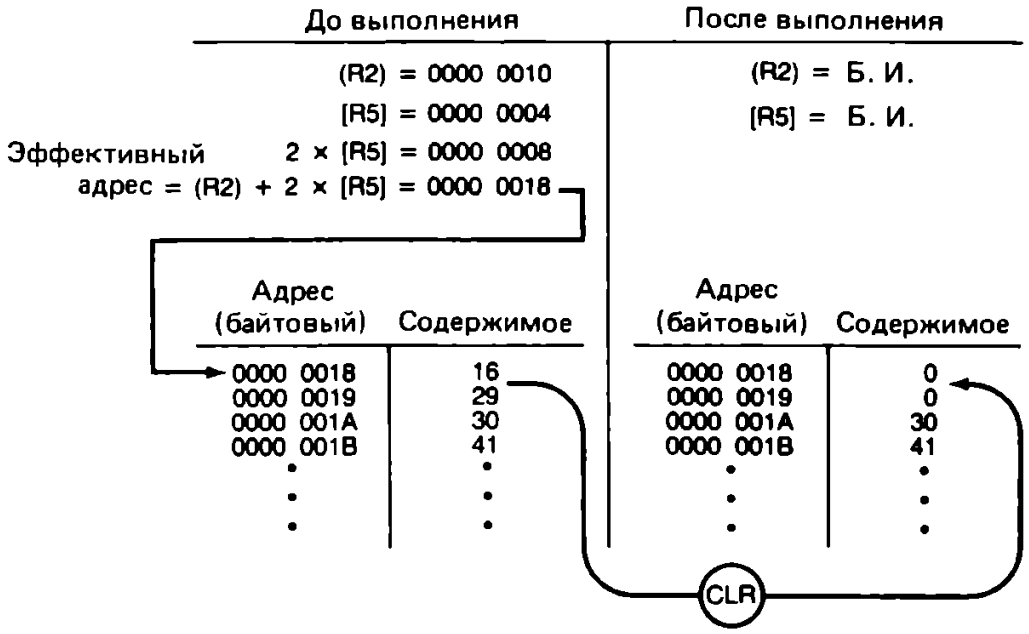
Здесь эффективный адрес равен x * [R5] + (R2), где

[R5] – содержимое регистра R5;

(R2) – содержимое регистра R2

$x =$ { 1 для байта; 2 для слова; 4 для длинного слова, для формата с плавающей точкой; 8 для четырехкратного слова, плавающего формата D, плавающего формата G; 16 для восьмикратного слова, плавающего формата H }

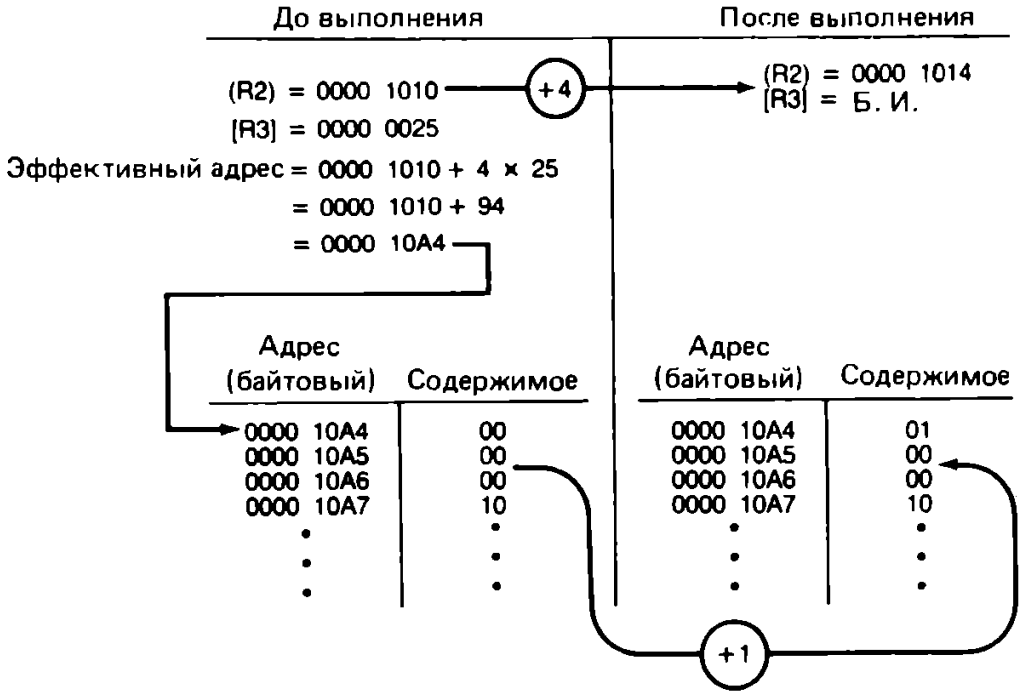
Карта памяти:



Пример 2:

```
INCL    (R2)+[R3]          ;РЕЖИМЫ АВТОУВЕЛИЧЕНИЯ И ИНДЕКСНЫЙ
```

Карта памяти:



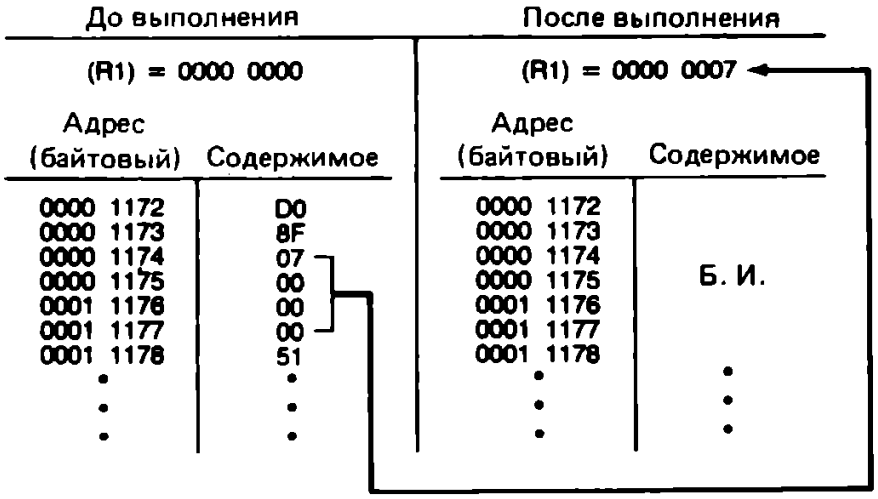
РЕЖИМЫ АДРЕСАЦИИ С РЕГИСТРОМ PC

Подобно режимам адресации в системе PDP-11 существует класс режимов адресации с вовлечением регистра PC. Но в системе VAX-11 в качестве регистра PC используется регистр R15. И так же, как в ЭВМ PDP-11 ЭВМ, VAX-11 имеет непосредственный режим, абсолютный режим, относительный режим и относительный косвенный режим.

Непосредственный режим. *Пример:*

```
A:      MOVL    #7,R1    ;ПУСТЬ A = 00001172
```

Эта инструкция просто переносит целое число 7 в длинное слово регистра R1. Это целое число занимает 32-битовое слово памяти (4 байта), непосредственно следующее за инструкцией:

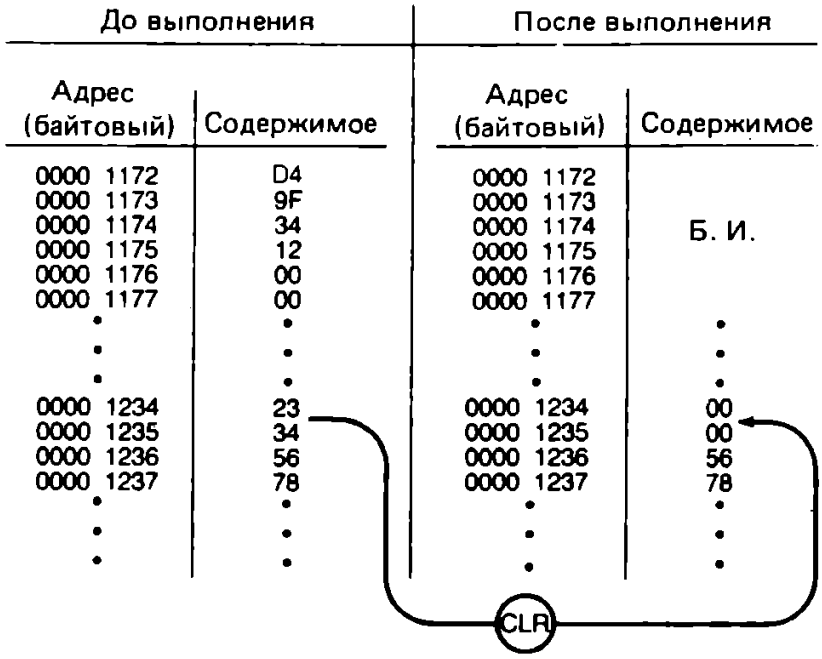


где D0 – код операции для инструкции MOVL; F – регистр R15 (или регистр RC); 8 – номер непосредственного режима адресации; 0000 0007 – целое число 7 в 32-битовом длинном слове; 1 – регистр R1; 5 – режим адресации номер 5, т. е. регистровый режим.

Абсолютный режим. *Пример:*

```
A:      CLRW    @#^X1234        ;ПУСТЬ A = 0000 1172
```

Эта инструкция очищает содержимое ячейки 0000 1234. Нотация ↑ X означает, что следующее число является шестнадцатеричным. Это необходимо потому, что ассемблер VAX-11 будет интерпретировать неспецифицированное число как десятичное.

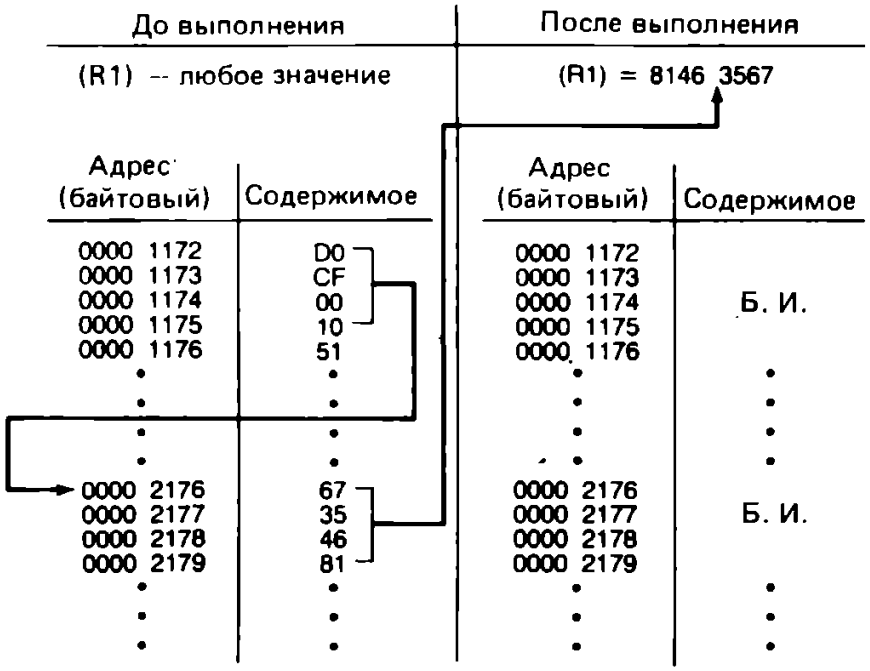


где D4 – код операции инструкции CLRW; F – регистр PC (или регистр R15); 9 – кодовое число для абсолютного режима адресации; 0000 1234 – абсолютный адрес операнда.

Относительный режим. *Пример:*

```
A:      MOVL    ^X2176,R1       ;ПУСТЬ A = 0000 1172
```

Эта инструкция иллюстрирует режим относительной адресации. Адресом назначения здесь является адрес 2176, скорректированное содержимое регистра PC равно 1176 и, следовательно, относительное расстояние между скорректированным содержимым регистра PC и адресом назначения равно 1000. Режим относительной адресации ЭВМ VAX-11 идентичен такому же режиму в системе PDP-11, но ассемблер системы VAX-11 выполняет дополнительную функцию: он следит за эффективностью использования пространства памяти для относительного расстояния. Другими словами, при коротких относительных расстояниях (⩽ FF) ассемблер отведет для значения относительного расстояния только однобайтовое пространство памяти, для средних расстояний (⩽ FFFF) – однословное пространство памяти и т. д.:

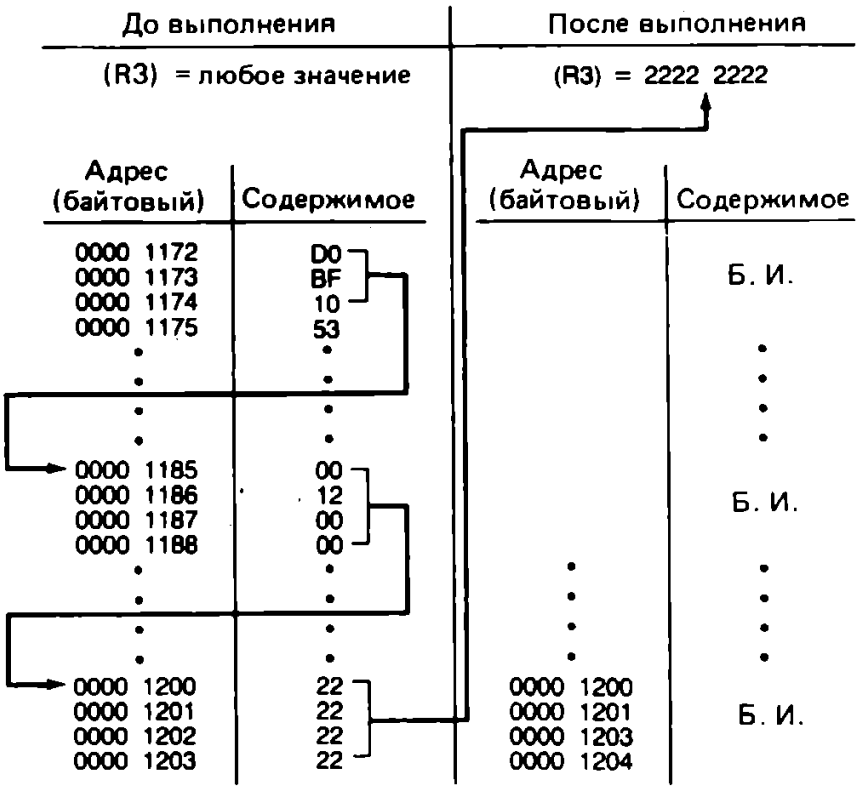


где D0 – код операции для инструкции MOVL; F – регистр PC (или регистр R15); C – код относительного расстояния, для которого требуется однословное пространство; 1000 – относительное расстояние, равное адресу назначения минус скорректированное содержимое регистра PC, 1 – R1; 5 – код для регистрового режима адресации.

Режим относительной косвенной адресации. *Пример:*

```
A:      MOVL    @^X1185,R3      ;ПУСТЬ A = 0000 1172
```

Этот режим такой же, как и для ЭВМ PDP-11, за исключением того, что для значения относительного расстояния отводится область памяти переменной длины:



где D0 – код операции для инструкции MOVL; F – регистр PC (или регистр R15); B – код для косвенного относительного расстояния, для которого требуется однобайтовая область; 10 – относительное расстояние; 3 – регистр R3; 5 – код для регистрового режима адресации.

Адресация ветвлений

Принцип адресации ветвлений одинаков и в системе PDP-11, и в системе VAX-11. Однако относительное расстояние в системе PDP-11 ограничено диапазоном от –256 до 254, в системе VAX-11 диапазон определяется либо одним байтом, либо одним 16-битовым словом. Для 16-битового слова мы располагаем диапазоном от –32768 до 32766. И ассемблер будет вычислять значение относительного расстояния и записывать его после кода операции. Например, инструкция BRB порождает байтовое относительное расстояние, а инструкция BRW – относительное расстояние в 16-битовом слове. То есть инструкция

```
A:   BRW  LOOP
```

осуществляет ветвление к метке LOOP, которая может находиться на расстоянии до 32766 байт в прямом направлении и до 32768 байт в обратном направлении от метки A.

10.4. МАКРОСЫ, ПОДПРОГРАММЫ И ПРОЦЕДУРЫ

МАКРОСЫ

Точно так же, как в системе PDP-11, в системе VAX-11 пользователь может создавать макроинструкции по следующему формату:

```
.MACRO  имя  аргумент 1, аргумент 2, . . . , ? L1, ?L2, . . .
.
.
тело макроса, состоящее из инструкций ЭВМ VAX-11
.
.
.ENDM  имя
```

где ?L1, ?L2, . . . – локальные адресные метки. Аналогично пользователи имеют возможность создавать условные макросы (см. гл. 6).

ПОДПРОГРАММЫ

Принцип использования подпрограмм, описанный в гл. 5 для системы PDP-11, применим и в системе VAX-11. То есть для использования подпрограмм в системе VAX-11 нужна пара инструкций для перехода к подпрограмме и возврата из нее, связывающая главную программу и подпрограмму. Кроме того, нужен хорошо определенный способ передачи параметров для пересылки данных в подпрограмму и из нее. При этом часто требуется сохранять и восстанавливать содержимое некоторых регистров общего назначения. Реализацию всех этих требований осуществляет программист. В системе VAX-11 программисты вместо использования в качестве связующей пары инструкций JSP, RTS могут использовать инструкцию BSBB (байтовое ветвление к подпрограмме) и инструкцию BSBW (словное ветвление к подпрограмме) для входа в подпрограмму в зависимости от относительного расстояния между вызывающей инструкцией и стартовым адресом вызываемой подпрограммы. Для возврата в системе VAX-11 программисты используют инструкцию RSB (возврат из подпрограммы). Ниже показан общий формат для входа в подпрограмму и возврата из нее:

```
;ГЛАВНАЯ ПРОГРАММА
START:  .
        .
        .
        BSBW    SUBR    ;ВЕТВЛЕНИЕ К ПОДПРОГРАММЕ, КОТОРАЯ
        .               ;МОЖЕТ БЫТЬ УДАЛЕНА ДО +/-32 КБАЙТ
        .
        .
SUBR:   .
        .
        .
        RSB
```

Когда ЦП выполняет инструкцию перехода к подпрограмме, он проталкивает в стек скорректированное содержимое регистра PC и помещает адресное значение метки SUBR в регистр PC. При выполнении инструкции возврата из стека выталкивается скорректированное содержимое регистра PC и помещается снова в регистр PC.

ПРОЦЕДУРЫ

Для пользователей ЭВМ PDP-11 термин "процедура" может быть новым. Фактически процедура – это подпрограмма, но с более строгими правилами оформления. Она создается для удобства пользователя. Вспомним, что для написания подпрограммы программист обязан выбрать и реализовать подходящий способ передачи параметров. За сохранение и восстановление содержимого регистров общего назначения с помощью стековой памяти (если это необходимо) также отвечает программист. Эти требования хотя и оправданы, но отнимают слишком много времени у программиста, особенно если сохранение и восстановление содержимого регистров требуются часто.

Обычно сохраненное содержимое регистров перемешивается с другой информацией в одном блоке стековой памяти, относящемуся к одному указателю стека. Процесс может быть заметно упрощен, если иметь указатель, специально предназначенный для процесса сохранения, и указатель для транспортировки данных. Эта идея и реализована в ЭВМ VAX-11 и названа **процедурой**. Теперь пользователь может по выбору писать или подпрограммы, или процедуры.

В ЭВМ VAX-11 регистр R12 называется **указателем аргументов (УА)** и используется для передачи параметров; регистр R13 называется **указателем кадра (УК)** и использует-

ся как базовый регистр, указывающий на базовый или начальный адрес блока стековой памяти, зарезервированного исключительно с целью сохранения ключевой информации главной программы: PSL, скорректированного содержимого регистра PC, содержимого всех регистров общего назначения и т. п. Таким образом, пользователь всегда может найти информацию, обращаясь к регистру УК вместо системного указателя стека, поскольку содержимое регистра SP обычно со временем изменяется, тогда как содержимое регистра УК остается фиксированным. Давайте посмотрим, как вышеописанный подход отражается на формате вызова процедуры.

В системе VAX-11 пользователю для связывания главной программы и процедуры даются два набора пар инструкций: CALLG и RET; CALLS и RET. Инструкция CALLG позволяет главной программе осуществить автоматическую передачу процедуре начального адреса того массива данных, с которым должна работать вызываемая процедура. Инструкция CALLS требует, чтобы главная программа сначала протолкнула данные в системный стек, после чего процедура сможет работать с этими данными в стековой памяти.

Пример 1:

```
;ГЛАВНАЯ ПРОГРАММА
ARRAY:  .LONG   N               ;РАЗМЕР МАССИВА ДАННЫХ В ДЛИННОМ СЛОВЕ
        .LONG   D1,D2,...,DN    ;ДИРЕКТИВА .LONG АНАЛОГИЧНА ДИРЕКТИВЕ
                                ;.WORD, НО С ДАННЫМИ В ДЛИННЫХ СЛОВАХ
START:  .
        .
        .
        CALLG   ARRAY,PROC      ;ВЫЗОВ ПРОЦЕДУРЫ
        .
        .
        .
PROC:   .WORD   ^M<R0,R1,...>   ;ФОРМАТ ДЛЯ СОХРАНЕНИЯ РЕГИСТРОВ
        .
        .
        RET
        .END    START
```

Обратите внимание, что структура массива данных должна быть такой, чтобы в первом длинном слове содержался размер массива. При выполнении инструкции CALLG ARRAY, PROC аппаратно автоматически осуществляется последовательность шагов, помещающих адресное значение #ARRAY в регистр УА (или R12), а адресное значение начального адреса процедуры #PROC – в регистр PC. Помимо этого она сохраняет скорректированное содержимое регистров PC, PSL и другую информацию в блоке стековой памяти, определяемом регистром УК (или R13).

Обратите внимание, что первая строка процедуры задает регистровую маску, которая будет интерпретироваться аппаратурой как желание пользователя сохранить содержимое регистров, перечисленных внутри угловых скобок. Содержимое перечисленных регистров проталкивается в стековый блок, определяемый регистром УК (или R13). Инструкция RET в конце процедуры восстанавливает содержимое регистров и возвращает управление главной программе. Инструкция CALLG имеет два операнда – один для массива данных, другой для стартового адреса процедуры.

Пример 2:

```
;ГЛАВНАЯ ПРОГРАММА
ARRAY:  .LONG   D1,D2,...,DN
START:  PUSHL   #DN
        .
        .
        .
        PUSHL   #D2
        PUSHL   #D1
```

```
        CALLS   #N,PROC         ;ВЫЗЫВАЮЩАЯ ИНСТРУКЦИЯ
        .
        .
        .
PROC:   .WORD   ^M<R0,R1,...>
        .
        .
        RET
        .END
```

В этом примере мы воспользовались инструкцией CALLS, так что перед вызовом процедуры массив данных должен быть протолкнут в стек. Обратите внимание, что эта инструкция имеет два операнда. Второй операнд используется так же, как в инструкции CALLG, а первый операнд представляет собой число элементов данных, протолкнутых в стек. При выполнении инструкции CALLS в стек проталкивается первый операнд, *#n*, регистры, скорректированное содержимое регистров PC, PSL и т. п. Содержимое регистров УА и УК фиксировано и используется так же, как в инструкции CALLG. Однако инструкция RET помимо обычных функций еще удаляет аргументы, протолкнутые в стек.

В этом примере можно было бы просто протолкнуть в стек адрес массива данных и размер массива, а процедура для извлечения данных может воспользоваться этой адресной информацией с помощью метода косвенной адресации. При этом мы протолкнем в стек два длинных слова, а инструкция вызова будет следующей: CALLS #2, PROC.

10.5. ПРЕРЫВАНИЯ И ИСКЛЮЧЕНИЯ

Как описано в гл. 8, в системе PDP-11 мы разделили процессы прерывания на два класса: аппаратные (внешние) прерывания и программные (внутренние) прерывания. Строго говоря, программные прерывания, такие как инструкция TRAP и т. п., нельзя называть прерываниями. В системе VAX-11 программные прерывания называются **исключениями**. При аппаратных (внешних) прерываниях источниками прерываний обычно являются периферийные устройства, функционально независимые от текущей работающей программы, и они требуют обслуживания, которое, вероятно, не зависит от главной программы. Процессы исключения в определенной степени относятся к главной программе. Например, если в результате выполнения инструкции главной программы возникает ошибка, то нормальная последовательность главной программы будет остановлена и ЦП выберет для выполнения подпрограмму для обработки исключения.

В системе VAX-11 — 32 уровня приоритетов прерывания. Уровни определяются битами $b_{21}, \ldots, b_{16}$ в PSL, причем нулевому уровню соответствует самый низкий приоритет. Для исключений есть три типа: ловушки, ошибки и выбросы. Подробную информацию можно найти в фирменном справочнике по архитектуре ЭВМ VAX-11.

10.6. ЛИТЕРАТУРА ДЛЯ ДАЛЬНЕЙШЕГО ЧТЕНИЯ

Lemone, Karen A., and Martin E. Kaliski. *Assembly Language Programming for the VAX-11*. Boston: Little, Brown, 1983.

Levy, Henry M., and Richard H. Eckhouse, Jr. *Computer Programming and Architecture: The VAX-11*. Bedford, Mass.: Digital Press, 1980.

VAX Hardware Handbook. Digital Equipment Corporation (1981).

VAX Architecture Handbook. Digital Equipment Corporation (1981).

VAX Software Handbook. Digital Equipment Corporation (1981).

10.7. УПРАЖНЕНИЯ

1. Выполните упр. 4, 11 из гл. 5 (подпрограммы), но используйте язык ассемблера системы VAX-11 для различных типов данных.
2. Выполните упр. 3, 4 из гл. 6 (макросы), но используйте язык ассемблера системы VAX-11.

ГЛАВА 11
ЛАБОРАТОРНЫЕ УПРАЖНЕНИЯ

11.1. ВВЕДЕНИЕ

Опыт преподавания показывает, что наиболее важным и эффективным путем изучения программирования на языке ассемблера является непосредственное получение опыта в лабораторных условиях. Как отмечалось в гл. 1, существует аналогия между обучением плаванию и программированию на ЭВМ. Никто еще не стал хорошим пловцом, прочитав множество книг, но не окунувшись в воду, чтобы на практике отработать то, о чем он прочитал. Поэтому, если вы собираетесь освоить программирование на языке ассемблера, практика на ЭВМ для вас необходима. Кроме того, лекции, лабораторные работы и экзамены — важные средства обучения любой научной или технической дисциплине. Следует подчеркнуть и важность экзаменов, дополняющих лабораторные упражнения. В этой главе мы представляем некоторые задачи, использованные для лабораторных упражнений.

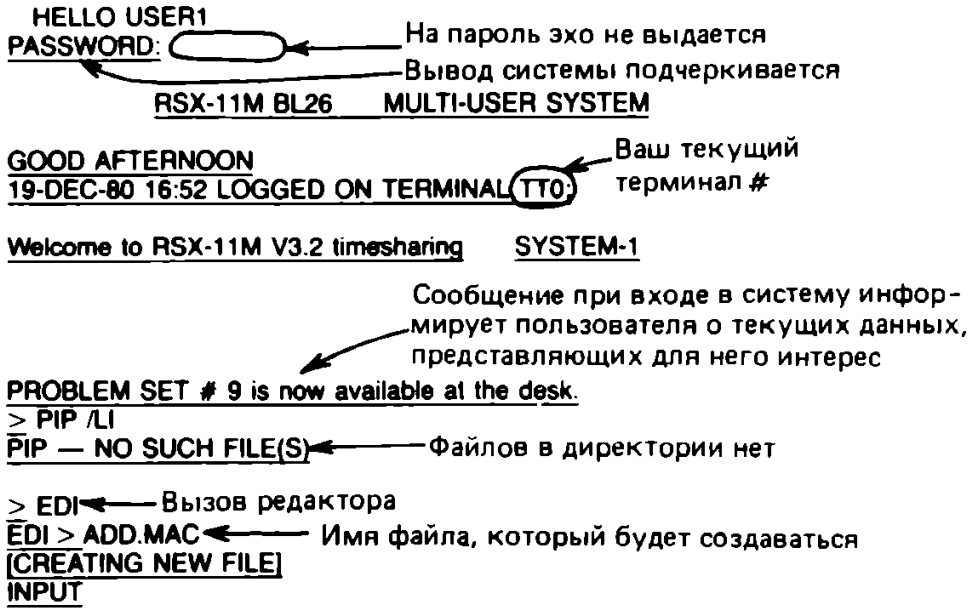
Поскольку разные вычислительные системы могут отличаться друг от друга, то не все описанное ниже окажется подходящим для всякой системы. Однако с небольшими модификациями эти задачи могут оказаться полезными для любого читателя.

11.2. ПРИМЕРЫ ЗАДАЧ ДЛЯ ЛАБОРАТОРНЫХ УПРАЖНЕНИЙ

Далее следует набор задач для системы PDP-11. В них используются все разнообразные возможности языка ассемблера; они даются в порядке возрастания трудности. Разрешив все задачи, студент использует все, что он изучил в аудитории.

ЗАДАЧА 1

Это простое упражнение должно помочь студенту освоить ввод в ЭВМ текста программы с имитацией некоторых типичных ошибок человека, которые могут быть исправлены с помощью системной программы редактора текста. На рис. 11.1 показан



```
;THIS IS A VERY SIMPLE PDP-11 PROGRAM
;WHICH ADDS TWO NUMBERS AND TESTS THE RESULT.
;LITU^UTIL^IT IS USED TO EM^ME^DEMONSTRATE THE SIMPLE
;USE OF THE ^U
USE OF THE EDITOR AND ASSEMBLER.
;
        .TITLE  ADD
A:      .BLKW           ;WORD RESERVED FOR A
B;^;^:  .WORD   299.    DECIMAL NUMBER
C:      .WORD   -476    ;OCTAL #
START:  MOVE    C,R0    ;R0 IS NAME FOR REGISTER 0
        ADD     B,R0    ;FORM SUM
        BMI     NEG
        HALT            ;HALT HERE IF POS
NEG:    HALT            ;HALT HERE IF NEG
        .END    START
```

4 нажатия клавиши стирания для устранения ошибок

Комбинация CTRL/U стирает строку

Текст, вводимый пользователем в файл

Режим ввода текста заканчивается нажатием на клавишу возврата каретки

Команда для распечатки содержимого буфера страницы

```
*TOP
*LI
```

```
; THIS IS A VERY SIMPLE PDP-11 PROGRAM
;WHICH ADDS TWO NUMBERS AND TESTS THE RESULT.
;IT IS USED TO DEMONSTRATE THE SIMPLE
USE OF THE EDITOR AND ASSEMBLER.
;
        .TITLE  ADD
A:      .BLKW           ;WORD RESERVED FOR A
B:      .WORD   299.    DECIMAL NUMBER
C:      .WORD   -476    ;OCTAL #
START:  MOVE    C,R0    ;R0 IS NAME FOR REGISTER 0
        ADD     B,R0    ;FORM SUM
        BMI     NEG
        HALT            ;HALT HERE IF POS
NEG:    HALT            ;HALT HERE IF NEG
        .END    START
```

```
*TOP
*LOC USE
;IT IS USED TO DEMONSTRATE THE SIMPLE
USE OF THE EDITOR AND ASSEMBLER.
*C/U/;U
;USE OF THE EDITOR AND ASSEMBLER
*EXIT
[EXIT]
```

Клавиша возврата каретки заставляет редактор продвинуться на одну строку

Команда замены исправляет отсутствие точки с запятой перед комментарием

Исправленная строка

Эта команда завершает работу редактора и закрывает файл

```
>PIP /LI

DIRECTORY DK0:[100,1]
19-DEC-80 16:59

ADD.MAC;1       1.          19-DEC-80 16:52
TOTAL OF 1./5. BLOCKS IN 1. FILE
```

Размер в блоках

Дата и время создания

Имя, тип и версия файла

Нажатия клавиши стирания для исправления ошибок при печатании текста

```
>MAC ADD.TI: = MAC^CAM^ADD
```

Файл листинга должен выводиться на терминал

Результат работы ассемблера должен выводиться в файл с именем ADD.OBJ

Листинг

```
 ADD        MACRO  M113  19-DEC-80 16:59  PAGE 1


                1
                2       ;THIS IS A VERY SIMPLE PDP-11 PROGRAM
SULT            ;WHICH ADDS TWO NUMBERS AND TESTS THE RE
```

Остаток от предыдущей строки, поскольку она слишком длинная

```
        3                                ;IT IS USED TO DEMONSTRATE THE SIMPLE
        4                                ;USE OF THE EDITOR AND ASSEMBLER.
        5                                ;
        6                                        .TITLE  ADD
        7  000000                        A:      .BLKW           ;WORD RESERVED F
OR A
(A)     8  000002  000453  000000G  000000G  B:      .WORD   299.    DECIMAL NUMBER
        9  000010  177302                   C:      .WORD   -476    ;OCTAL #
(A)    10  000012  000000G  000010'  000000  START:  MOVE    C,R0    ;R0 IS NAME FOR
REGISTER 0
       11  000020  066700  177756                   ADD     B,R0    ;FORM SUM
       12  000024  100401                           BMI     NEG
       13  000026  000000                           HALT            ;HALT HERE IF PO
S
       14  000030  000000                   NEG:    HALT            ;HALT HERE IF NE
G
       15          000012'                          END     START
```

Ошибки: забыта точка с запятой перед комментарием; вместо инструкции MOV использована инструкция MOVE

Из-за неправильного написания инструкции MOV ассемблер пытается вставить метку

Из комментария без точки с запятой

```
ADD     MACRO M1113  19-DEC-80 16:59  PAGE 1-1
SYMBOL TABLE

A       000000R          C       000010R        MOVE = ****** GX        NUMBER =
****** GX         START   000012R
B       000002R          DECIMA = ****** GX     NEG     000030R

.ABS.   000000      000
        000032      001
ERRORS DETECTED: 2
ERRORS DETECTED: 2

VIRTUAL MEMORY USED: 79 WORDS ( 1 PAGES)
DYNAMIC MEMORY: 3012 WORDS ( 11 PAGES)
ELAPSED TIME: 00:00:18
ADD,TI: = ADD
ADD,TI: = ADD
```

Все приведенное выше — это вывод, порожденный ЭВМ

Теперь мы вызываем редактор EDI для исправления своих ошибок

```
>EDI ADD.MAC
[00015 LINES READ IN]
[PAGE 1]
*LI

;THIS IS A VERY SIMPLE PDP-11 PROGRAM
;WHICH ADDS TWO NUMBERS AND TESTS THE RESULT.
;IT IS USED TO DEMONSTRATE THE SIMPLE
;USE OF THE EDITOR AND ASSEMBLER.
;
        .TITLE  ADD
A:      .BLKW           ;WORD RESERVED FOR A
B:      .WORD   299.    DECIMAL NUMBER
C:      .WORD   -476    ;OCTAL #
START:  MOVE    C,R0    ;R0 IS NAME FOR REGISTER 0
        ADD     B,R0    ;FORM SUM
        BMI     NEG
        HALT            ;HALT HERE IF POS
NEG:    HALT            ;HALT HERE IF NEG
        .END    START
*LOC B
B:      .WORD   299.    DECIMAL NUMBER
*C DEC ;DEC
B:      .WORD   299.    ;DECIMAL NUMBER
*LOC MOVE
START:  MOVE    C,R0    ;RO IS NAME FOR REGISTER 0
*C MOVE;MOV
START:  MOV     C,R0    ;R0 IS NAME FOR REGISTER 0
*EX
[EXIT]
```

Команда для поиска строки с первой ошибкой

Команда замены

Еще одна команда замены

Редактирование закончено

```
>PIP /LI                                  Первый файл программы (с ошибками)

DIRECTORY DK0:[100,1]                     Выходной файл ассемблера после ассемблирования
19-DEC-80 17:01                           с ошибками
                                                  Исправленный файл
ADD.MAC;1          1.      19-DEC-80 16:52
ADD.OBJ;1          1.      19-DEC-80 16:59
ADD.MAC;2          1.      19-DEC-80 17:00

TOTAL OF 3. 11. BLOCKS IN 3. FILES


                                  Ассемблирование самой последней
>MAC ADD,TI: = ADD                версии файла ADD.MAC

                                  Ниже приведен вывод, порожденный ЭВМ
ADD     MACRO M1113  19-DEC-80 17:02  PAGE 1

        1                                 ;THIS IS A VERY SIMPLE PDP-11 PROGRAM
        2                                 ;WHICH ADDS TWO NUMBERS AND TESTS THE RE
SULT.
        3                                 ;IT IS USED TO DEMONSTRATE THE SIMPLE
        4                                 ;USE OF THE EDITOR AND ASSEMBLER.
        5                                 ;
        6                                       .TITLE  ADD
        7  000000                         A:    .BLKW           ;WORD RESERVED F
OR A
        8  000002   000453                B:    .WORD   299     ;DECIMAL NUMBER
        9  000004   177302                C:    .WORD   -476    ;OCTAL #
       10  000006   016700   177772      START: MOV    C,R0    ;R0 IS NAME FOR
REGISTER 0
       11  000012   066700   177764            ADD    B,R0    ;FORM SUM
       12  000016   100401                     BMI    NEG
       13  000020   000000                     HALT           ;HALT HERE IF PO
S
       14  000022   000000                NEG:  HALT           ;HALT HERE IF NE
G
       15           000006'                    END    START

   Относительный адрес первого слова инструкции
   Значение
   Значение второго слова инструкции (если оно есть)
   Метка, с которой начнется выполнение программы

ADD     MACRO M1113  19-DEC-80 17:02  PAGE 1-1
SYMBOL TABLE

A       000000R          B       000002R        C       000004R        NEG
000022R          START   000006R

  ABS.  000000     000
        000024     001
ERRORS DETECTED: 0

VIRTUAL MEMORY USED: 59 WORDS ( 1 PAGES)
DYNAMIC MEMORY: 3012 WORDS ( 11 PAGES)
ELAPSED TIME: 00:00:17
ADD,TI: = ADD                     Выше приведен вывод, порожденный
>TKB                              программой MAC
TKB > ADD,TI: = ADD
TKB >//


ADD.TSK;1     MEMORY ALLOCATION MAP   TKB        PAGE 1
                                19-DEC-80   17:03

PARTITION NAME    : GEN
IDENTIFICATION    :
TASK UIC          : [100,1]
```

```
STACK       LIMITS: 000172 001171 001000 00512.
PRG XFR ADDRESS: 001200
TOTAL ADDRESS WINDOWS: 1.
TASK IMAGE SIZE      : 352. WORDS
TASK ADDRESS LIMITS: 000000 001217
R-W DISK BLK LIMITS  : 000002 000003 000002 00002.

*** ROOT SEGMENT: ADD

R/W MEM LIMITS: 000000 001217 001220 00656.
DISK BLK LIMITS: 000002 000003 000002 00002.

MEMORY ALLOCATION SYNOPSIS:

SECTION                                    TITLE  INDENT  FILE
  BLK.:(RW,I,LCL,REL,CON)  001172 000024 00020.
                           001172 000024 00020.  ADD          ADD.OBJ;2
        Начальный адрес программы
                                  Число слов в восьмеричном и десятичном представлении
                                                  Имя из оператора .TITLE
***  TASK BUILDER STATISTICS:

     TOTAL WORK FILE REFERENCES: 131.
     WORK FILE READS:  0.
     WORK FILE WRITES: 0.
     SIZE OF CORE POOL:6280. WORDS (24. PAGES)
     SIZE OF WORK FILE: 768. WORDS (3. PAGES)

     ELAPSED TIME:00:00:13
>PIP /LI

DIRECTORY DK0:[100,1]
19-DEC-80 17:03

ADD.MAC;1       1.          19-DEC-80 16:52
ADD.OBJ;1       1.          19-DEC-80 16:59
ADD.MAC;2       1.          19-DEC-80 17:00
ADD.OBJ;2       1.          19-DEC-80 17:02
ADD.TSK;1       4.      C   19-DEC-80 17:03

TOTAL OF 8./16. BLOCKS IN 5. FILES


>PIP *.*/PU
>PIP /LI

DIRECTORY DK0:[100,1]
19-DEC-80 17:04

ADD.MAC;2       1.          19-DEC-80 17:00
ADD.OBJ;2       1.          19-DEC-80 17:02
ADD.TSK;1       4.      C   19-DEC-80 17:03

TOTAL OF 6./10. BLOCKS IN 3. FILES
```

Выше показан вывод карты памяти, составленной построителем задач

Первая версия

Исправленная версия

```
>LGO                    Первый метод: загрузить и запустить
LGO > ADD               Нужно задать имя файла, т. е. ADD, ADD.TSK и т. п.
[Выполнение программы]
001216(CR)              Инструкция HALT останавливает ЦП в режиме отладчика ODT
@R0/177755(CR)          Проверка содержимого регистра R0 для нахождения результата
@165000G(CR)            Возврат микроЭВМ LSI-11 в терминальный режим
>GET
GET > ADD
[LOADED]                Сообщение о том, что программа загружена
^Z
>(BREAK)                Клавиша BREAK заставляет ЭВМ перейти в режим отладчика
  164060(CR)            Адрес ячейки, где прервано выполнение программы
```

```
@1172/000000(LF)     Базовым адресом для всех программ является адрес 1172
001174/000453(LF)    Содержимое загруженной программы в восьмеричном представлении
001176/177302(LF)
001200/016700(LF)
001202/177772(LF)
001204/066700(LF)
001206/177764(LF)
001210/100401(LF)
001212/000000(LF)
001214/000000(CR)
@1200G               Выполнить программу с адреса первой инструкции
001216(CR)           Инструкция HALT останавливает программу
                     и приводит к отображению содержимого регистра PC
@R0/177755(CR)       Проверка содержимого регистра R0, в котором
                     должен быть результат
@165000G(CR)         Возврат к терминальному режиму
>                    Подсказка монитора MCR
```

Рис. 11.1. Подготовка программы (упражнение)

реальный процесс интерактивного взаимодействия программиста и системы PDP-11 по созданию нового файла и использованию редактора текста для ревизии содержимого файла до тех пор, пока не будут исправлены все ошибки редактирования. Чтобы диалог человека и ЭВМ стал более наглядным, все ответы ЭВМ подчеркнуты и даны комментарии. Давайте последовательно изучим это упражнение. Сначала мы проводим диалог для входа в систему и получаем подсказку монитора MCR; затем используем команду PIP/LI, чтобы посмотреть, есть ли на диске уже созданные файлы. В данном случае мы получаем ответ, что таких файлов нет.

Затем мы вызываем редактор текста EDI, чтобы он помог нам создать программу, которая будет складывать два заданных числа и проверять результат. Далее следует интерактивный процесс использования команд редактора EDI для исправления типичных ошибок, возникающих при вводе текста с клавиатуры. Когда все ошибки исправлены, мы вызываем программу MAC, чтобы ассемблировать нашу программу.

Ассемблер отмечает ошибки в программе, что вынуждает нас для их исправления снова обратиться к редактору текста. После исправления всех ошибок ассемблер в конце концов сообщает, что число ошибок равно нулю, и выдает файл листинга программы. Тогда, чтобы создать выполнимую программу, которая должна быть загружена в оперативную память для выполнения, мы вызываем построитель задач или компоновщик TKB. Оставшаяся часть упражнения показывает, каким образом мы применяем команду загрузки и запуска LGO и используем отладчик ODT для просмотра деталей и результата выполнения программы.

Для удобства чтения в упражнении мы использовали символ ⊡ для идентификации главных частей процесса: редактирования текста, ассемблирования, построения задачи и выполнения.

ЗАДАЧА 2

Напишите программу, которая находит максимальное значение среди трех чисел A, B и C и помещает результат в переменную MAX. Ассемблируйте программу вручную, полагая, что первое ее слово окажется в ячейке 1172. Далее выполните следующее:

1. Используйте отладчик ODT для ввода машинных кодов в микроЭВМ LSI-11.
2. Введите следующие значения для A, B и C: (A) = –15; (B) = –15; (C) = –100.
3. Запустите программу.
4. Используйте отладчик ODT для проверки результата.
5. Дайте преподавателю проверить вашу работу.
6. Аккуратно запишите вариант программы на языке ассемблера и вариант ассемблированной программы.

ЗАДАЧА 3

Напишите программу, которая вычисляет среднее значение AV степеней M1, M2, HW и F, а затем останавливается. Вам нужно отвести 5 ячеек памяти в конце вашей программы (после инструкции останова HALT!) под эти величины плюс 1 ячейку памяти под среднее значение. (*Подсказка:* деление на 2 может выполняться с помощью инструкции ASR.) Ассемблируйте программу вручную, полагая, что первое слово попадает в ячейку 1172_8. Вы должны назначить адреса и тем 5 ячейкам памяти, которые зарезервированы под данные M1, M2, HW, F и AV. Далее выполните следующее:

1. Используйте отладчик ODT для ввода машинных кодов в микроЭВМ LSI-11.
2. Используйте отладчик ODT для ввода значений для данных M1, M2, HW, F: (M1) = = 89; (M2) = 62; (HW) = 92; (F) = 75.
3. Запустите программу.
4. Используйте отладчик ODT для проверки результатов.
5. Дайте товарищу проверить правильность работы вашей программы. Каждый член группы студентов должен иметь тетрадь с указанием своего имени, имен своих партнеров, учетной информацией, датой и номером, назначенным при верификации выполнения программы. Запишите в этой тетради свои решения.
6. Аккуратно запишите вариант программы на языке ассемблера и вариант ассемблированной программы.

ЗАДАЧА 4

Введите, ассемблируйте и запустите приведенную ниже программу. Добавьте, пожалуйста, комментарии (чтобы было видно, что вы понимаете, для чего предназначена программа). Вставьте свое имя в строку комментариев в начале программы. Далее выполните следующее:

1. Для редактирования программы с комментариями используйте системную программу EDI.
2. Для ассемблирования программы используйте системную программу MAC.
3. Для построения задачи используйте компоновщик TKB.
4. Для загрузки программы в ЭВМ используйте команду GET.
5. Для демонстрации программы выполните ее.
6. Каждый студент должен получить окончательный листинг программы и проверить его с помощью преподавателя.

```
START:  CLR     R0
        MOV     #A,R1
LOOP:   ADD     (R1)+,R0
        CMP     #A+18.,R1
        BFL     LOOP
        MOV     R0,SUM
        HALT
A:      .WORD   1,2,3,4,5,6,7,8.,9.,10.
SUM:    .BLKW
        .END
```

ЗАДАЧА 5

Напишите программу на языке ассемблера, которая будет извлекать число из ячейки STD и помещать его в ячейку REV в обратном порядке. Например, если в первой ячейке содержится число 0 101 001 101 110 110, то во второй должно быть число 0 110 111 011 001 010. Выполните следующее:

1. Введите программу в файл, используя редактор EDI.
2. Ассемблируйте программу, используя ассемблер MAC.
3. Постройте задачу с помощью компоновщика TKB.

4. Загрузите программу с ЭВМ командой GET.
5. Используйте отладчик ODT для загрузки значения в ячейку STD.
6. Выполните программу.
7. Используйте отладчик ODT для проверки ячейки REV.
8. Повторите шаги 5 – 7 для нескольких разных чисел.
9. Попросите преподавателя помочь проверить правильность работы программы.
10. Аккуратно перепишите листинг программы.

ЗАДАЧА 6

Напишите программу, которая подсчитывает число единичных бит в слове X. Результат должен помещаться в ячейку NUM. В качестве значения числа X используйте значение 170532, т. е. X: .WORD 170532.

ЗАДАЧА 7

Напишите программу на языке ассемблера для преобразования 16-битового числа, расположенного в ячейке NUMBER, в коды ASCII для знака и 5 восьмеричных цифр, представляющих число. Поместите полученные таким образом шесть байт в шесть последовательных байт памяти, начинающихся от правого байта слова OUTPUT. Символ знака должен быть в первом байте, во втором – символ первой печатаемой цифры. Например:

```
NUMBER: 173762  (1 111 011 111 110 010)          ;(-4016)
OUTPUT: 055     ;(-)
        060
        060
        064
        060
        061
        066
```

Затем выполните следующее:
1. Ассемблируйте программу.
2. Постройте задачу.
3. Загрузите эту задачу в ЭВМ.
4. Выполните программу в присутствии преподавателя.
5. Аккуратно перепишите листинг программы.

ЗАДАЧА 8

Напишите программу на языке ассемблера, которая будет сортировать до 30 чисел, расположенных в массиве ARRAY, используя "пузырьковую" сортировку. Числа должны быть упорядочены в порядке возрастания, так что ячейка ARRAY должна содержать наименьшее число, а ячейка ARRAY + n–2 – наибольшее число. (*Подсказка:* для обращения к массиву используйте режимы адресации 1 или 2; это упростит последующую работу.) Затем выполните следующее:
1. Ассемблируйте программу и аккуратно перепишите листинг.
2. Постройте задачу и загрузите программу.
3. Выполните программу в присутствии преподавателя. Воспользуйтесь следующей исходной информацией (все числа – восьмеричные):

```
N:      20
ARRAY:  -10
        126
        15
        -30
        0
        5000
        -256
        0
```

ЗАДАЧА 9

Цель этой проблемы состоит в том, чтобы дать вам возможность приобретения опыта по связыванию основной программы с подпрограммой и использованию системных макрокоманд. Выполните следующее:

1. Сделайте подпрограммой вашу программу сортировки, написанную для задачи 8. Вызов подпрограммы может выполняться с помощью следующей последовательности команд:

```
JSR     R5,SORT
.WORD   N       ;АДРЕС РАЗМЕРА МАССИВА В БАЙТАХ
.WORD   ARRAY   ;АДРЕС МАССИВА, ПОДЛЕЖАЩЕГО СОРТИРОВКЕ
----    -------- ;ИНСТРУКЦИЯ, КОТОРАЯ БУДЕТ ВЫПОЛНЕНА
                ;ПЕРВОЙ ПОСЛЕ ВОЗВРАТА
```

Метка SORT будет определена как глобальная и в подпрограмме, и в основной программе. Ассемблируйте вашу подпрограмму и получите ее листинг.

2. Напишите основную программу, которая:
   а) определяет массив UNSRT из 14_8 слов, содержащий числа 0, –15, –256, –4123, +27, –1777, 0, –27, –15, +7775, +27, –101;
   б) распечатывает эти числа, используя макрокоманду .PNUM;
   в) копирует массив UNSRT в другой массив –SRTD;
   г) вызывает подпрограмму SORT для сортировки массива SRTD;
   д) распечатывает отсортированный массив;
   е) возвращается к монитору MCR, используя макрокоманду .EXIT.
3. Ассемблируйте и получите листинг основной программы.
4. Скомпонуйте объектный модуль основной программы и объектный модуль подпрограммы, а также сделайте библиотеку макросов (MLIB).
5. Запустите программу в присутствии преподавателя и перепишите ее листинг.

ЗАДАЧА 10

Напишите программу, которая сливает два уже отсортированных массива в один массив так, чтобы все элементы полученного массива оказались правильно отсортированными. Поскольку оба входных массива уже отсортированы, то для этого необходима только сортировка начальных элементов обоих массивов, нахождение наименьшего элемента и выдача его на выход. Теперь надо продвинуть указатель того массива, который содержит выбранный элемент, снова сравнить элементы двух массивов и поместить наименьший элемент в выходной массив. Продолжать до тех пор, пока не опустеет один из входных массивов. Затем нужно переслать оставшуюся часть другого входного массива в выходной массив и остановиться.

Подпрограмма вызывается следующим образом:

```
JSR     R5,MRGE
.WORD   N1      ;АДРЕС РАЗМЕРА МАССИВА 1
.WORD   ARRAY1  ;АДРЕС ПЕРВОГО ВХОДНОГО МАССИВА
.WORD   N2      ;АДРЕС РАЗМЕРА МАССИВА 2
.WORD   ARRAY2  ;АДРЕС ВТОРОГО ВХОДНОГО МАССИВА
.WORD   N3      ;АДРЕС РАЗМЕРА ВЫХОДНОГО МАССИВА
                ;ПОДПРОГРАММЕ ПЕРЕСЫЛАЕТСЯ МАКСИМАЛЬНОЕ
                ;ЗНАЧЕНИЕ. ВОЗВРАЩАЕТСЯ ФАКТИЧЕСКОЕ
                ;ЗНАЧЕНИЕ (N1 + N2)
.WORD   ARRAY3  ;АДРЕС ВЫХОДНОГО МАССИВА
```

Выполните следующее:

1. Ассемблируйте подпрограмму и получите ее листинг.

2. Напишите основную программу, которая:

а) читает (используя макрокоманду .PNUM) значения N1, ARRAY1, N2, ARRAY2;

б) сортирует массивы ARRAY1 и ARRAY2;

в) распечатывает отсортированные массивы, используя макрокоманду .PNUM;

г) использует подпрограмму MERG для слияния массивов ARRAY1 и ARRAY2 в массив ARRAY3;

д) распечатывает массив, полученный после слияния двух массивов.

Замечание. Будьте внимательны и правильно используйте макрокоманду .PTEXT ля вывода титула и запроса ввода.

3. Ассемблируйте основную программу и постройте задачу вместе с подпрограммой.

4. Продемонстрируйте преподавателю работу программы.

ЗАДАЧА 11

Полезный алгоритм вычисления наибольшего общего делителя определяется екурсивно следующим образом:

```
GCD(X,Y) :=
        IF (X > Y) THEN GCD(X,Y);
        ELSE
        IF REM(Y,X) = 0 THEN X;
        ELSE GCD(REM(Y,X),X);
        ENDIF;
        ENDIF;
(REM(Y,X) - ЭТО ОСТАТОК, ОБРАЗУЮЩИЙСЯ ПРИ ДЕЛЕНИИ Y/X)
```

Напишите рекурсивную подпрограмму GCD для вычисления наибольшего общего делителя любых двух чисел X и Y и основную программу, которая принимает на входе значения чисел X и Y и затем вызывает подпрограмму GCD. Для передачи чисел X и Y воспользуйтесь стеком. Результат должен возвращаться в регистре R0.

Вам потребуется использовать два системных макроса — .RNUM и .PNUM, а также макрос .REM, имеющий три аргумента NUM, DEM и RSLT и вычисляющий остаток от деления NUM/DEM с помещением его в ячейку RSLT. Для аргументов макроса .REM нельзя использовать стек или регистр R0.

Основная программа использует макрос .RNUM для считывания значений X и Y с клавиатуры, после чего она вызывает подпрограмму GCD. Числа X и Y передаются подпрограмме через стек, куда они проталкиваются перед вызовом подпрограммы. Далее выполните следующее:

1. Ассемблируйте по отдельности подпрограмму GCD и основную программу, воспользовавшись директивой .GLOBL для объявления точки входа в подпрограмму.

2. Используйте построитель задач для связывания двух частей программы друг с другом.

3. Загрузите и выполните свою программу. Используйте следующие значения для X и Y: 45. и 45.; 97. и 12.; 32765. и 293. (все числа десятичные).

4. Проверьте правильность выполнения и запишите листинг программы.

ЗАДАЧА 12

Осуществляя ввод-вывод непосредственно, без прерываний, напишите следующие макросы:

а) макрос READ R, где R — регистр или метка ячейки, где должен разместиться один символ, считанный с клавиатуры;

б) макрос WRITE R, где R — регистр или метка ячейки, содержащей код символа, который должен быть послан на терминал.

1. Напишите, ассемблируйте и тестируйте программу, которая использует указанные макросы и делает следующее:

а) считывает символ;

б) выдает эхо на этот символ. Все управляющие символы (с кодами от 1 до 37) должны выдаваться как эхо в виде комбинации ↑ <буква>. Например, для EOT (код 004) эхо должно быть ↑ D;

в) останавливается после получения и выдачи соответствующего эха для кода 004.

2. Не используйте директиву .EXIT или макрос .PTEXT.

3. Выполните свою программу при участии преподавателя и запишите ее листинг.

ЗАДАЧА 13

1. Напишите программу, которая:

а) разрешает прерывания от приемника и очищает две ячейки памяти, CTR0 и CTR1, используемые в качестве счетчика;

б) прибавляет 1 к содержимому ячейки памяти CTR0.

в) всякий раз, когда происходит переполнение ячейки CTR0, увеличивает содержимое ячейки CTR1 и возвращается к шагу б).

2. Напишите подпрограмму обслуживания прерываний, которая всякий раз, когда происходит прерывание от клавиатуры:

а) печатает введенный символ (как в задаче 12);

б) распечатывает числа из ячеек памяти CTR1 и CTR0, используя макрос .PNUM1;

в) осуществляет возврат из прерывания к основной программе.

3. Ассемблируйте программу, постройте задачу, загрузите и выполните программу.

4. Проверьте работу программы и зафиксируйте листинг.

ЗАДАЧА 14

Вы должны написать небольшую программу терминального драйвера. Программа начнет работу с распечатки на терминале сообщения HELLO. Затем она будет ждать ввода с клавиатуры. Ваша программа должна выдавать эхо на каждый вводимый с клавиатуры символ. Если вводится непечатный символ (коды ASCII от 0 до 037), то никакого эха выдавать не надо.

Если вводится код возврата каретки, то нужно прекратить выдачу эха для текущей строки, перейти на следующую строку и снова напечатать HELLO и ждать последующего ввода.

Если вводится код CTRL/Z, то программа должна прекратить выдачу эха для текущей строки, перейти на следующую строку и напечатать BYE, после чего выйти в монитор MCR (с помощью директивы .EXIT).

Основная программа должна:

1. Инициализировать векторы прерываний, регистры состояния устройств и счетчик двойной точности (целое число без знака, занимающее два слова).

2. Ожидая прерывания от клавиатуры, постоянно увеличивать счетчик.

Подпрограмма обслуживания клавиатуры должна:

1. Принимать вводимый код и исследовать его:

а) если вводимый код – печатный, выдавать на него эхо;

б) если вводится CTRL/Z или возврат каретки, то перейти к соответствующей ячейке;

в) если не выполняются а) и б), то ничего не делать.

2. Распечатывать содержимое счетчика после вывода сообщения BYE. (Распечатывать надо сначала старшее слово, затем младшее слово, используя макрос .PNUM1.)

ЗАДАЧА 15

1. Напишите макрос MYPTX, который будет действовать аналогично системному макросу .PTEXT.

2. Используя свой макрос MYPTX, создайте макрос MYPNM, который заменит системный макрос .PNUM. За основу можно взять программу, написанную для задачи 7.

3. Напишите простую программу для распечатки нескольких строчек в коде ASCII и нескольких чисел со знаками "+" и "–" для демонстрации действия этих макросов.

4. Аккуратно перепишите свободный от ошибок листинг и с помощью преподавателя осуществите верификацию.

ЗАДАЧА 16

Напишите программу, работающую как терминальный драйвер, в микроЭВМ LSI-11. Она должна принимать ввод с терминала и пересылать его в ЭВМ PDP-11/34. В то же время она должна принимать данные от ЭВМ PDP-11/34 и посылать их на терминал. Проблема возникает из-за того, что связь между двумя ЭВМ осуществляется в четыре раза быстрее, чем связь микроЭВМ с терминалом. Создайте очередь или буфер для временного хранения, чтобы справиться с быстро приходящими данными без потерь.

Ваша программа должна состоять из трех частей: основной программы и двух подпрограмм обслуживания прерываний – одна для ввода с клавиатуры, другая для ввода данных с PDP-11/34.

Основная программа:

1. Следит, нет ли данных в очереди от PDP-11/34. Если есть, то программа выводит один байт на терминал, используя метод опроса.

2. Если PDP-11/34 был послан код CTRL/S и если очередь полна менее чем на 1/8, посылает PDP-11/34 код CTRL/O.

Подпрограмма обслуживания прерываний ввода от PDP-11/34 выполняет следующее:

1. Добавляет символ в очередь.

2. Если очередь полна более чем на 7/8, посылает PDP-11/34 код CTRL/S.

Подпрограмма обслуживания прерываний от клавиатуры считывает введенный байт и незамедлительно пересылает его PDP-11/34, используя метод опроса. Ассемблируйте программу, постройте задачу, загрузите и продемонстрируйте правильную работу вашей программы.

ПРИЛОЖЕНИЕ А

КОДЫ СИМВОЛЬНОГО НАБОРА ASCII

Коды символьного набора ASCII, в принципе, довольно просты, но значение первых 32 символов, таких как NUL, SOH или STX, сразу не ясно, да и большинства этих символов нет на клавиатуре терминала ЭВМ. Давайте внимательно рассмотрим символьный набор ASCII. Обратите внимание, что он кодируется 7-битовыми двоичными числами, поэтому в одном байте можно закодировать $2^7 = 128$ различных символов при использовании 7-битовых двоичных чисел. Из этих 128 двоичных чисел 96 чисел нужны для кодирования цифр, прописных и строчных букв, знаков препинания и математических знаков. Оставшиеся 32 кода используются как управляющие; они не являются печатными, но необходимы для организации взаимодействия пользователя с ЭВМ. Именно эти коды порождают столь большую путаницу. Это происходит, в частности, из-за того, что большинство клавиатур не располагает 128 клавишами; поэтому нужен какой-то компромисс. Обычно на большинстве клавиатур имеются явные клавиши возврата каретки (RETURN), перевода строки (LF) и клавиша расши-

рения кода (ESC). Оставшиеся из 32 управляющих кодов (с некоторыми исключениями) вводятся путем нажатия одновременно двух клавиш. Первая из них обычно имеет метку CONTROL или CTRL, ALT или CTL, а вторая – клавиша конкретной буквы (табл. А.1). Возможность генерации любых управляющих кодов ASCII одновременным нажатием двух клавиш решает проблему, связанную с отсутствием явных управляющих клавиш на клавиатуре. Некоторые управляющие коды требуют одновременного нажатия трех клавиш (см. таблицу).

Таблица А.1. **Символьный код ASCII (ч.1)**

Мнемоника	Двоичный код	7-битовый восьмеричный код	Шестнадцатеричный код	Ввод с клавиатуры	Примечание
NUL	00 00 00 00	000	00	CONT'L/SHIFT/P	Пусто, перевод формата
SOH	00 00 00 01	001	01	CONT'L/A	Начало заголовка: или начало сообщения (SOM)
STX	00 00 00 10	002	02	CONT'L/B	Начало текста; или конец адреса (EOA)
ETX	00 00 00 11	003	03	CONT'L/C	Конец текста; или конец сообщения (EOM)
EOT	00 00 01 00	004	04	CONT'L/ D	Конец передачи; останавливает телетайп
ENQ	00 00 01 01	005	05	CONT'L/E	Запрос; или WRU
ACK	00 00 01 10	006	06	CONT'L/ F	Подтверждение; или RU
BEL	00 00 01 11	007	07	CONT'L/ G	Включение звонка
BS	00 00 10 00	010	08	CONT'L/H	Возврат на шаг; или воздействие на формат (FEO)
HT	00 00 10 01	011	09	CONT'L/ I	Горизонтальная табуляция
LF	00 00 10 10	012	0A	CONT'L/ J	Перевод строки (новая строка)
VT	00 00 10 11	013	0B	CONT'L/K	Вертикальная табуляция (VTAB)
FF	00 00 11 00	014	0C	CONT'L/ L	Перевод формата на следующую страницу
CR	00 00 11 01	015	0D	CONT'L/M	Возврат каретки в начало строки
SO	00 00 11 10	016	0E	CONT'L/ N	Переключение красящей ленты на красный цвет
SI	00 00 11 11	017	0F	CONT'L/O	Переключение красящей ленты на черный цвет
DLE	00 01 00 00	020	10	CONT'L/P	Расширение кода строки данных
DC1	00 01 00 01	021	11	CONT'L/ Q	Управление устройством 1, включает передатчик (считыватель)
DC2	00 01 00 10	022	12	CONT'L/ R	Управление устройством 2, включает перфоратор или дополнительное устройство
DC3	00 01 00 11	023	13	CONT'L/ S	Управление устройством 3, выключает передатчик (считыватель)
DC4	00 01 01 00	024	14	CONT'L/T	Управление устройством 4, выключает перфоратор или дополнительное устройство

Мнемоника	Двоичный код	7-битовый восьмеричный код	Шестнадцатеричный код	Ввод с клавиатуры	Примечание
NAK	00 01 01 01	025	15	CONT'L/ U	Отрицательное подтверждение: или ошибка
SYN	00 01 01 10	026	16	CONT'L/ V	Синхронизация (SYNC)
ETB	00 01 01 11	027	17	CONT'L/ W	Конец блока передачи; или логический конец носителя информации (LEM)
CAN	00 01 10 00	030	18	CONT'L/X	Отмена
EM	00 01 10 01	031	19	CONT'L/ Y	Конец носителя информации
SUB	00 01 10 10	032	1A	CONT'L/ Z	Замена
ESC	00 01 10 11	033	1B	CONT'L/ SHIFT/K	Расширение ко
FS	00 01 11 00	034	1C	CONT'L/ SHIFT/ L	Разделитель файлов
GS	00 01 11 01	035	1D	CONT'L/ SHIFT/M	Разделитель групп
RS	00 01 11 10	036	1E	CONT'L/ SHIFT/ N	Разделитель записей
US	00 01 11 11	037	1F	CONT'L/ SHIFT/O	Разделитель единиц обмена

Существует еще одна проблема: каким образом должно выдаваться на экран ЭЛТ эхо для одновременного нажатия двух клавиш. Обычно это делается с помощью символа ↑ совместно с той буквой, клавиша которой была нажата. Но существует правило, позволяющее определить, какую нужно выводить букву для данного управляющего кода. Если, например, мы хотим ввести управляющий код EOT, то выбираем символ, код ASCII которого равен сумме числа 100_8 и кода EOT. В данном случае EOT = 004_8, после прибавления 100_8 получаем 104_8. Этот код соответствует букве D. Следовательно, если мы хотим ввести на клавиатуре код EOT, то должны нажать одновременно клавиши CTRL и D, тогда ЦП примет код EOT, т. е. 004_8. В качестве эха на ЭЛТ системное программное обеспечение пошлет два последовательных символа ^D. Для удобства таблица кодов ASCII разделена на две части – табл. А.1 и табл. А.2; в первой таблице представлены только 32 управляющих кода. В обеих таблицах в первой колонке дается мнемоника кода ASCII; во второй, третьей и четвертой колонках показаны соответствующие двоичные, 7-битовые восьмеричные и шестнадцатеричные числа. Поскольку программы ассемблера большинства вычислительных систем способны распознавать коды ASCII по тому представлению, которое дано в первой колонке, она называется "мнемоника". При двоичном представлении самый старший бит обычно резервируется для бита четности.

Таблица А.2. **Символьный код ASCII (ч.2)**

Мнемоника кода ASCII	Двоичный код	7-байтовый восьмеричный код	Шестнадцатеричный код	Примечание
Пробел	00 10 00 00	040	20	
!	00 10 00 01	041	21	
"	00 10 00 10	042	22	
#	00 10 00 11	043	23	
$	00 10 01 00	044	24	
%	00 10 01 01	045	25	
&	00 10 01 10	046	26	
'	00 10 01 11	047	27	Апостроф

Продолжение табл. А.2

Мнемоника кода ASCII	Двоичный код	7-байтовый восьмеричный код	Шестнадцатеричный код	Примечание
(	00 10 10 00	050	28	
)	00 10 10 01	051	29	
*	00 10 10 10	052	2A	
+	00 10 10 11	053	2B	
,	00 10 11 00	054	2C	
−	00 10 11 01	055	2D	Дефис
.	00 10 11 10	056	2E	
/	00 10 11 11	057	2F	
ϕ	00 11 00 00	060	30	
1	00 11 00 01	061	31	
2	00 11 00 10	062	32	
3	00 11 00 11	063	33	
4	00 11 01 00	064	34	
5	00 11 01 01	065	35	
6	00 11 01 10	066	36	
7	00 11 01 11	067	37	
8	00 11 10 00	070	38	
9	00 11 10 01	071	39	
:	00 11 10 10	072	3A	
;	00 11 10 11	073	3B	
<	00 11 11 00	074	3C	
=	00 11 11 01	075	3D	
>	00 11 11 10	076	3E	
?	00 11 11 11	077	3F	
@	01 00 00 00	100	40	
A	01 00 00 01	101	41	
B	01 00 00 10	102	42	
C	01 00 00 11	103	43	
D	01 00 01 00	104	44	
E	01 00 01 01	105	45	
F	01 00 01 10	106	46	
G	01 00 01 11	107	47	
H	01 00 10 00	110	48	
I	01 00 10 01	111	49	
J	01 00 10 10	112	4A	
K	01 00 10 11	113	4B	
L	01 00 11 00	114	4C	
M	01 00 11 01	115	4D	
N	01 00 11 10	116	4E	
O	01 00 11 11	117	4F	
P	01 01 00 00	120	50	
Q	01 01 00 01	121	51	
R	01 01 00 10	122	52	
S	01 01 00 11	123	53	
T	01 01 01 00	124	54	
U	01 01 01 01	125	55	
V	01 01 01 10	126	56	
W	01 01 01 11	127	57	

Окончание табл. А.2

Мнемоника кода ASCII	Двоичный код	7-байтовый восьмеричный код	Шестнадцатеричный код	Примечание
X	01 01 10 00	130	58	
Y	01 01 10 01	131	59	
Z	01 01 10 11	132	5A	
[	01 01 10 11	133	5B	
	01 01 11 00	134	5C	
]	01 01 11 01	135	5D	
∧или↑	01 01 11 10	136	5E	
—или←	01 01 11 11	137	5F	Подчеркивание
	01 10 00 00	140	60	Знак ударения
a	01 10 00 01	141	61	
b	01 10 00 10	142	62	
c	01 10 00 11	143	63	
d	01 10 01 00	144	64	
e	01 10 01 01	145	65	
f	01 10 01 10	146	66	
g	01 10 01 11	147	67	
h	01 10 10 00	150	68	
i	01 10 10 01	151	69	
j	01 10 10 10	152	6A	
k	01 10 10 11	153	6B	
l	01 10 11 00	154	6C	
m	01 10 11 01	155	6D	
n	01 10 11 10	156	6E	
o	01 10 11 11	157	6F	
p	01 11 00 00	160	70	
q	01 11 00 01	161	71	
r	01 11 00 10	162	72	
s	01 11 00 11	163	73	
t	01 11 01 00	164	74	
u	01 11 01 01	165	75	
v	01 11 01 10	166	76	
w	01 11 01 11	167	77	
x	01 11 10 00	170	78	
y	01 11 10 01	171	79	
z	01 11 10 10	172	7A	
{	01 11 10 11	173	7B	
|	01 11 11 00	174	7C	
}	01 11 11 01	175	7D	Перемена режима
~	01 11 11 10	176	7E	
DEL	01 11 11 11	177	7F	Стирание, удаление

ПРИЛОЖЕНИЕ Б

УКАЗАТЕЛЬ МНЕМОНИК И НАБОР ИНСТРУКЦИЙ ЭВМ PDP-11

* означает, что таких команд нет в наличии для LSI-11.

Б.1. НАБОР ИНСТРУКЦИЙ ЭВМ PDP-11. ВВЕДЕНИЕ

Спецификация каждой инструкции включает: мнемонику, восьмеричный код; диаграмму, показывающую формат инструкции; символьную нотацию, описывающую выполнение инструкции и воздействие на коды условий; описание; специальные комментарии; примеры.

Мнемоника. Приводится в верхнем углу каждой спецификации. Если для словной инструкции существует байтовый эквивалент, приводится также и байтовая мнемоника.

Формат инструкции. Сопровождающая каждую инструкцию диаграмма показывает восьмеричный код операции, двоичный код операции и распределение бит (обратите внимание, что в байтовых инструкциях самый старший бит (бит b_{15}) всегда равен 1).

Символы

( ) – содержимое чего-либо;

SS или **исх** – исходный адрес;

DD или **назн** – адрес назначения;

ячка – ячейка;

← – становится;

↑ – выталкивается из стека;

↓ – проталкивается в стек;

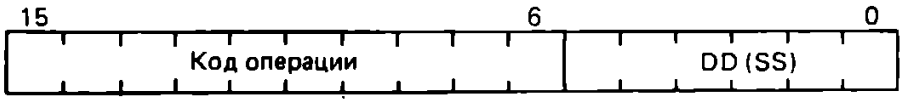
^ – логическое И;

∨ – логическое ИЛИ;
⊻ – исключающее ИЛИ;
~ – логическое НЕ;
Рег или R – регистр;
В – байт;
■ – {0 – для слова; 1 – для байта;
, – операция конкатенации.

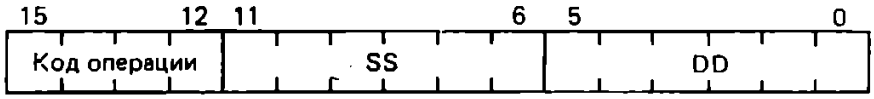
Б.2. ФОРМАТЫ ИНСТРУКЦИЙ

МикроЭВМ LSI-11 имеет следующие форматы инструкций. (Более подробную информацию см. в описании отдельных инструкций.)

1. Группа однооперандных инструкций (CLR, CLRB, COM, COMB, INC, INCB, DEC, DECB, NEG, NEGB, ADC, ADCB, SBC, SBCB, TST, TSTB, ROR, RORB, ROL, ROLB, ASR, ASRB, ASL, ASLB, JMP, SWAB, MFPS, MTPS, SXT, XOR):

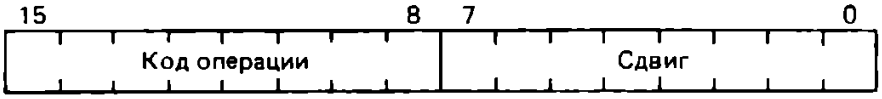


2. Группа двухоперандных инструкций (BIT, BITB, BIC, BICB, BIS, BISB, ADD, SUB, MOV, MOVB, CMP, CMPB):

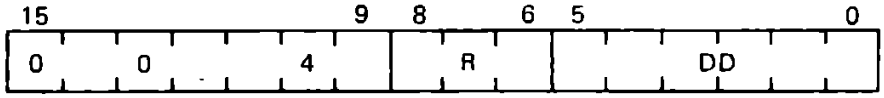


3. Группа инструкций управления ходом программы:

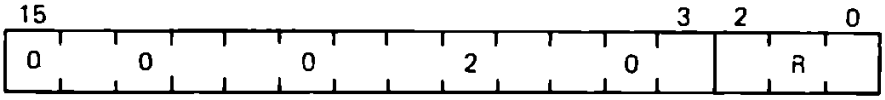
а) инструкция ветвления:



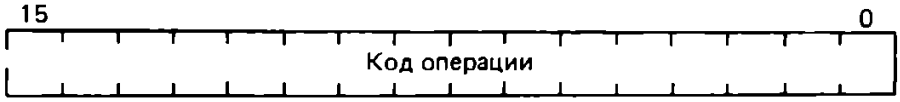
б) инструкции перехода к подпрограмме (JSR):



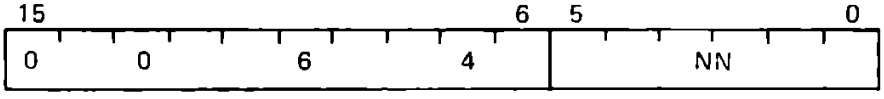
в) инструкция возврата из подпрограммы (RTS):



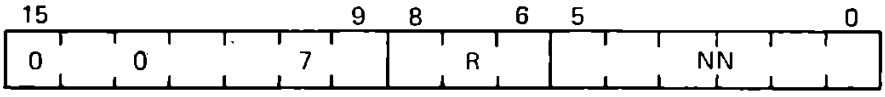
г) инструкции ловушек (BPT, IOT, EMT, TRAP):



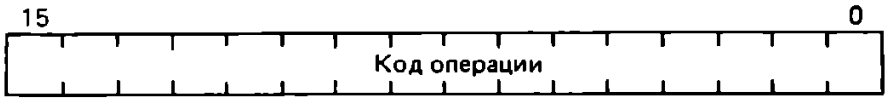
д) инструкция метки стека (MARK):



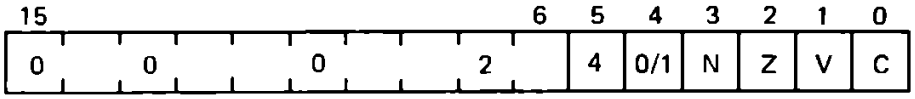
е) инструкция вычитания единицы и ветвления, если 0 (SOB):



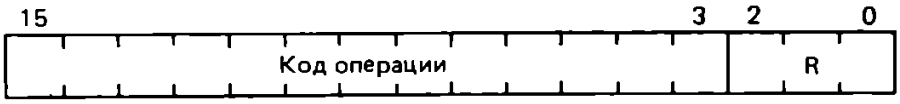
4. Инструкции операционной группы (HALT, WAIT, RTI, RESET, RTT, NOP):



5. Инструкции операций над кодами условий:

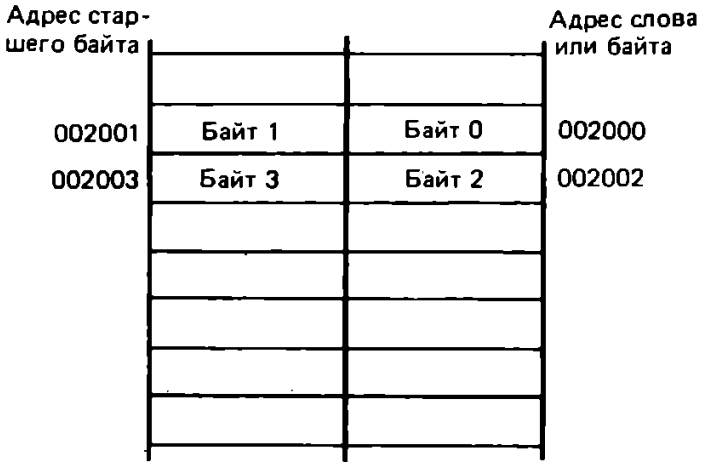


6. Инструкции арифметики с фиксированной и плавающей точкой (необязательные: расширение набора инструкций и набор инструкций с плавающей точкой) (FADD, FSUB, FMUL, FDIV, MUL, DIV, ASH, ASHC):



БАЙТОВЫЕ ИНСТРУКЦИИ

МикроЭВМ LSI-11 располагает полным набором инструкций для манипулирования байтовыми операндами. Поскольку эта ЭВМ – с байтовой организацией, адресация при манипуляциях с байтовыми операциями весьма проста. Байтовые инструкции при непосредственной адресации с автоувеличением или автоуменьшением вызывают изменение содержимого указанного в инструкции регистра на единицу, чтобы он указывал на следующий байт данных. Байтовые операции с регистровым режимом адресации осуществляют обращение к младшему байту указанного регистра. Все это делает возможной обработку как слов, так и байт. Числовая схема для адресации слов и байт в памяти:



Для индикации байтовой инструкции устанавливается в единицу самый старший бит ее первого слова.

Пример:

Символьное представление	*Восьмеричное представление*	
CLR	0050DD	Очистить слово
CLRB	1050DD	Очистить байт

Б.3. СПИСОК ИНСТРУКЦИЙ

Инструкции микроЭВМ LSI-11 далее приводятся в следующей последовательности.

ОДНООПЕРАНДНЫЕ ИНСТРУКЦИИ

Общие инструкции

Мнемоника	*Инструкция*	*Код операции*
CLR(B)	очистить **назн**	■ 050DD
COM(B)	взять дополнение **назн**	■ 051DD
INC(B)	увеличить **назн**	■ 052DD
DEC(B)	уменьшить **назн**	■ 053DD
NEG(B)	изменить знак **назн**	■ 054DD
TST(B)	проверить **назн**	■ 057DD

Сдвиг и циклический сдвиг

Мнемоника	*Инструкция*	*Код операции*
ASR(B)	арифметически сдвинуть вправо	■ 062DD
ASL(B)	арифметически сдвинуть влево	■ 063DD
ROR(B)	циклически сдвинуть вправо	■ 060DD
ROL(B)	циклически сдвинуть влево	■ 061DD
SWAB	обменять байты	■ 0003DD

Многократная точность

Мнемоника	*Инструкция*	*Код операции*
ADC(B)	прибавить бит переноса	■ 055DD
SBC(B)	вычесть бит переноса	■ 056DD
SXT	распространить знаковый бит	0067DD

Операции со словом состояния процессора

Мнемоника	*Инструкция*	*Код операции*
MFPS	перенести байт из слова состояния	1067DD
MTPS	перенести байт в слово состояния	1064SS

ДВУХОПЕРАНДНЫЕ ИНСТРУКЦИИ

Общие инструкции

Мнемоника	*Инструкция*	*Код операции*
MOV(B)	перенести источник по назначению	■ 1SSDD
CMP(B)	сравнить **ист** и **назн**	■ 2SSDD
ADD	прибавить **ист** к **назн**	06SSDD
SUB	вычесть **ист** из **назн**	16SSDD

Логические операции

Мнемоника	*Инструкция*	*Код операции*
BIT(B)	проверить биты	■ 3SSDD
BIC(B)	очистить биты	■ 4SSDD
BIS(B)	установить биты	■ 5SSDD
XOR	исключающее ИЛИ	074RDD

ИНСТРУКЦИИ УПРАВЛЕНИЯ ХОДОМ ПРОГРАММЫ

Ветвления

Мнемоника	*Инструкция*	*Код операции*
BR	ветвление (безусловное)	000400
BNE	ветвление, если не равно (нулю)	001000
BEQ	ветвление, если равно (нулю)	001400
BPL	ветвление, если плюс	100000
BMI	ветвление, если минус	100400
BVC	ветвление, если бит переполнения сброшен	102000
BVS	ветвление, если бит переполнения установлен	102400
BCC	ветвление, если бит переноса сброшен	103000
BCS	ветвление, если бит переноса установлен	103400

Условные ветвления с учетом знака

Мнемоника	*Инструкция*	*Код операции*
BGE	ветвление, если больше или равно (нулю)	002000
BLT	ветвление, если меньше (нуля)	002400
BGT	ветвление, если больше (нуля)	003000
BLE	ветвление, если меньше или равно (нулю)	003400

Условные ветвления без учета знака

Мнемоника	*Инструкция*	*Код операции*
BHI	ветвление, если выше	101000
BLOS	ветвление, если ниже или то же самое	101400
BHIS	ветвление, если выше или то же самое	103000
BLO	ветвление, если ниже	103400

Переходы и переходы к программам

Мнемоника	*Инструкция*	*Код операции*
JMP	перейти	0001DD
JSR	перейти к подпрограмме	004RDD
RTS	вернуться из подпрограммы	00020R
MARK	пометить стек	006400
SOB	вычесть единицу и перейти (если не нуль)	077R00

Ловушки и прерывания

Мнемоника	*Инструкция*	*Код операции*
EMT	ловушка имитатора	104000–104377
TRAP	ловушка	104400–104777
BPT	ловушка точки разрыва	000003
IOT	ловушка ввода-вывода	000004
RTI	возврат из прерывания	000002
RTT	возврат из прерывания	000006

СМЕШАННЫЕ ИНСТРУКЦИИ

Мнемоника	*Инструкция*	*Код операции*
HALT	остановить процессор	000000
WAIT	ждать прерывания	000001
RESET	сбросить шину в исходное состояние	000005

ЗАРЕЗЕРВИРОВАННЫЕ ИНСТРУКЦИИ

Код операции
00021R
00022

ОПЕРАЦИИ НАД КОДАМИ УСЛОВИЙ

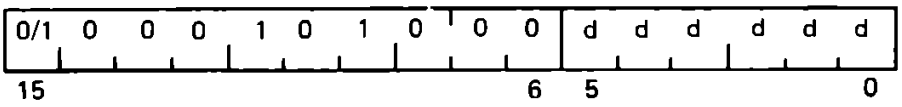
Мнемоника	*Инструкция*	*Код операции*
CLC	очистить бит C	000241
CLV	очистить бит V	000422
CLZ	очистить бит Z	000244
CLN	очистить бит N	000250
CCC	очистить все биты условий	000257
SEC	установить бит C	000261
SEV	установить бит V	000262
SEZ	установить бит Z	000264
SEN	установить бит N	000270
SCC	установить все биты условий	000277
NOP	нет операции	000240

Б.4. ОДНООПЕРАНДНЫЕ ИНСТРУКЦИИ

Общие инструкции

CLR
CLRB

Очистить операнд назначения ■050DD



Операция: (назн) ← 0
Коды условий: N: Очищается.
Z: Устанавливается.
V: Очищается.
C: Очищается.
Описание: *Для слова и для байта.* Содержимое указанного операнда назначения заменяется нулем.

Пример:

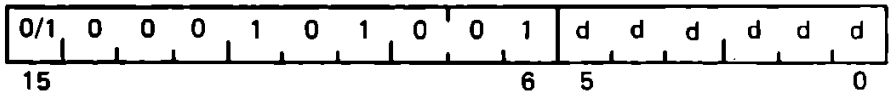
CLR R1

До выполнения	После выполнения
(R1) = 177777	(R1) = 000000
N Z V C	N Z V C
1 1 1 1	0 1 0 0

Замечание. Во время выполнения инструкций CLR и CLRB последним циклом на шине является цикл DATIO. Часть DATI этого цикла в данном случае не требует внимания, но во избежание ошибки тайм-аута на шине адресуемая память или устройство должно быть способно ответить на этот цикл DATI.

COM
COMB

Получить дополнение операнда назначения ■051DD



Операция: (назн) ← ~(назн)
Коды условий: N: Устанавливается, если установлен самый старший бит результата; иначе очищается.
Z: Устанавливается, если результат равен нулю; иначе очищается.
V: Очищается.
C: Устанавливается.
Описание: *Для слова и для байта.* Заменяет содержимое адреса назначения его логическим дополнением (каждый бит, равный 0, устанавливается, а каждый бит, равный 1, очищается).

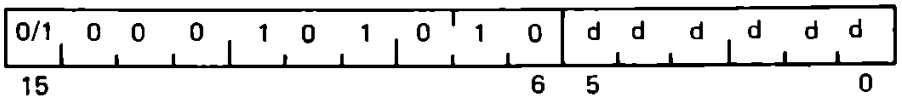
Пример:

COM R0

До выполнения	После выполнения
(R0) = 013333	(R0) = 164444
N Z V C	N Z V C
0 1 1 0	1 0 0 1

INC
INCB

Увеличить операнд назначения ■052DD



Операция: (назн) ← (назн) + 1
Коды условий: N: Устанавливается, если результат меньше нуля; иначе очищается.
Z: Устанавливается, если результат равен нулю; иначе очищается.
V: Устанавливается, если (назн) содержит 077777; иначе очищается.
C: без изменения.
Описание: *Для слова и для байта.* Прибавляет единицу к содержимому адреса назначения.

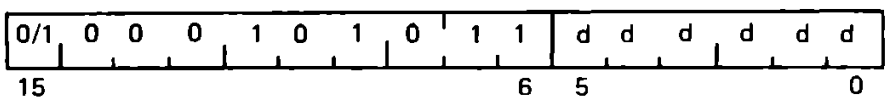
Пример:

INC R2

До выполнения	После выполнения
(R2) = 000333	(R2) = 000334
N Z V C	N Z V C
0 0 0 0	0 0 0 0

DEC
DECB

Уменьшить операнд назначения ■053DD



Операция: (назн) ← (назн) – 1

Коды условий: N: Устанавливается, если результат меньше нуля; иначе очищается.
Z: Устанавливается, если результат равен нулю; иначе очищается.
V: Устанавливается, если (назн) равно 100000; иначе очищается.
C: без изменения.

Описание: *Для слова и для байта.* Вычитает единицу из содержимого адреса назначения.

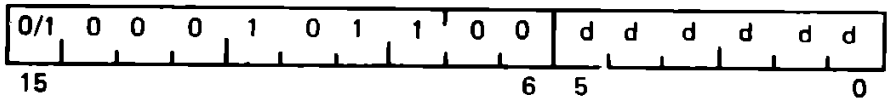
Пример:

DEC R5

До выполнения	После выполнения
(R5) = 000001	(R5) = 000000
N Z V C	N Z V C
1 0 0 0	0 1 0 0

NEG
NEGB

Изменить знак операнда назначения ■054DD



Операция: (назн) ← – (назн)

Коды условий: N: Устанавливается, если результат меньше нуля; иначе очищается.
Z: Устанавливается, если результат равен нулю; иначе очищается.
V: Устанавливается, если результат равен 100000; иначе очищается.
C: Очищается, если результат равен нулю; иначе устанавливается.

Описание: *Для слова и для байта.* Заменяет содержимое адреса назначения его дополнением до двух. Обратите внимание, что значение 100000 заменяется самим собой (в нотации с дополнением до двух для наибольшего отрицательного числа нет соответствующего положительного числа).

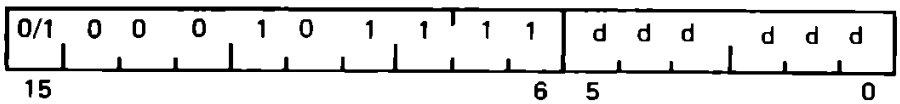
Пример:

NEG R0

До выполнения	После выполнения
(R0) = 000010	(R0) = 177770
N Z V C	N Z V C
0 0 0 0	1 0 0 1

TST
TSTB

Проверить операнд назначения ■057DD



Операция: (назн) ← (назн)

Коды условий: N: Устанавливается, если результат меньше нуля; иначе очищается.
Z: Устанавливается, если результат равен нулю; иначе очищается.
V: Очищается.
C: Очищается.

Описание: *Для слова и для байта.* Устанавливает коды условий N и Z в соответствии с содержимым адреса назначения, которое при этом остается без изменения.

Пример:

TST R1

До выполнения	После выполнения
(R1) = 012340	(R1) = 012340
N Z V C	N Z V C
0 0 1 1	0 0 0 0

Сдвиги

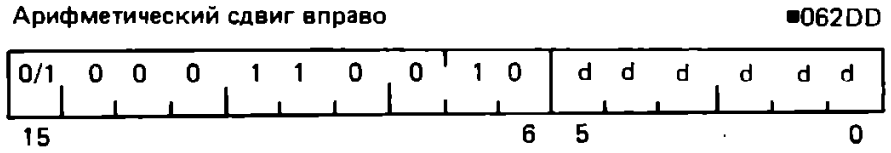
Масштабирование данных с коэффициентом 2 выполняется инструкциями сдвига:
ASR – арифметически сдвинуть вправо;
ASL – арифметически сдвинуть влево.

При сдвигах вправо знаковый бит (бит b_{15}) воспроизводится. При сдвигах в самый младший бит задвигаются нули. Биты, выдвигаемые из бита переноса С, как показано в последующих примерах, теряются.

Циклические сдвиги

Инструкции циклического сдвига оперируют со словом назначения и битом переноса С так, как если бы они представляли собой 17-битовый циклический буфер. Эти инструкции помогают осуществлять последовательные проверки бит и манипуляцию конкретными битами.

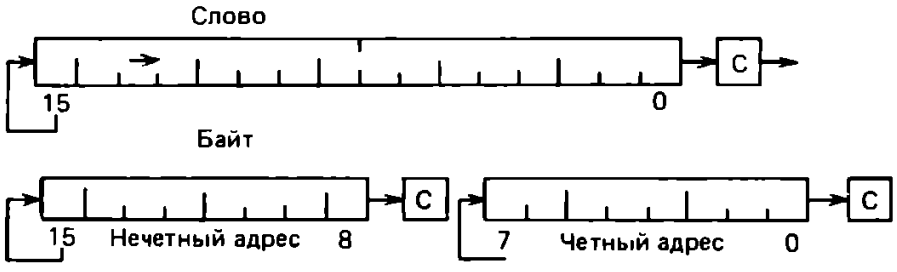
ASR
ASRB



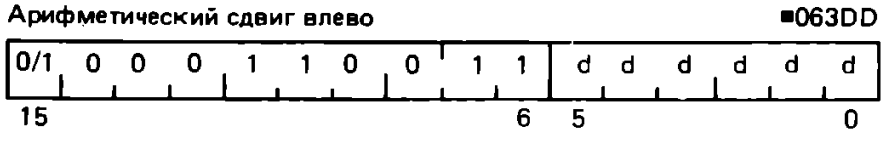
Операция: (назн) ← (назн), сдвинутое на одну позицию вправо.

Коды условий: N: Устанавливается, если установлен самый старший бит результата (результат меньше нуля); иначе очищается.
Z: Устанавливается, если результат равен нулю; иначе очищается.
V: Загружается результатом операции ИСКЛЮЧАЮЩЕЕ ИЛИ над битами N и С, когда берутся их значения после завершения операции сдвига.
С: Загружается из самого младшего бита операнда назначения.

Описание: *Для слова.* Сдвигает все биты операнда назначения на одну позицию вправо. Бит b_{15} воспроизводится. Бит С загружается из бита 0. Инструкция выполняет деление операнда назначения на два как числа со знаком.



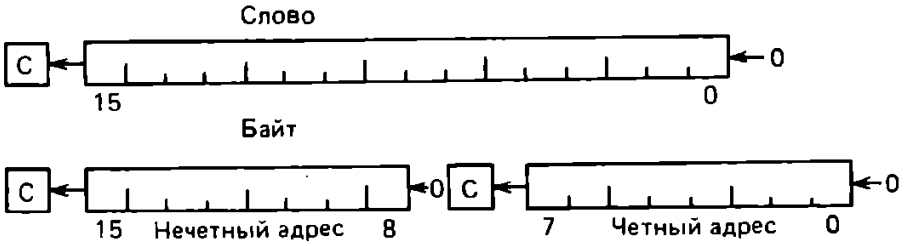
ASL
ASLB



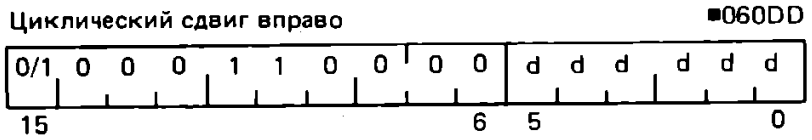
Операция: (назн) ← (назн), сдвинутое на одну позицию влево.

Коды условий: N: Устанавливается, если установлен самый старший бит результата (результат меньше нуля); иначе очищается.
Z: Устанавливается, если результат нулевой; иначе очищается.
V: Загружается результатом операции ИСКЛЮЧАЮЩЕЕ ИЛИ над битами N и C, когда берутся их значения после завершения операции сдвига.
C: Загружается значением самого старшего бита операнда назначения.

Описание: *Для слова.* Сдвигает все биты операнда назначения на одну позицию влево. Бит 0 загружается значением 0. Бит переноса C слова состояния процессора загружается значением самого старшего бита операнда назначения. Инструкция выполняет умножение на два операнда назначения как числа со знаком.



ROR
RORB

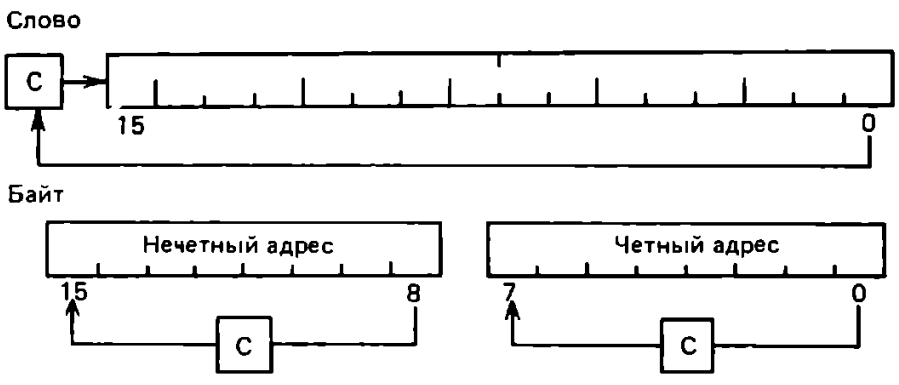


Операция: (назн) ← (назн), циклически сдвинутое вправо на одну позицию.

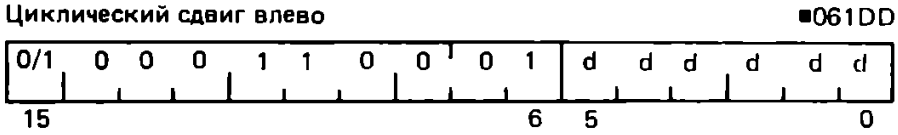
Коды условий: N: Устанавливается, если установлен самый старший бит результата (результат меньше нуля); иначе очищается.
Z: Устанавливается, если все биты результата равны 0; иначе очищается.
V: Загружается результатом операции ИСКЛЮЧАЮЩЕЕ ИЛИ над битами N и C, когда берутся их значения после завершения операции циклического сдвига.
C: Загружается самым младшим битом операнда назначения.

Описание: *Для слова.* Циклически сдвигает все биты операнда назначения на одну позицию вправо. Бит 0 загружается в бит переноса C, а предыдущее содержимое бита C загружается в бит b_{15} операнда назначения. *Для байта* аналогично.

Пример:



ROL
ROLB

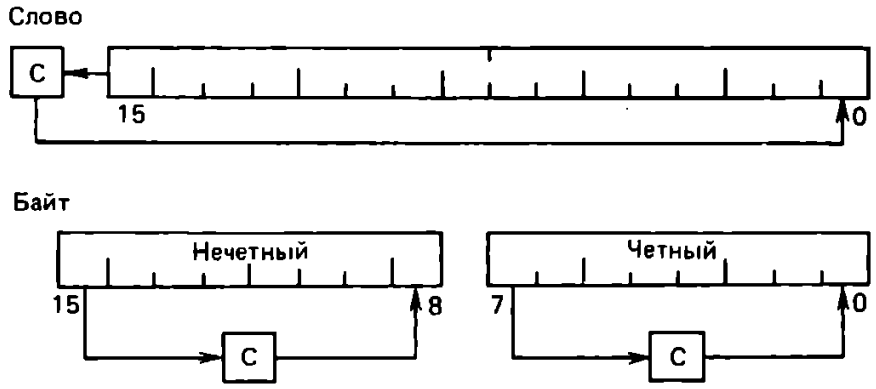


Операция: (назн) ← (назн), циклически сдвинутое на одну позицию влево

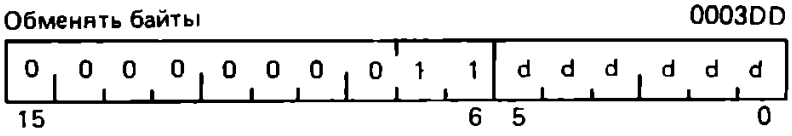
Коды условий: N: Устанавливается, если установлен самый старший бит результата (результат меньше нуля); иначе очищается.
Z: Устанавливается, если все биты результата равны нулю; иначе очищается.
V: Загружается результатом операции ИСКЛЮЧАЮЩЕЕ ИЛИ над битами N и C, когда берутся их значения после завершения операции циклического сдвига.
C: Загружается самым старшим битом операнда назначения.

Описание: *Для слова.* Циклически сдвигает все биты операнда назначения на одну позицию влево. Бит b_{15} загружается в бит переноса C слова состояния процессора, а прежнее содержимое бита C загружается в бит 0 операнда назначения. *Для байта* аналогично.

Пример:



SWAB



Операция: Байт 1 – Байт 0 ← Байт 0 – Байт 1

Коды условий: N: Устанавливается, если установлен самый старший бит младшего байта (бит b_7) результата; иначе очищается.
Z: Устанавливается, если младший байт результата равен нулю; иначе очищается.
V: Очищается.
C: Очищается.

Описание: Меняет местами старший и младший байты слова назначения (операнд назначения должен иметь словный адрес).

Пример:

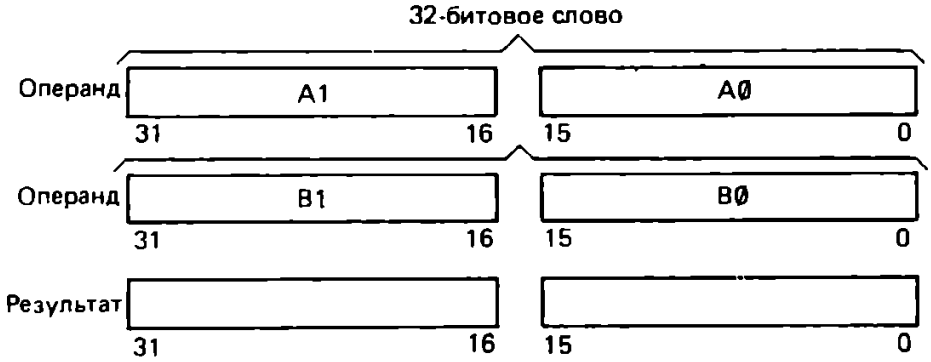
SWAB R1

До выполнения	После выполнения
(R1) = 077777	(R1) = 177577
N Z V C	N Z V C
1 1 1 1	0 0 0 0

Многократная точность

Иногда бывает необходимо выполнять арифметические операции над операндами, состоящими из нескольких слов или байт. В микроЭВМ LSI-11 для обеспечения таких операций имеются две инструкции: ADC (прибавить бит переноса) и SBC (вычесть бит переноса) и их байтовые эквиваленты.

Например, два 16-битовых слова можно объединить в одно 32-битовое слово двойной точности и выполнять сложение и вычитание следующим образом:



Пример:

Сложение чисел –1 и –1 может быть выполнено так:

–1 = 37777777777

(R1) = 177777 (R2) = 177777 (R3) = 177777 (R4) = 177777

ADD R1, R2

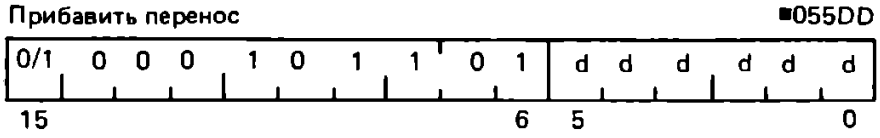
ADC R3

ADD R4, R3

1. После сложения (R1) и (R2) в бит переноса C загружается 1.
2. Инструкция ADC прибавляет бит C к (R3); (R3) = 0.
3. Складываются (R3) и (R4).
4. Результат равен 37777777776 или –2.

ADC

ADCB



Операция: (назн) ← (назн) + (бит C)

Коды условий: N: Устанавливается, если результат меньше нуля; иначе очищается.
Z: Устанавливается, если результат равен нулю; иначе очищается.
V: Устанавливается, если (назн) было 077777 и (C) = 1; иначе очищается.
C: Устанавливается, если (назн) было 177777 и (C) = 1; иначе очищается.

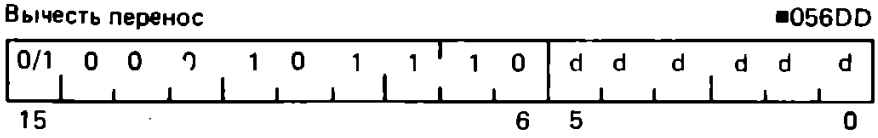
Описание: *Для слова и для байта.* Прибавляет содержимое бита переноса C к операнду назначения. Это позволяет учитывать перенос, получаемый при сложении младших слов, при сложении старших слов.

Пример.

Сложение двойной точности может быть выполнено с помощью следующей последовательности инструкций:

```
ADD     A0,B0   ;СЛОЖИТЬ МЛАДШИЕ ЧАСТИ
ADC     B1      ;ПРИБАВИТЬ ПЕРЕНОС К СТАРШЕЙ ЧАСТИ
ADD     A1,B1   ;СЛОЖИТЬ СТАРШИЕ ЧАСТИ
```

SBC
SBCB



Операция: (назн) ← (назн) – (C)

Коды условий: N: Устанавливается, если результат меньше нуля; иначе очищается.
Z: Устанавливается, если результат равен нулю; иначе очищается.
V: Устанавливается, если (назн) равно 100000; иначе очищается.
C: Устанавливается, если (назн) равно 0 и (C) равно 1; иначе очищается.

Описание: *Для слова.* Вычитает содержимое бита переноса C из операнда назначения. Это позволяет учитывать перенос, полученный при вычитании младших слов, при вычитании старших слов. *Для байта* аналогично.

Пример.

Вычитание двойной точности выполняется с помощью следующей последовательности инструкций:

```
SUB     A0,B0
SBC     B1
SUB     A1,B1
```

SXT

Операция: (назн) ← 0, если бит N очищен;
(назн) ← –1, если бит N установлен.

Коды условий: N: Без изменения.
Z: Устанавливается, если бит N очищен.
V: Очищается.
C: Без изменения.

Описание: Если установлен бит условия N, то в операнд назначения помещается –1; если бит N очищен, то в операнд назначения помещается 0. Эта инструкция полезна, в частности, для арифметики многократной точности, так как позволяет распространять знак числа на несколько слов.

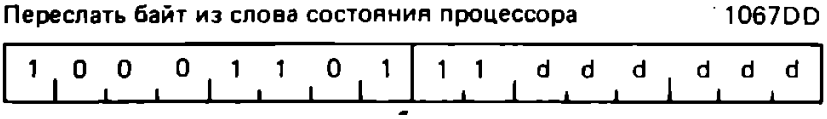
Пример:

SXT A

До выполнения	После выполнения
(A) = 012345	(A) = 177777
N Z V C	N Z V C
1 0 0 0	1 0 0 0

Б.5. ОПЕРАЦИИ СО СЛОВОМ СОСТОЯНИЯ ПРОЦЕССОРА

MFPS



Операция: (назн) ← PSW; операндом назначения являются 8 младших бит
Коды условий: N: Устанавливается, если бит 7 PSW равен 1; иначе очищается.
Z: Устанавливается, если в PSW 0 : 7 = 0; иначе очищается.
V: Очищается.
C: Без изменения.
Описание: Из содержимого слова состояния процессора PSW 8 младших бит извлекаются и пересылаются в операнд назначения. Если для операнда назначения используется режим 0, то бит 7 слова состояния распространяется в качестве знакового бита на старший байт регистра. Адрес операнда назначения рассматривается как байтовый.

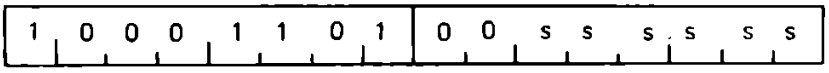
Пример:

```
MFPS R0

До выполнения          После выполнения
R0   [0]               R0   [000014]
PSW  [000014]          PSW  [000000]
```

MTPS

Переслать байт в слово состояния процессора 1064SS



Операция: PSW ← (исх)
Коды условий: Устанавливаются в соответствии с битами 0–3 исходного операнда.
Описание: Младшие 8 бит эффективного операнда заменяют текущее содержимое PSW. Адрес исходного операнда рассматривается как байтовый. Обратите внимание, что бит трассировки T (бит 4 PSW) не может быть установлен этой инструкцией. Исходный операнд остается без изменения. Эта инструкция может использоваться для изменения бита приоритета (бита 7) в PSW.

Замечание. При выполнении инструкции MTPS процессор микроЭВМ LSI-11 извлекает исходный операнд с помощью цикла шины DATIO, а не цикла DATI. Если исходный операнд содержится в только читаемой памяти, то произойдет ошибка шины (тайм-аут), так как процессор после извлечения операнда будет пытаться произвести запись в адресованную ячейку. Если вам нужно воспользоваться инструкцией MTPS в программах, которые будут записаны в только читаемую память, обратитесь к фирменному справочнику по устройствам памяти и периферийным устройствам (гл 3).

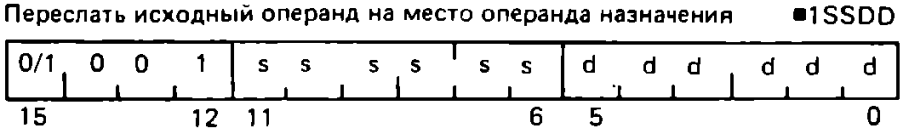
Б.6. ДВУХОПЕРАНДНЫЕ ИНСТРУКЦИИ

Двухоперандные инструкции позволяют уменьшить число инструкций (и время выполнения) по сравнению с аккумуляторно-ориентированными ЭВМ, так как они избавляют от загрузки и сохранения аккумулятора.

ОБЩИЕ ИНСТРУКЦИИ

MOV
MOVB

Переслать исходный операнд на место операнда назначения ●1SSDD



Операция: (назн) ← (исх)

Коды условий: N: Устанавливается, если (исх) < 0; иначе очищается.
Z: Устанавливается, если (исх) = 0; иначе очищается.
V: Очищается.
C: Без изменения.

Описание: *Для слова.* Переносит исходный операнд в ячейку назначения. Предыдущее содержимое ячейки назначения теряется. Содержимое исходного операнда остается без изменения. *Для байта* аналогично. Инструкция переноса байта в регистр (уникальная среди байтовых инструкций) распространяет самый старший бит младшего байта (осуществляет распространение знака). В других случаях инструкция MOVB оперирует байтами точно так же, как инструкция MOV словами.

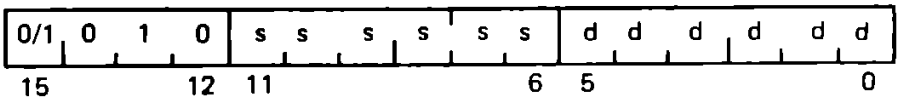
Примеры:

```
MOV     XXX,R1          ;ЗАГРУЗИТЬ РЕГИСТР 1 СОДЕРЖИМЫМ ЯЧЕЙКИ ПАМЯТИ
                        ;XXX - ЭТО ОПРЕДЕЛЕННАЯ ПОЛЬЗОВАТЕЛЕМ МНЕМОНИКА
                        ;ДЛЯ ПРЕДСТАВЛЕНИЯ ЯЧЕЙКИ ПАМЯТИ
MOV     #20,R0          ;ЗАГРУЗИТЬ ЧИСЛО 20 В РЕГИСТР 0
MOV     @#20,-(R6)      ;ПРОТОЛКНУТЬ В СТЕК ОПЕРАНД,
                        ;СОДЕРЖАЩИЙСЯ В ЯЧЕЙКЕ 20
MOV     (R6)+,@#177566  ;ВЫТОЛКНУТЬ ОПЕРАНД ИЗ СТЕКА И ПЕРЕНЕСТИ
                        ;ЕГО В ЯЧЕЙКУ 177566 (БУФЕРНЫЙ РЕГИСТР
                        ;ПЕЧАТАЮЩЕГО УСТРОЙСТВА ТЕРМИНАЛА)
MOV     R1,R3           ;ВЫПОЛНЯЕТ МЕЖРЕГИСТРОВУЮ ПЕРЕСЫЛКУ
MOV     @#177562,@#177566  ;ПЕРЕНОСИТ КОД СИМВОЛА ИЗ БУФЕРА
                        ;КЛАВИАТУРЫ В БУФЕР ПЕЧАТАЮЩЕГО
                        ;УСТРОЙСТВА ТЕРМИНАЛА
```

Замечание. Инструкция MOVB последним циклом на шине выполняет цикл DATIOB (хотя достаточно было бы цикла DATOB). Часть DATI цикла шины DATIOB относится к тому типу условий, которые "не имеют значения", однако адресуемая память или устройство должно быть способно ответить на цикл DATI во избежание ошибки тайм-аута на шине. Инструкция MOV выполняет последним циклом только цикл DATO.

CMP
CMPB

Сравнить исходный операнд с операндом назначения ■2SSDD



Операция: (исх) – (назн)

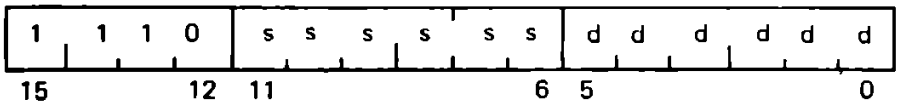
Коды условий: N: Устанавливается, если результат меньше 0; иначе очищается.
Z: Устанавливается, если результат равен 0; иначе очищается.
V: Устанавливается, если было арифметическое переполнение, т. е. операнды имели противоположные знаки, а знак операнда назначения совпадал со знаком результата; иначе очищается.
C: Очищается, если был перенос из самого старшего бита результата; иначе устанавливается.

Описание: Сравнивает исходный операнд и операнд назначения и устанавливает коды условий, которые далее могут использоваться для арифметических и логических условных переходов. Оба операнда остаются без изменения. Единственным действием является установка кодов условий. За сравнением, как правило, следует инструкция условного ветвления. Обратите внимание, что в противоположность инструкции

вычитания операция выполняется в порядке (исх) – (назн), а не (назн) – (исх).

ADD

Прибавить исходный операнд к операнду назначения 06SSDD



Операция: (назн) ← (исх) + (назн)

Коды условий: N: Устанавливается, если результат меньше 0; иначе очищается.
Z: Устанавливается, если результат равен 0; иначе очищается.
V: Устанавливается, если в результате операции было арифметическое переполнение, т. е. оба операнда были одного знака, а результат – противоположного; иначе очищается.
C: Устанавливается, если был перенос из самого старшего бита результата; иначе очищается.

Описание: Прибавляет исходный операнд к операнду назначения и помещает результат по адресу назначения. Первоначальное содержимое адреса назначения теряется. Содержимое исходного операнда остается без изменения. Выполняется сложение дополнений до двух. *Замечание.* Эквивалентная байтовая форма отсутствует.

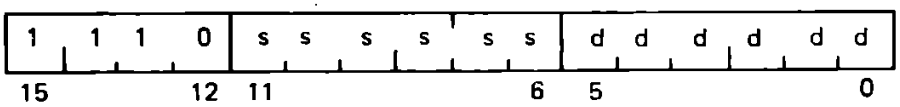
Примеры:

Прибавить к регистру:	ADD 20, R0
Прибавить к памяти:	ADD R1, XXX
Прибавить регистр к регистру:	ADD R1, R2
Прибавить память к памяти:	ADD @#177560, XXX

где XXX – это определенная пользователем мнемоника для ячейки памяти.

SUB

Вычесть исходный операнд из операнда назначения 16SSDD



Операция: (назн) ← (назн) – (исх)

Коды условий: N: Устанавливается, если результат меньше 0; иначе очищается.
Z: Устанавливается, если результат равен нулю; иначе очищается.
V: Устанавливается, если было арифметическое переполнение в результате операции, т. е. операнды имели противоположные знаки, а знак исходного операнда совпадал со знаком результата; иначе очищается.
C: Очищается, если был перенос из самого старшего бита результата; иначе устанавливается.

Описание: Вычитает исходный операнд из операнда назначения и оставляет результат по адресу назначения. Первоначальное содержимое операнда назначения теряется. Содержимое исходного операнда не меняется. Для арифметики двойной точности установленное состояние бита C означает заем.

Пример:

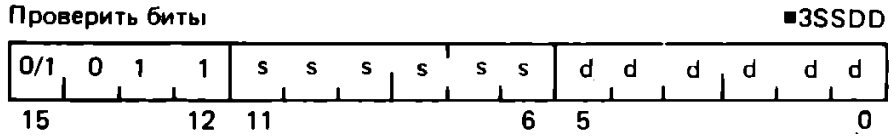
SUB R1, R2

До выполнения	После выполнения
(R1) = 011111	(R1) = 011111
(R2) = 012345	(R2) = 001234
N Z V C	N Z V C
1 1 1 1	0 0 0 0

Логические операции

Эти инструкции имеют тот же формат, что и двухоперандные инструкции арифметической группы. Они выполняют операции над данными побитно.

BIT
BITB



Операция: (исх) ∧ (назн)

Коды условий: N: Устанавливается, если самый старший бит результата установлен; иначе очищается.
Z: Устанавливается, если результат равен нулю; иначе очищается.
V: Очищается.
C: Не меняется.

Описание: Выполняет сравнение исходного операнда и операнда назначения с помощью логической операции И и соответственно модифицирует коды условий. Ни один из операндов не меняется. Эта инструкция может использоваться для проверки того, установлены ли в операнде назначения те биты, которые установлены в исходном операнде, или того, что все соответствующие биты, установленные в операнде назначения, сброшены в исходном операнде.

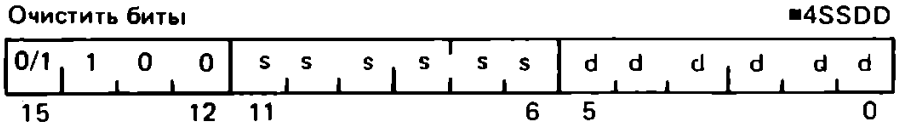
Пример:

```
BIT     #30,R3  ;ПРОВЕРИТЬ, СБРОШЕНЫ ЛИ В РЕГИСТРЕ R3 БИТЫ 3 И 4
```

R3 = 0 000 000 000 011 000

До выполнения	После выполнения
N Z V C	N Z V C
1 1 1 1	0 0 0 1

BIC
BICB



Операция: (назн) ← ~ (исх) ∧ (назн)

Коды условий: N: Устанавливается, если установлен самый старший бит результата; иначе очищается.
Z: Устанавливается, если результат равен нулю; иначе очищается.
V: Очищается.
C: Не изменяется.

Описание: Очищает все биты в операнде назначения, которые соответствуют установленным битам в исходном операнде. Первоначальное содержимое операнда назначения теряется. Содержимое исходного операнда не изменяется.

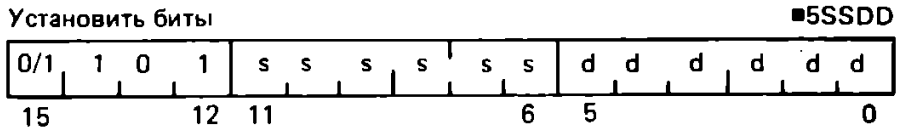
Пример:

```
BIC  R3, R4
```

До выполнения	После выполнения
(R3) = 001234	(R3) = 001234
(R4) = 001111	(R4) = 000101

N Z V C	N Z V C
1 1 1 1	0 0 0 1
До выполнения	После выполнения
(R3) = 0 000 001 010 011 100	(R4) = 0 000 000 001 000 001
(R4) = 0 000 001 001 001 001	

BIS
BISB



Операция: (назн) ← (исх) ∨ (назн)

Коды условий: N: Устанавливается, если установлен самый старший бит результата; иначе очищается.
Z: Устанавливается, если результат равен нулю; иначе очищается.
V: Очищается.
C: Не изменяется.

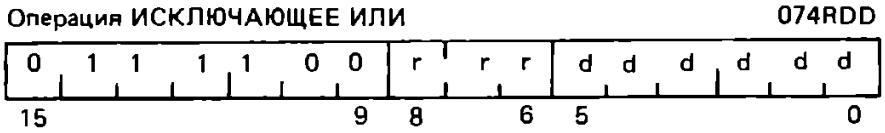
Описание: Выполняет операцию ИСКЛЮЧАЮЩЕЕ ИЛИ над исходным операндом и операндом назначения и оставляет результат по адресу назначения, т. е. соответствующие биты, установленные в исходном операнде, устанавливаются и в операнде назначения. Первоначальное содержимое операнда назначения теряется.

Пример:

BIC R0, R1

До выполнения	После выполнения
(R0) = 001234	(R0) = 001234
(R1) = 001111	(R1) = 001335
N Z V C	N Z V C
0 0 0 0	0 0 0 0
До выполнения	После выполнения
(R0) = 0 000 001 010 011 100	(R1) = 0 000 001 011 011 101
(R1) = 0 000 001 001 001 001	

XOR



Операция: (назн) ← R⊻ (назн)

Коды условий: N: Устанавливается, если результат меньше 0; иначе очищается.
Z: Устанавливается, если результат равен 0; иначе очищается.
V: Очищается.
C: Не изменяется.

Описание: Результат операции ИСКЛЮЧАЮЩЕЕ ИЛИ над содержимым регистра и операнда назначения записывается по адресу назначения. Содержимое регистра не изменяется. Формат ассемблера: XOR R, D

Пример:

XOR R0, R2

До выполнения	После выполнения
(R0) = 001234	(R0) = 001234
(R2) = 001111	(R2) = 000325
N Z V C	N Z V C
1 1 1 1	0 0 0 1

До выполнения	После выполнения
(R0) = 0 000 001 010 011 100	(R2) = 0 000 000 011 010 101
(R2) = 0 000 001 001 001 001	

Б.7. ИНСТРУКЦИИ УПРАЛЕНИЯ ХОДОМ ПРОГРАММЫ

ИНСТРУКЦИИ ВЕТВЛЕНИЯ

Эти инструкции вызывают ветвление к ячейке, определяемой суммой смещения (умноженного на 2) и текущего содержимого программного счетчика, если:

а) инструкция задает безусловное ветвление;

б) ветвление условное и после проверки кодов условий (NZVC) они удовлетворяются.

Смещение – это число слов от текущего содержимого регистра PC вперед или назад. Следует иметь в виду, что текущее содержимое регистра PC указывает на слово, следующее за инструкцией ветвления.

Смещение выражает байтовый адрес, но регистр PC выражает пословный адрес. Поэтому перед операцией сложения с содержимым регистра PC смещение автоматически умножается на два и производится распространение знакового бита. Бит 7 – это знак смещения. Если он установлен, смещение отрицательное, т. е. ветвление выполняется в обратном направлении. Аналогично если он не установлен, то смещение положительное, а ветвление осуществляется в прямом направлении.

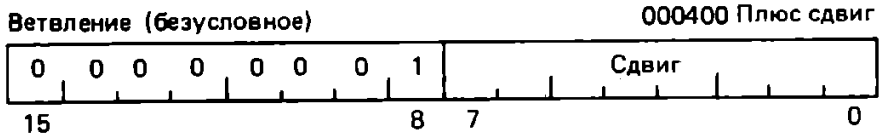
Восьмибитовое смещение дает возможность выполнить ветвления в обратном направлении до 200_8 слов (400_8 байт) от текущего содержимого регистра PC и в прямом направлении до 177_8 слов (376_8 байт) от текущего содержимого регистра PC.

Ассемблер ЭВМ PDP-11 выполняет для пользователя адресную арифметику, вычисляет и заполняет соответствующие поля смещения для инструкций ветвления в формате:

Bxx ячка

где Bxx – это инструкция ветвления, а ячка – адрес, к которому должно осуществляться ветвление. Если заданное ветвление выходит за допустимый диапазон, то ассемблер помечает такую инструкцию индикатором ошибки. Инструкции ветвления на коды условий никакого воздействия не оказывают. Любая инструкция условного ветвления, для которой условие ветвления не удовлетворяется, трактуется как инструкция отсутствия операции NOP.

BR



Операция: PC ← PC + (2 X сдвиг)

Коды условий: Без изменения.

Описание: Обеспечивает возможность передачи управления в диапазоне -128_{10} до $+127_{10}$ с помощью однословной инструкции. Новый адрес в регистре PC равен $(PC)_{скор.}$ + (2 X сдвиг), где $(PC)_{скор.}$ равно адресу инструкции ветвления плюс 2.

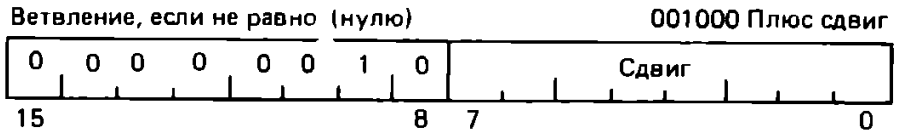
Пример.

При расположении инструкции ветвления в ячейке 500 получаются следующие сдвиги:

Новый адрес в PC	Код сдвига	Сдвиг (десятичный)
474	375	–3
476	376	–2
500	377	–1
502	000	0

	504	001	+1
	506	002	+2

BNE



Операция: PC ← PC + (2 X сдвиг), если Z = 0.
Коды условий: Без изменения.
Описание: Проверяет состояние бита Z и вызывает ветвление, если бит очищен. Эта инструкция осуществляет операцию, дополнительную к операции инструкции BEQ. Она применяется для проверки неравенства после инструкции CMP, а также для проверки того, что в операнде назначения установлены те же самые биты, которые установлены в исходном операнде, когда эта инструкция следует за инструкцией BIT, и вообще для проверки того, что результат предыдущей операции не был равен нулю.

Пример:

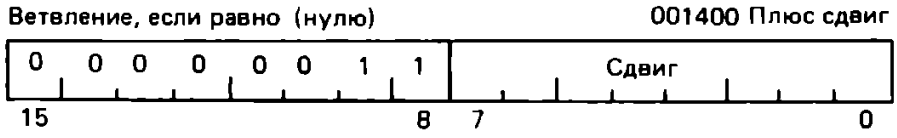
```
CMP     A,B     ;СРАВНИТЬ А И В
BNE     C       ;ВЕТВЛЕНИЕ, ЕСЛИ ОНИ НЕ РАВНЫ
```

Эта последовательность инструкций осуществит ветвление к C, если A ≠ B, а последовательность инструкций

```
ADD     A,B     ;ПРИБАВИТЬ А К В
BNE     C       ;ВЕТВЛЕНИЕ, ЕСЛИ РЕЗУЛЬТАТ НЕ РАВЕН 0
```

вызовет ветвление к C, если A + B ≠ 0.

BEQ



Операция: PC ← PC + (2 X сдвиг), если Z = 1.
Коды условий: Без изменения.
Описание: Проверяет состояние бита Z и вызывает ветвление, если бит установлен. Применяется для проверки равенства после инструкции CMP, а также для проверки того, что в операнде назначения не установлен ни один бит, установленный в исходном операнде, когда эта инструкция располагается вслед за инструкцией BIT, и вообще для проверки того, что результат предыдущей операции равен нулю.

Пример.

Последовательность инструкций

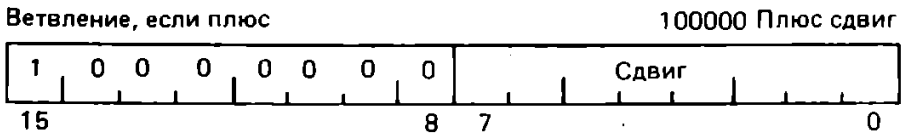
```
CMP     A,B     ;СРАВНИТЬ А И В
BEQ     C       ;ВЕТВЛЕНИЕ, ЕСЛИ ОНИ РАВНЫ
```

вызовет ветвление к C, если A = B, а последовательность инструкций

```
ADD     A,B     ;ПРИБАВИТЬ А К В
BEQ     C       ;ВЕТВЛЕНИЕ, ЕСЛИ РЕЗУЛЬТАТ РАВЕН 0
```

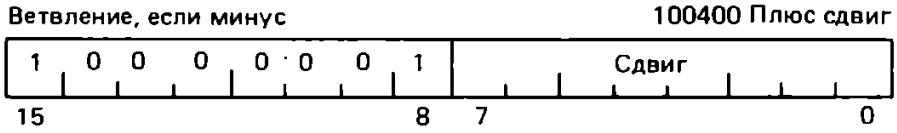
осуществит ветвление к C, если A + B = 0.

BPL



Операция: PC ← PC + (2 X сдвиг), если N = 0.
Коды условий: Без изменения.
Описание: Проверяет состояние бита N и вызывает ветвление, если бит очищен (т. е. результат положительный). Это дополнительная операция к инструкции BMI.

BMI



Операция: PC ← PC + (2 X сдвиг), если N = 1.
Коды условий: Без изменения.
Описание: Проверяет состояние бита N и вызывает ветвление, если бит установлен. Применяется для проверки знака (самого старшего бита) результата предыдущей операции, осуществляя ветвление, если он был отрицательным. Это дополнительная операция к инструкции BPL.

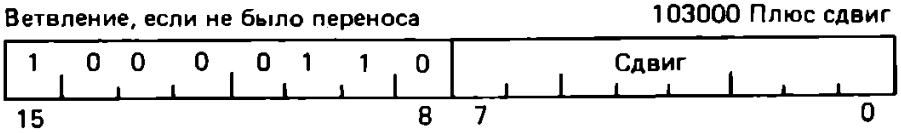
BVC

Операция: PC ← PC + (2 X сдвиг), если V = 0.
Коды условий: Без изменения.
Описание: Проверяет состояние бита V и вызывает ветвление, если бит очищен. Это дополнительная операция к инструкции BVS.

BVS

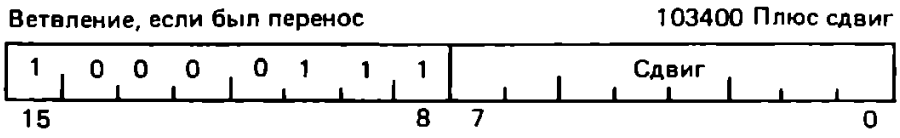
Операция: PC ← PC + (2 X сдвиг), если V = 1.
Коды условий: Без изменения.
Описание: Проверяет состояние бита переполнения V и вызывает ветвление, если он установлен. Применяется для обнаружения арифметического переполнения в предыдущей операции.

BCC



Операция: PC ← PC + (2 X сдвиг), если C = 0.
Коды условий: Без изменения.
Описание: Проверяет состояние бита C и вызывает ветвление, если бит очищен. Это дополнительная операция к инструкции BCS.

BCS



Операция: PC ← PC + (2 X сдвиг), если C = 1.
Коды условий: Без изменения.
Описание: Проверяет состояние бита C и вызывает ветвление, если бит установлен. Применяется для проверки бита переноса по результату предыдущей операции.

УСЛОВНЫЕ ВЕТВЛЕНИЯ С УЧЕТОМ ЗНАКА

Определенные комбинации бит кодов условий проверяются инструкциями ветвления с учетом знака. Эти инструкции применяются для проверки результатов выполнения таких инструкций, в которых операнды рассматриваются как значения со знаком (в виде дополнения до двух).

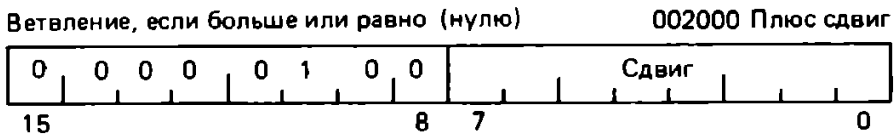
Обратите внимание, что по смыслу сравнения с учетом знака и без учета знака различаются. При 16-битовой арифметике дополнений до двух с учетом знака последовательность значений упорядочена следующим образом:

Наибольшее значение	077777
	077776
Положительные значения	.
	.
	000001
	000000
	177777
	177776
Отрицательные значения	.
	.
	.
	100001
Наименьшее значение	100000

При 16-битовой арифметике без учета знака последовательность значений упорядочена следующим образом:

Наибольшее значение	177777
	.
	.
	.
	.
	000002
	000001
Наименьшее значение	000000

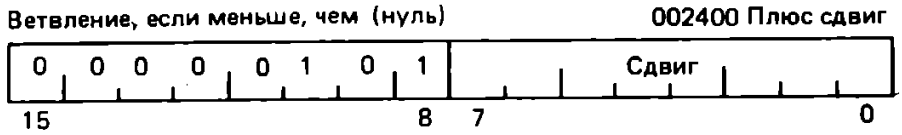
BGE



Операция: PC ← PC + (2 X сдвига), если N ⊻ V = 0.
Коды условий: Без изменения.
Описание: Вызывает ветвление, если биты N и V или оба очищены, или оба установлены. Это дополнительная операция к инструкции BLT. Эта инст-

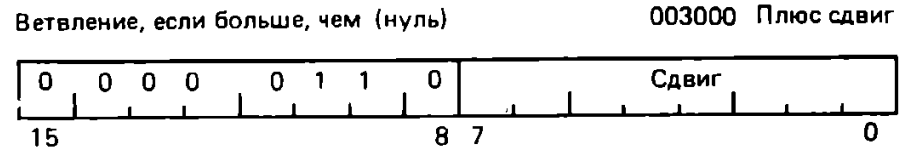
рукция всегда вызывает ветвление, если она следует за операцией сложения двух положительных чисел. Она будет вызывать ветвление также при нулевом результате.

BLT



Операция: PC ← PC + (2 × сдвиг), если N ⊻ V = 1.
Коды условий: Без изменения.
Описание: Вызывает ветвление, если равен 1 результат выполнения операции ИСКЛЮЧАЮЩЕЕ ИЛИ над битами N и V. Таким образом, эта инструкция всегда будет вызывать ветвление, если она следует за операцией, при которой складываются два отрицательных числа (даже если происходит переполнение). В частности, она всегда будет вызывать ветвление, если следует за инструкцией CMP с отрицательным исходным операндом и положительным операндом назначения (даже если происходит переполнение). Эта инструкция никогда не вызывает ветвления, если она следует за инструкцией сравнения CMP с положительным исходным операндом и отрицательным операндом назначения. Она никогда не вызывает ветвления, если результат предыдущей операции равен нулю (без переполнения).

BGT



Операция: PC ← PC + (2 × сдвиг), если Z ∨ (N ⊻ V) = 0.
Коды условий: Без изменения.
Описание: Операция, вызванная этой инструкцией, аналогична инструкции BGE, за исключением того, что в данном случае не происходит ветвления по результату, равному нулю.

BLE

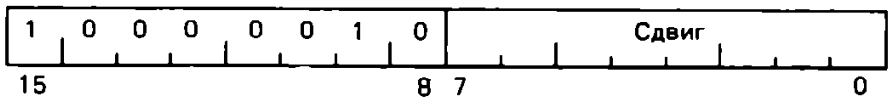
Операция: PC ← PC + (2 × сдвиг), если Z ∨ (N ⊻ V) = 1.
Коды условий: Без изменения.
Описание: Операция аналогична инструкции BLT, за исключением того, что дополнительно будет еще выполняться ветвление, если результат предыдущей операции был нулевым.

УСЛОВНЫЕ ВЕТВЛЕНИЯ БЕЗ УЧЕТА ЗНАКА

Инструкции условного ветвления без учета знака обеспечивают возможность проверки результата таких операций сравнения, в которых операнды рассматриваются как значения без знака.

BHI

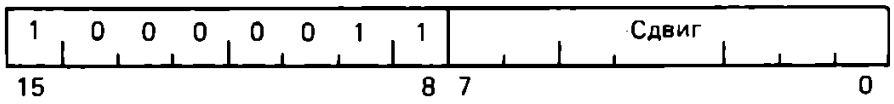
Ветвление, если выше 101000 Плюс сдвиг



Операция: PC → PC + (2 X сдвиг), если C = 0 и Z = 0.
Коды условий: Без изменения.
Описание: Вызывает ветвление, если в результате предыдущей операции не было ни переноса, ни нулевого значения. Такая ситуация возникает в операциях сравнения CMP, когда исходный операнд имеет большее значение без учета знака, чем операнд назначения.

BLOS

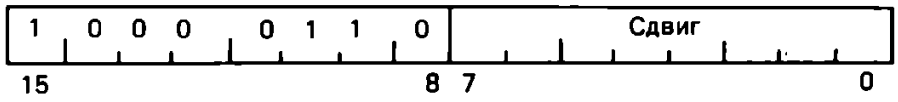
Ветвление, если ниже или то же самое 101400 Плюс сдвиг



Операция: PC ← PC + (2 X сдвиг), если C ∨ Z = 1.
Коды условий: Без изменения.
Описание: Вызывает ветвление, если предыдущая операция привела или к переносу, или к нулевому результату. Это дополнительная операция к инструкции BHI. В операциях сравнения ветвление будет происходить тогда, когда исходный операнд равен или имеет меньшее значение без учета знака, чем операнд назначения.

BHIS

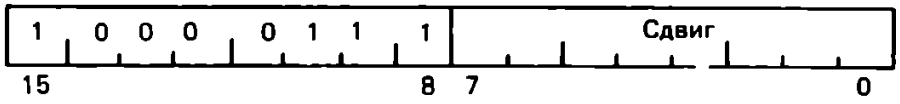
Ветвление, если выше или то же самое 103000 Плюс сдвиг



Операция: PC ← PC + (2 X сдвиг), если C = 0.
Коды условий: Без изменения.
Описание: Эта инструкция аналогична инструкции BCC. Мнемоника BHIS включена только для удобства программиста.

BLO

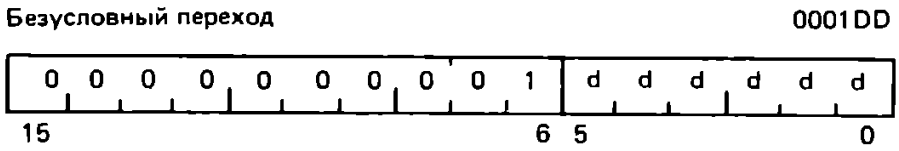
Ветвление, если ниже 103400 Плюс сдвиг



Операция: PC ← PC + (2 X сдвиг), если C = 1.
Коды условий: Без изменения.
Описание: Эта инструкция аналогична инструкции BCS. Мнемоника инструкции BLO включена только для удобства программиста.

ИНСТРУКЦИИ ПЕРЕХОДА

JMP



Операция: PC ← (назн)
Коды условий: Без изменения.
Описание: Эта инструкция обеспечивает большие возможности организации программных ветвлений, чем инструкции ветвления. Управление может быть передано любой ячейке памяти (без ограничения диапазона), причем с использованием всех возможностей режимов адресации, исключая только регистровый режим 0. Поскольку управление в программе не может быть передано регистру, выполнение перехода в режиме 0 равносильно условию недопустимой инструкции, в результате которого ЦП переходит к ловушке с адресом вектора 4. Косвенный регистровый режим является допустимым, при нем управление в программе передается тому адресу, который содержится в указанном регистре. Заметим, что инструкции располагаются в словах, и, следовательно, должны извлекаться из четных адресов.

Инструкции JMP с косвенным индексным режимом адресации позволяют передавать управление адресам, содержащимся в выборочных элементах таблицы диспетчеризации.

Пример:

```
        JMP     FIRST   ;ПЕРЕХОД К МЕТКЕ FIRST
        ...
        ...
FIRST:  ...
        JMP     @LIST   ;ПЕРЕХОД К ЯЧЕЙКЕ, НА КОТОРУЮ
        ...             ;УКАЗЫВАЕТ ЯЧЕЙКА LIST
LIST:   FIRST           ;УКАЗАТЕЛЬ НА ЯЧЕЙКУ FIRST
        JMP     @(SP)+  ;ПЕРЕХОД К ЯЧЕЙКЕ, НА КОТОРУЮ УКАЗЫВАЕТ
                        ;ВЕРХНИЙ ЭЛЕМЕНТ СТЕКА, И ВЫТАЛКИВАНИЕ
                        ;ИЗ СТЕКА ЭТОГО УКАЗАТЕЛЯ
```

Инструкции для подпрограмм

Метод вызова подпрограмм в ЭВМ PDP-11 обеспечивает автоматическую вложенность подпрограмм, повторную входимость и несколько точек входа в подпрограмму. Одни подпрограммы могут вызывать другие подпрограммы (или даже сами себя) на любом уровне вложенности без необходимости принимать специальные меры по сохранению адресов возврата на каждом уровне подпрограммных вызовов. Поскольку при этом методе вызова подпрограмм не модифицируются никакие фиксированные ячейки памяти, обеспечивается возможность повторной входимости, что позволяет разделять одну копию подпрограммы между несколькими прерывающими процессами.

JSR

Операция: (врем) ← (назн) (**врем** – это внутренний регистр ЦП).
↓(SP) ← рег (проталкивание содержимого регистра в стек процессора).
рег ← PC (регистр PC содержит адрес ячейки, следующей за инструкцией JSR; теперь этот адрес помещается в регистр **рег**).
PC ← (назн) (теперь регистр PC указывает на подпрограмму назначения).

Описание: При выполнении этой инструкции исходное содержимое указанного регистра (связующего указателя) автоматически проталкивается в процессорный стек, а в регистр помещается новая связующая информация. Таким образом, все подпрограммы, вложенные в другие подпрограммы на любую глубину, могут вызываться с помощью одного и того же связующего регистра. Нет нужды ни в планировании того, на какой максимальной глубине будет вызываться конкретная подпрограмма, ни в том, чтобы включать в каждую подпрограмму инструкции для сохранения и восстановления содержимого связующего регистра. Более того, поскольку связующая информация сохраняется в процессорном стеке способом, обеспечивающим повторную входимость, выполнение подпрограммы может быть прервано и та же самая подпрограмма может быть вызвана вновь и выполнена подпрограммой обработки прерываний. Затем (после удовлетворения всех других запросов) выполнение подпрограммы может быть возобновлено. Такой процесс (называемый вложением) может продолжаться на любую глубину.

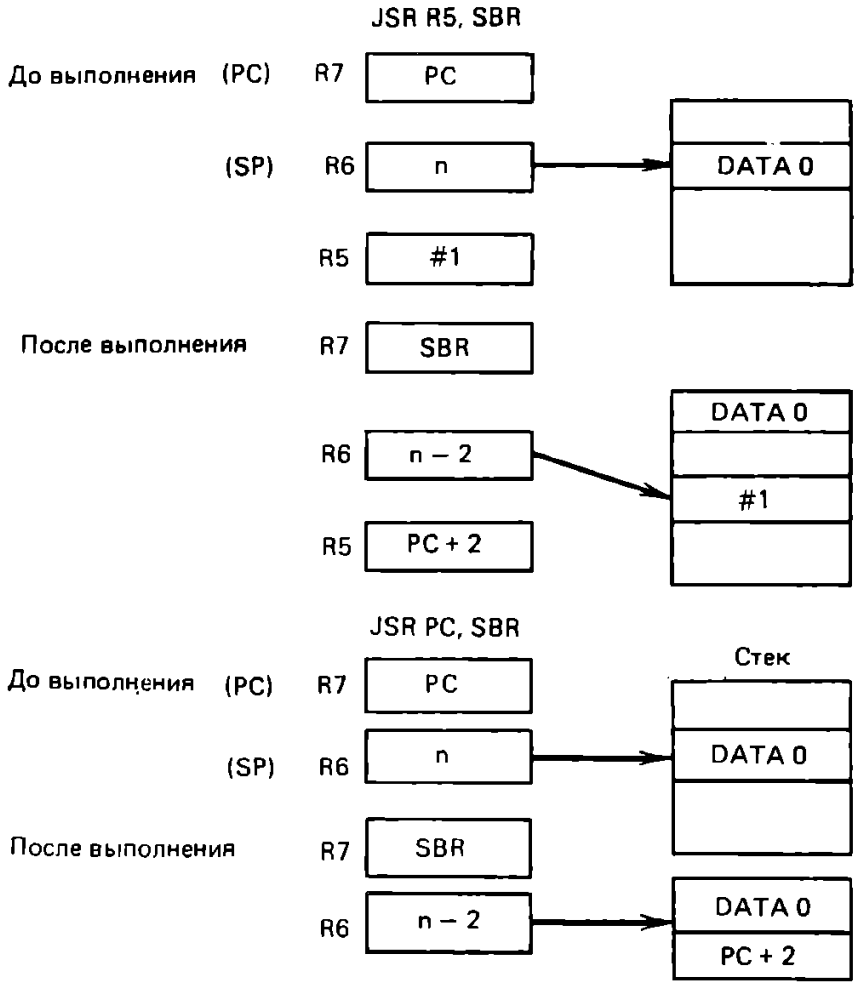
Подпрограмма, вызванная инструкцией JSR рег, назн, может иметь доступ к аргументам, расположенным после инструкции вызова, или с помощью режима адресации с автоувеличением (рег)+ (если доступ к аргументам осуществляется последовательно), или с помощью индексной адресации, X(рег) (при прямом доступе к аргументам). Режимы @(рег)+ И @X(рег) могут быть и косвенными, если параметры являются адресами операндов, а не самими операндами.

Инструкция JSP PC, назн – специальная, пригодная для вызова подпрограмм с передачей параметров через регистры общего назначения. При таком вызове могут изменяться только регистры SP и PC.

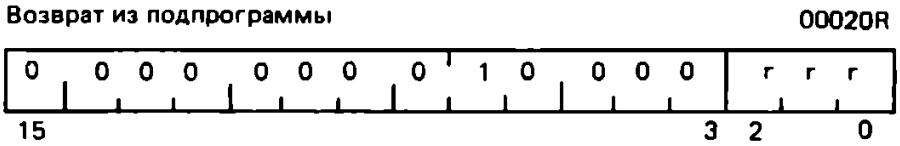
Другая специальная инструкция – JSR PC,@(SP)+, меняющая местами верхний элемент стека и содержимое программного счетчика. Применение такой инструкции дает возможность подпрограммам обмениваться управлением с последующим возобновлением выполнения с того места, откуда было передано управление партнеру. Такие подпрограммы называют сопрограммами.

Возврат из подпрограммы осуществляется с помощью инструкции RTS. Инструкция RTS рег загружает содержимое регистра **рег** в регистр PC и выталкивает в указанный регистр верхний элемент процессорного стека.

Пример:



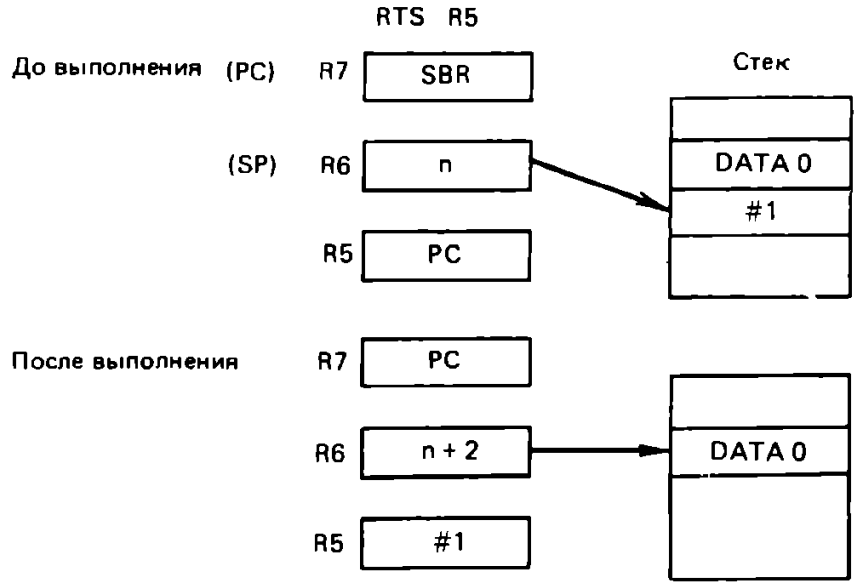
RTS



Операция: PC ← (рег)
(рег) ← (SP)↑

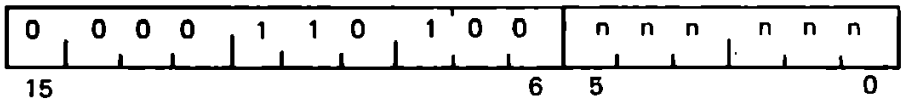
Описание: Загружает содержимое регистра в регистр PC и выталкивает в указанный регистр верхний элемент процессорного стека. Возврат из подпрограммы, не обладающей возможностью повторной входимости, обычно осуществляется через регистр, используемый для ее вызова. Таким образом, выход из подпрограммы, которая была вызвана инструкцией JSR PC, назн, выполняется с помощью инструкции RTS PC, а подпрограмма, вызванная инструкцией JSR R5, назн, может извлечь параметры с помощью режимов адресации (R5)+, X(R5) или @X(R5) и в конце концов осуществить выход с помощью инструкции RTS R5.

Пример:



MARK

Пометить стек 0064NN



Операция: $SP \leftarrow (PC)_{скор} + 2 + 2n$, где n – число параметров
$PC \leftarrow R5$
$R5 \leftarrow (SP)\uparrow$

Коды условий: Без изменения.

Описание: Входит составной частью в стандартное соглашение ЭВМ PDP-11 о возврате из подпрограммы. Облегчает выполнение очистки стека при выходе из подпрограммы. Ассемблерный формат: MARK N.

Пример:

```
MOV     R5,-(SP)        ;ПОМЕЩАЕТ В СТЕК СТАРОЕ СОДЕРЖИМОЕ R5
MOV     P1,-(SP)        ;ПОМЕЩАЕТ В СТЕК N ПАРАМЕТРОВ,
.                       ;КОТОРЫЕ БУДУТ ИСПОЛЬЗОВАТЬСЯ
.                       ;ПОДПРОГРАММОЙ
MOV     PN,-(SP)
MOV     #MARKN,-(SP)    ;ПОМЕЩАЕТ В СТЕК ИНСТРУКЦИЮ MARK N
MOV     SP,R5           ;УСТАНАВЛИВАЕТ АДРЕС ИНСТРУКЦИИ MARK N
JSR     PC,SUB          ;ПЕРЕХОД К ПОДПРОГРАММЕ
```

После выполнения этой инструкции стек имеет вид:

Подпрограмма начинается с адреса SUB.

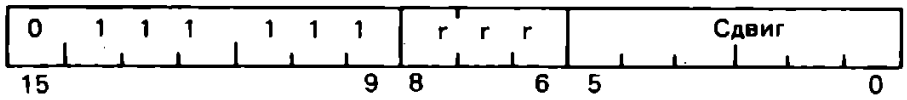
```
SUB:    ...             ;ВЫПОЛНЕНИЕ САМОЙ ПОДПРОГРАММЫ
        ...
        RTS     R5      ;ВОЗВРАТ ИЗ ПОДПРОГРАММЫ
```

Инструкция возврата приводит к тому, что содержимое регистра R5 помещается в регистр PC, тем самым выполняется инструкция MARK N. Исходное содержимое регистра PC помещается в регистр R5.

Выполнение инструкции MARK N приводит к тому, что: 1) указатель стека подстраивается так, что показывает на исходное содержимое регистра R5; 2) текущее содержимое регистра R5 (исходное содержимое регистра PC) помещается в регистр PC; 3) исходное содержимое регистра R5 выталкивается из стека в регистр R5, завершая таким образом возврат из подпрограммы.

SOB

Вычесть единицу и перейти (если не нуль) 077RNN



Операция: (R) ← (R) – 1; если этот результат не равен нулю, то PC ← PC – (2 X X сдвиг); если (R) = 0, PC ← PC

Коды условий: Без изменения.

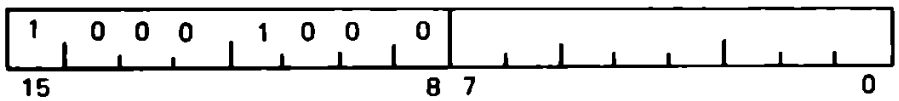
Описание: Содержимое указанного в инструкции регистра уменьшается. Если результат не равен нулю, то из содержимого регистра PC (показывающего уже на следующее слово) вычитается удвоенный сдвиг (сдвиг интерпретируется как 6-битовое положительное число). Эта инструкция обеспечивает быстрое, эффективное управление циклом. Ассемблерный синтаксис: SOB R, A, где A – адрес, на который должно передаваться управление, если уменьшенное содержимое регистра R не равно нулю. Обратите внимание, что эта инструкция не может использоваться для передачи управления по направлению вперед.

ЛОВУШКИ

Инструкции ловушек дают возможность вызова эмуляторов, мониторов ввода-вывода, отладочных пакетов и написанных пользователем интерпретаторов. Ловушка – это прерывание, порожденное программным обеспечением. При переходе к ловушке текущее содержимое программного счетчика и слова состояния процессора проталкиваются в стек и заменяются содержимым двухсловного вектора ловушки, содержащего новые значения регистра PC и слова состояния PSW. Последовательность возврата из ловушки содержит выполнение инструкции RTI или RTT, которая восстанавливает исходное содержимое регистра PC и слова состояния PSW, выталкивая их из стека. Векторы инструкций ловушек размещаются в памяти по заранее выделенным фиксированным адресам.

EMT

Перейти к ловушке эмулятора 104000 – 104377

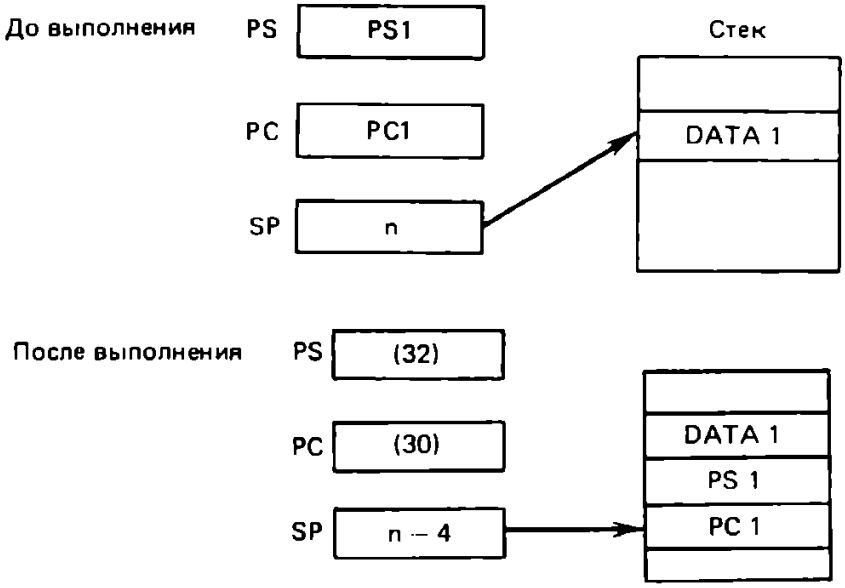


Операция: ↓(SP) ← PSW
↓(SP) ← PC
PC ← (30)
PSW ← (32)

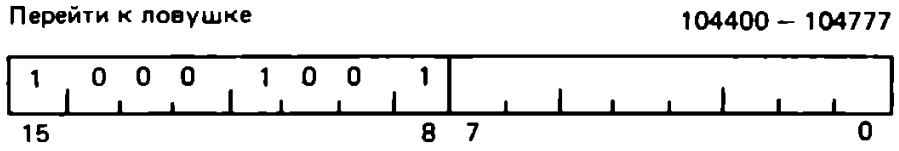
Коды условий: N: Загружается из вектора ловушки.
Z: Загружается из вектора ловушки.
V: Загружается из вектора ловушки.
C: Загружается из вектора ловушки.

Описание: Эта инструкция занимает коды 104000 – 104377; любой из этих кодов может использоваться для передачи информации эмулирующей подпрограмме (например, для передачи функции, которую требуется выполнить). Вектор ловушки для инструкции EMT расположен по адресу 30. Текущее содержимое регистра PC извлекается из слова с адресом 30, а текущее содержимое слова состояния PSW – из слова с адресом 32.

Предостережение. Инструкция EMT широко используется системным программным обеспечением фирмы DEC и поэтому не рекомендуется для общего применения.



TRAP



Операция: ↓(SP) ← PSW
↓(SP) ← PC
PC ← (34)
PSW ← (36)

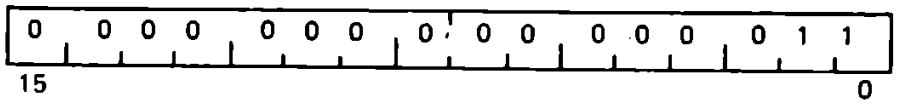
Коды условий: N: Загружается из вектора ловушки.
Z: Загружается из вектора ловушки.
V: Загружается из вектора ловушки.
C: Загружается из вектора ловушки.

Описание: Эта инструкция занимает коды 104400 – 104777. Она идентична инструкции EMT, за исключением того, что адрес ее вектора ловушки равен 34.

Замечание. Поскольку в системном программном обеспечении фирмы DEC очень часто используется инструкция EMT, для общего применения рекомендуется инструкция TRAP.

BPT

Перейти к ловушке точки прерывания 000003



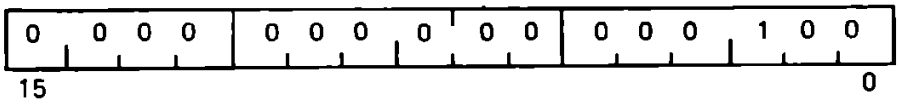
Операция: ↓(SP) ← PSW
↓(SP) ← PC
PC ← (14)
PSW ← (16)

Коды условий: N: Загружается из вектора ловушки.
Z: Загружается из вектора ловушки.
V: Загружается из вектора ловушки.
C: Загружается из вектора ловушки.

Описание: Выполняет последовательность перехода к ловушке с вектором ловушки по адресу 14. Применяется для вызова средств отладки. При работе с такими средствами пользователю следует воздерживаться от помещения в программу кода 000003. (В младшем байте никакой дополнительной информации не передается.)

IOT

Перейти к ловушке ввода-вывода 000004



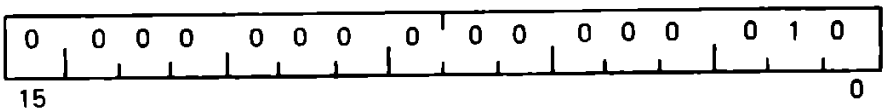
Операция: ↓(SP) ← PSW
↓(SP) ← PC
PC ← (20)
PSW ← (22)

Коды условий: N: Загружается из вектора ловушки.
Z: Загружается из вектора ловушки.
V: Загружается из вектора ловушки.
C: Загружается из вектора ловушки.

Описание: Выполняет последовательность перехода к ловушке с вектором ловушки по адресу 20. (В младшем байте никакой дополнительной информации не передается.)

RTI

Вернуться из прерывания 000002

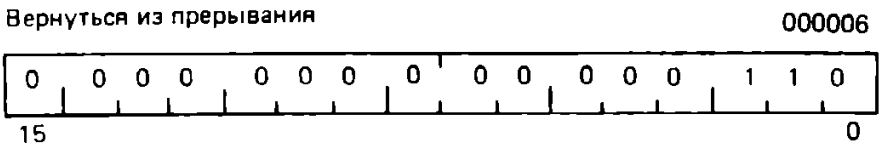


Операция: PC ← (SP)↑
PSW ← (SP)↑

Коды условий: N: Загружается из процессорного стека.
Z: Загружается из процессорного стека.
V: Загружается из процессорного стека.
C: Загружается из процессорного стека.

Описание: Применяется для выхода из подпрограммы обслуживания прерывания или ловушки. Содержимое регистра PC и слова состояния PSW восстанавливается из процессорного стека. Если установлен бит трассировки, то следующая за инструкцией RTI инструкция не будет выполняться перед последующими ловушками трассировки.

RTT



Операция: PC ← (SP)↑
PSW ← (SP)↑

Коды условий: N: Загружается из процессорного стека.
Z: Загружается из процессорного стека.
V: Загружается из процессорного стека.
C: Загружается из процессорного стека.

Описание: Операция такая же, как и в случае инструкции RTI, за исключением того, что запрещается ловушка трассировки. Если в новом слове состояния PSW установлен бит T, то переход к ловушке произойдет после выполнения следующей за инструкцией RTI инструкции.

Ловушки зарезервированных инструкций. Переход к такой ловушке вызывают попытки выполнить коды инструкций, зарезервированные для расширения возможностей процессора в будущем (зарезервированные инструкции), или инструкций с неправильными режимами адресации (неправильные инструкции). Порядковые коды, не соответствующие ни одной из описанных здесь инструкций, считаются зарезервированными инструкциями. Инструкции JMP и JSR с регистровым режимом для операнда назначения являются неправильными и вызывают переход к ловушке с вектором по адресу 4. Зарезервированные инструкции вызывают переход к ловушке с вектором по адресу 10.

Ловушки ошибок общей шины. Это ошибки тайм-аута, когда попытка обращения к адресу на шине не приводит к соответствующему ответу в пределах некоторого интервала времени. Такие ошибки происходят при попытке обращения к несуществующей памяти или к несуществующим периферийным устройствам. Эти ошибки вызывают переход к ловушке с вектором по адресу 4.

Ловушка трассировки. Ловушка трассировки разрешается битом 4 слова состояния процессора. Переход к ловушке происходит в конец выполнения инструкции. Выполнение инструкции, следующей за инструкцией, установившей бит T, будет доведено до конца, после чего произойдет переход к ловушке с вектором по адресу 14. Заметим, что ловушка трассировки является системным отладочным средством, которое для обычного программиста невидимо.

Замечание. Бит 4 в PSW может быть установлен только опосредованно – путем выполнения инструкции RTI или инструкции RTT с расположенным в стеке желаемым содержимым PSW.

Ниже перечисляются специальные ситуации, связанные с битом T, и затем дается их детальное описание.

1. Трассируемая инструкция очистила бит T.
2. Трассируемая инструкция установила бит T.
3. Трассируемая инструкция вызвала переход к ловушке по инструкции.
4. Трассируемая инструкция привела к ловушке по ошибке общей шины.
5. Работа процессора была прервана между моментами, когда был установлен бит T и когда была извлечена инструкция, подлежащая трассировке.
6. Трассируемой инструкцией была инструкция WAIT.
7. Трассируемой инструкцией была инструкция HALT.
8. Трассируемой инструкцией была инструкция возврата из прерывания.

Замечание. Трассируемой инструкцией является инструкция, следующая за инструкцией, установившей бит T.

Инструкция, очистившая бит Т. При извлечении трассируемой инструкции был установлен внутренний флаг трассировки. В конце выполнения этой инструкции опять произойдет переход к ловушке трассировки. Однако бит Т в слове состояния процессора, находящемся в стеке, будет очищенным.

Инструкция, установившая бит Т. Поскольку бит Т уже был установлен, его установка вновь никакого эффекта не дает. Произойдет переход к ловушке.

Инструкция, вызвавшая переход к ловушке по инструкции. Происходит переход к ловушке по инструкции, и полностью выполняется подпрограмма обслуживания ловушки. Если выход из подпрограммы обслуживания осуществляется с помощью инструкции RTI или помещенное в стек слово состояния процессора восстанавливается каким-либо другим способом, то бит Т опять устанавливается, инструкция следующая за трассируемой инструкцией, выполняется и (если она не относится к одному из упомянутых уже специальных случаев) происходит переход к ловушке трассировки.

Инструкция, вызвавшая переход к ловушке по ошибке общей шины. Эта ситуация трактуется так же, как ловушка по инструкции. Единственное отличие состоит в нежелательности того, чтобы подпрограмма обработки ошибки заканчивалась инструкцией RTI и был переход к трассировочной ловушке.

Обратите внимание, что прерывание может быть немедленно подтверждено после загрузки нового содержимого регистра PC и слова состояний PSW из вектора ловушки. Чтобы заблокировать все прерывания, нужно установить бит 7 в векторе ловушки.

Инструкция WAIT. Во время ожидания, вызванного этой инструкцией, на состояние бита Т процессор не реагирует.

Инструкция HALT. Процессор останавливается. Программный счетчик показывает на следующую инструкцию, которая может быть выполнена. Переход к ловушке произойдет сразу же после возобновления выполнения.

Инструкция возврата из прерывания. Эта инструкция или очищает, или устанавливает бит Т. Если бит Т был установлен и RTT является трассируемой инструкцией, то переход к ловушке будет задержан до завершения следующей инструкции.

Ловушка сброса питания. Переход к этой ловушке происходит, если во время нахождения процессора в активном режиме поступает сигнал сброса питания постоянным напряжением. Вектор ловушки размещается в ячейках 24 и 26. Переход к ловушке произойдет, если в подпрограмме обслуживания сброса питания выполняется инструкция RTI.

Приоритеты ловушек. В случаях, когда одновременно складываются условия нескольких внутренних и внешних процессорных ловушек, соблюдается порядок приоритетов:

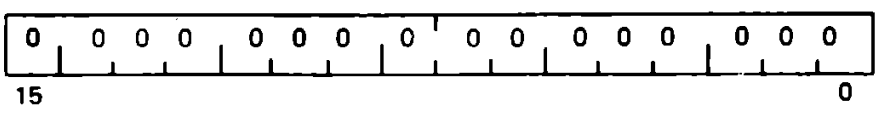
1) ловушка ошибок общей шины; 2) обновление памяти; 3) ловушки по инструкциям; 4) ловушка трассировки; 5) ловушка сброса питания; 6) линия останова; 7) линия внешнего прерывания; 8) запрос прерывания от устройства (на шине).

Если ошибка шины вызывается процессом обработки ловушки по инструкции, ловушки трассировки или предыдущей ошибки шины, процессор останавливается. Это называется двойной ошибкой шины.

Б.8. РАЗЛИЧНЫЕ ИНСТРУКЦИИ

HALT

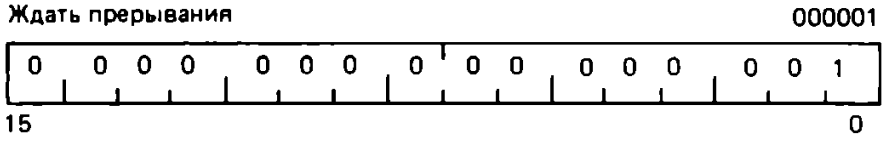
Остановить процессор 000000



Коды условий: Без изменения.

Описание: Выводит процессор из активного режима. Программный счетчик показывает на следующую инструкцию, которая может быть выполнена. Процессор переходит в режим останова. Содержимое регистра PC отображается на консольном терминале и дается разрешение на консольный режим работы.

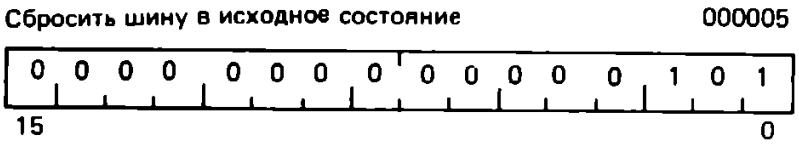
WAIT



Коды условий: Без изменения.

Описание: Предписывает процессору оставить использование общей шины на время, пока он ожидает поступления внешнего запроса прерывания. Получив инструкцию WAIT, процессор прекращает конкурировать за использование шины для извлечения из памяти инструкций или операндов. Поскольку процессор не вмешивается в работу шины и, следовательно, не задерживает работу, достижима наибольшая возможная скорость пересылки информации между каким-либо устройством и памятью. В этом случае, как обычно, регистр PC указывает на следующую за инструкцией WAIT инструкцию. Поэтому, когда прерывание приводит к проталкиванию в стек содержимого регистра PC и слова состояния PSW, адрес этой следующей инструкции сохраняется в стеке. При выходе из подпрограммы обслуживания прерываний (т. е. при выполнении инструкции RTI) прерванный процесс возобновляется с инструкции, следующей за инструкцией WAIT.

RESET

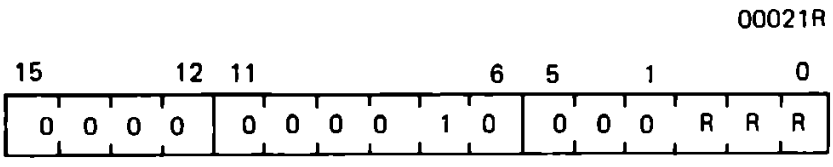


Коды условий: Без изменения.

Описание: Посылает на шину сигнал INIT в течение 10 мкс. Все устройства на шине сбрасываются в исходное состояние (в котором они оказываются после подачи питания). Вслед за выдачей сигнала INIT процессор остается неактивным в течение 90 мкс.

Б.9. ЗАРЕЗЕРВИРОВАННЫЕ ИНСТРУКЦИИ

(Мнемоника не присвоена)



Операция: (R) ← извлеченное содержимое пяти внутренних 16-битовых регистров; (R) ← (R) + 12 в конце инструкции.

Коды условий: Без изменения.

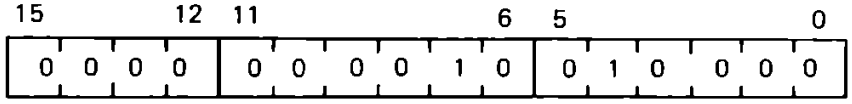
Описание: Содержимое регистра R (задаваемого тремя младшими битами инструкции) используется как указатель. Содержимое внутренних регистров, предназначенных для временной информации, заносится в после-

довательные ячейки памяти, а содержимое регистра R увеличивается на 2 до тех пор, пока не будет записано содержимое пяти 16-битовых регистров. (R) ← (R) + 12_8. В основном применяется в диагностических программах, используемых при эксплуатации ЭВМ. Эти пять слов памяти имеют следующую интерпретацию.

Ячейка памяти	Микроуровневый регистровый символ	Функция
(R)	RBA	Регистр адреса шины. Содержит последний адрес шины, использованный не для извлечения инструкции, а для режимов адресации операнда назначения 3, 5, 6 и 7.
(R) + 2	RSRC	Регистр исходного операнда. Содержит последний исходный операнд двухоперандной инструкции. При режиме адресации исходного операнда 0 старший байт может быть неверным.
(R) + 4	RDST	Регистр операнда назначения. Содержит последний операнда назначения, извлеченный процессором.
(R) + 6	RPSW	Сборный регистр. Старше 4 бита этого регистра проверяют биты 4—7 слова состояния процессора. Интерпретация остальных бит является функцией последней инструкции и в общем случае не может быть полезной.
(R) + 10	RIR	Регистр инструкции. Содержит текущую, а не предыдущую инструкцию, находящуюся в процессе выполнения, и имеет формат 36R. Число 360 появляется, поскольку к операционному коду инструкции прибавляется число 150 (21R + 150 = 36R) (из-за особенности микропрограммного декодирования инструкции).

(Мнемоника не присвоена)

00022N



Операция: Вызывает передачу микропрограммного управления к микроячейке 3000.

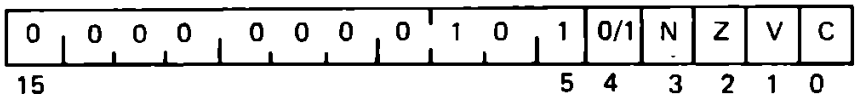
Коды условий: Без изменения.

Описание: Эта инструкция может использоваться для передачи микропрограммного управления микрокоду, размещенному по микроадресу 3000 в микропроцессоре. Если микроадрес 3000 не существует, этот операционный код вызывает переход к ловушке зарезервированных инструкций через ячейку памяти 10.

Б.10. ОПЕРАЦИИ НАД КОДАМИ УСЛОВИЙ

CLN SEN
CLZ SEZ
CLV SEV
CLC SEC
CCC SCC

Операции над кодами условий 0002XX



Описание: Эти инструкции устанавливают или очищают коды условий. Возможна совместная очистка или установка выборочных комбинаций этих бит. Биты кодов условий, соответствующие битам оператора, модифицируются в соответствии с состоянием бита 4 в операторе, определяющем, будет ли производиться установка или очистка бит условий. Установка бит условий, задаваемых битами 0, 1, 2 или 3 оператора, будет осуществляться, если бит 4 равен 1. Соответствующие биты условий будут очищаться, если бит 4 равен 0.

Мнемоника	Операция	Код операции
CLC	Очистить бит C	000241
CLV	Очистить бит V	000242
CLZ	Очистить бит Z	000244
CLN	Очистить бит N	000250
SEC	Установить бит C	000261
SEV	Установить бит V	000262
SEZ	Установить бит Z	000264
SEN	Установить бит N	000270
SCC	Установить все биты условий	000277
CCC	Очистить все биты условий	000257
	Очистить биты V и C	000243
NOP	Нет операции	000240

Вышеприведенные операции установки и очистки могут быть собраны вместе по операции ИЛИ для образования комбинированных инструкций.

ПРИЛОЖЕНИЕ В

ВРЕМЕННЫЕ ХАРАКТЕРИСТИКИ ИНСТРУКЦИИ ЭВМ PDP-11

В.1. ВРЕМЯ ВЫПОЛНЕНИЯ ИНСТРУКЦИИ ЭВМ LSI-11

Время выполнения инструкции зависит от ее операции, от использованных режимов адресации и типа памяти, с которой работает инструкция. В большинстве случаев время выполнения инструкции равно сумме основного времени, времени исходного адреса (**исх**) и времени адреса назначения (**назн**).

Время выполнения инструкции = основное время + время **исх** + время **назн** (основное время = время извлечения + время декодирования + время выполнения).

Для некоторых инструкций требуется только часть этого времени. Если не оговорено иное, вся временная информация далее дается в микросекундах. Приводятся типовые временные параметры; реальное время выполнения может варьироваться на ±20 %. Предполагается, что микроцикл равен 350 нс.

В.2. ВРЕМЯ ИСХОДНОГО АДРЕСА И АДРЕСА НАЗНАЧЕНИЯ

Режим	исх (слово)	исх (байт)	назн (слово)	назн (байт)
0	0	0	0	0
1	1,40	1,05	2,10	1,75
2	1,40	1,05	2,10	1,75
3	3,50	3,15	4,20	4,20
4	2,10	1,75	2,80	2,45
5	4,20	3,85	4,90	4,90
6	4,20	3,85	4,90	4,55
7	6,30	5,95	6,65	7,00

Замечание (режимы 2 и 4). Если в байтовой операции используется регистр R6 или R7, то следует прибавить 0,35 мкс ко времени **исх** и 0,70 мкс ко времени **назн**.

В.3. ОСНОВНОЕ ВРЕМЯ

Двухоперандные инструкции	РОН0	РОН1-7
MOV	3,50	2,45
ADD, XOR, SUB, BIC, BIS	3,50	4,20
CMP, BIT	3,50	3,15
MOVB	3,85	3,85
BICB, BISB	3,85	3,85
CMPB, BITB	3,15	2,80

Примечание. РОН0 — режим операнда назначения 0;
РОН1-7 — режимы 1—7 операнда назначения.

Режим адресации инструкции JMP или JSR	Время назн
1	0,70
2	1,40
3	1,75
4	1,40
5	2,45
6	2,45
7	4,20

Одноперандные инструкции	РОН0	РОН1-7
CLR	3,85	4,20
INC, ADC, DEC, SBC	4,20	4,90
COM, NEG	4,20	4,55
ROL, ASL	3,85	4,55
TST	4,20	3,85
ROR	5,25	5,95
ASR	5,60	6,30
CLRB, COMB, NEGB	3,85	4,20
ROLB, ASLB	3,85	4,20
INCB, DECB, SBCB, ADC	3,85	4,55
TSTB	3,85	3,50
RORB	4,20	4,90
ASRB	4,55	5,95
SWAB	4,20	3,85
SXT	5,95	6,65
MFPS (1067DD)	4,90	6,65
MTPS (1064SS)	7,00	7,00*

*Для инструкции MTPS нужно использовать время для байта **назн**, но не время **исх**. Если бит b_7 операции равен 1, то следует прибавить 0,35 мкс ко времени выполнения инструкции.

Инструкция	Основное время
JMP	3,50
JSR (регистр PC содержит адрес связи)	5,25
JSR (регистр PC не содержит адреса связи)	8,40
Все инструкции ветвления (независимо от выполнения условий)	3,50
SOB (с ветвлением)	4,90
SOB (без ветвления)	4,20
Установка кодов условий	3,50

Окончание табл.

Инструкция	Основное время
Очистка кодов условий	3,50
NOP	3,50
RTS	5,25
MARK	11,55
RTI	8,75*
RTT	8,75**
TRAP, EMT	16,80*
IOT, RPT	18,55*
WAIT	6,30
HALT	5,60
RESET	5,95 + 10,0 для INIT + + 90,0
Эксплуатационная инструкция (00021R)	20,30
Зарезервированная инструкция (00022N)	5,95

*Если в новом слове состояния PSW установлен бит 4 или бит 7, следует для каждого из этих бит прибавить 0,35 мкс.

**Если в новом слове состояния PSW установлен бит 4 (бит T), следует прибавить 2,10 мкс.

B.4. ВРЕМЕННЫЕ ХАРАКТЕРИСТИКИ ИНСТРУКЦИЙ РАСШИРЕННОЙ АРИФМЕТИКИ (KEV11)

ВРЕМЕННЫЕ ХАРАКТЕРИСТИКИ РАСШИРЕННОГО НАБОРА ИНСТРУКЦИИ (EIS)

Режим	Время исх в микросекундах
0	0,35
1	2,10
2	2,80
3	3,15
4	2,80
5	3,85
6	3,85
7	5,60

Инструкция	Основное время в микросекундах
MUL	от 24,0 до 37,0 (если оба числа меньше 256 по абсолютному значению при 16-битовом умножении) 64,0 (в наихудшем случае)
DIV	78,0 (в наихудшем случае)
ASH (вправо)	10,1 + 1,75 на сдвиг
ASH (влево)	10,8 + 2,45 на сдвиг
ASHC (вправо)	10,1 + 2,80 на сдвиг
ASHC (влево)	10,1 + 3,15 на сдвиг

Времена выполнения инструкций при операциях с плавающей точкой (FIS)

Время инструкции = основное время + время сдвига двоичных разрядов + время сдвига при нормализации

Инструкция	Основное время в микросекундах
FADD	42,1
FSUB	42,4

Разность экспонент	Выравнивание двоичных разрядов
0–7	2,45 на сдвиг
8–15	3,50 + 2,45 на сдвиг через 8
16–23	7,00 + 2,45 на сдвиг через 16

Разность экспонент	Нормализация
0–7	2,1 на сдвиг
8–15	2,1 + 2,1 на сдвиг через 8
16–23	4,2 + 2,1 на сдвиг через 16

Основное время выполнения инструкции в микросекундах	
FMUL	от 74,2 до 80,9 (если оба аргумента имеют 7-битовую точность, т. е. содержимое второго слова 32-битового аргумента равно 0)
FDIV	121,1 (в наихудшем случае, т.е. когда точность аргументов больше 7-битовой) 151 типичное значение 232 для наихудшего случая

В.5. СКРЫТАЯ ЗАДЕРЖКА ПРЯМОГО ДОСТУПА В ПАМЯТЬ

Скрытая задержка прямого доступа в память (ПДП), представляющая собой время от запроса до получения права владения шиной для первого устройства с ПДП, равна максимум 6,45 мкс. Это время самого длинного процессорного цикла DATIO, имеющего место для инструкции ASR с режимом адресации операнда назначения 1–7. Запрос ПДП удовлетворяется процессором во время обновления содержимого памяти.

В.6. СКРЫТАЯ ЗАДЕРЖКА ПРЕРЫВАНИЯ

Скрытая задержка прерывания имеет место в следующих случаях:

а) если процессор выполняет обновление памяти (независимо от того, присутствует ли блок KEV11):

время от запроса прерывания (BIRQ L) до подтверждения прерывания (BIAK L) равно максимум 118 мкс;

время от подтверждения (BIAK L) до извлечения первой инструкции подпрограммы обслуживания равно максимум 16,5 мкс; полное время от запроса до первой инструкции подпрограммы обслуживания равно максимум 134,5 мкс (сумма двух предыдущих времен);

б) если процессор не выполняет обновление памяти (и блок KEV11 не присутствует):

время от запроса прерывания (BIRQ L) до подтверждения прерывания (BIAK L) (самая длительная по времени выполнения инструкция – инструкция IOT) равно максимум 18,55 мкс;

время от подтверждения до извлечения первой инструкции подпрограммы обслуживания составляет еще максимум 16,5 мкс;

полное время от запроса прерывания до первой инструкции подпрограммы обслуживания равно максимум 35,05 мкс;

в) если процессор не выполняет цикл извлечения (и блок KEV11 присутствует):

время от запроса прерывания до подтверждения равно максимум 27,6 мкс;

время от подтверждения до извлечения первой инструкции подпрограммы обслуживания равно максимум 16,5 мкс;

полное время от запроса до первой инструкции подпрограммы обслуживания равно максимум 44,1 мкс.

Замечание. В процессе выполнения любой инструкции блока KEV11 (EIS и FIS) периодически просматриваются флаги запросов прерывания от устройств и событий. Если обнаружится запрос, то выполнение инструкции аварийно обрывается и восстанавливается состояние процессора в начале выполнения инструкции. После обработки прерывания прерванная инструкция выполняется повторно с самого начала. Следует обратить внимание на частоту прерываний по событиям, поскольку при слишком большой частоте инструкция блока KEV11 никогда не будет завершена. Предполагается, что, когда присутствует блок KEV11, максимальная частота прерываний по событиям не превышает 3,3 кГц. Без этого блока максимальная частота не должна превышать 20 кГц. В обоих случаях для выполнения подпрограммы обслуживания прерываний отводится примерно 50 мкс.

ПРИЛОЖЕНИЕ Г

СПИСОК ИНСТРУКЦИЙ ЭВМ PDP-11 В ПОРЯДКЕ ВОЗРАСТАНИЯ ОПЕРАЦИОННОГО КОДА

Код операции	Мнемоника	Код операции	Мнемоника	Код операции	Мнемоника
00 00 00	HALT	00 02 64	SEZ	00 67	SXT
00 00 01	WAIT	00 02 70	SEN	00 70 00	
00 00 02	RTI	00 02 77	SCC	.	Не используются
00 00 03	BPT	00 03 DD	SWAB	.	
00 00 04	IOT	00 04 XXX	BR	00 77 00	
00 00 05	RESET	00 10 XXX	BNE	01 SS DD	MOV
00 00 06	RTT	00 14 XXX	BEQ	02 SS DD	CMP
00 00 07		00 20 XXX	BGE	03 SS DD	BIT
.	Не используются	00 24 XXX	BLT	04 SS DD	BIC
.		00 30 XXX	BGT	05 SS DD	BIC
00 00 77		00 34 XXX	BLE	06 SS DD	BIS
00 01 DD	JMP	00 4R DD	JSR	07 0R SS	ADD
00 02 0R	RTS	00 50 DD	CLR	07 1R SS	MUL
00 02 10		00 51 DD	COM	07 2R SS	ASH
.	Не используются	00 52 DD	INC	07 3R SS	ASHC
.		00 53 DD	DEC	07 4R DD	XOR
.		00 54 DD	NEG	07 50 0R	FADD
00 02 27		00 55 DD	ADC	07 50 1R	FSUB
00 02 3	SPL	00 56 DD	SBC	07 50 2R	FMUL
00 02 40	NOP	00 57 DD	TST	07 50 3R	FDIV
00 02 41	CLC	00 60 DD	ROR	07 50 40	
00 02 42	CLV	00 61 DD	ROL	.	Не используются
00 02 44	CLZ	00 62 DD	ASR	.	
00 02 50	CLN	00 63 DD	ASL	.	
00 02 57	CCC	00 64 NN	MARK	07 67 77	
00 02 61	SEC	00 65 SS	MFPI	07 7R NN	SOB
00 02 62	SEV	00 66 DD	MTPI	10 00 XXX	BPL
10 04 XXX	BMI	10 51 DD	COMB	10 65 SS	MFPD
10 10 XXX	BHI	10 52 DD	INCB	10 66 DD	MTPD

Окончание табл.

Код операции	Мнемоника	Код операции	Мнемоника	Код операции	Мнемоника
10 14 XXX	BLOS	10 53 DD	DECB	10 67 00	Не используются
10 20 XXX	BVC	10 54 DD	NEGB	.	
10 24 XXX	BVS	10 55 DD	ADCB	.	
10 30 XXX	BCC, BHIS	10 56 DD	SBCB	10 77 77	
10 34 XXX	BCS, BLO	10 57 DD	TSTB	11 SS DD	MOVB
10 40 00	EMT	10 60 DD	RORB	12 SS DD	CMPB
.		10 61 DD	ROLB	13 SS DD	BITB
.		10 62 DD	ASRB	14 SS DD	BICB
10 43 77		10 63 DD	ASLB	15 SS DD	BISB
10 44 00	TRAP	10 64 00	Не используются	16 SS DD	SUB
.		.		17 00 00	Инструкции процессора для операций с плавающей точкой
.		.		.	
.		.		.	
10 47 77		.		.	
10 50 DD	CLRB	10 64 77		17 77 77	

ПРИЛОЖЕНИЕ Д

РАЗРАБОТКА ПРОГРАММ И СИСТЕМНОЕ ПРОГРАММНОЕ ОБЕСПЕЧЕНИЕ: ОПЕРАЦИОННАЯ СИСТЕМА RSX-11M

Когда мы вводили понятие системного программного обеспечения, то сравнивали программы операционной системы с административной системой ресторана. Ниже следует описание конкретной операционной системы для ЭВМ PDP-11, а именно RSX-11M. Для ЭВМ PDP-11 существует целый ряд операционных систем, ниже приведено описание ОС RSX-11M. Это довольно мощная многопользовательская дисковая операционная система реального времени. Однако у нас нет возможности изучить все ее детали, поэтому, чтобы Вы могли научиться использовать важные системные программные модули ОС для разработки и выполнения прикладных программ, познакомимся со структурой этой системы с точки зрения пользователя. Напоминаем, что Вы можете рассматривать конкретные системные программные модули как штатных сотрудников административной системы ресторана или как инструменты в вашем гараже. Для Вас важно освоить эти инструменты. В частности, для каждого модуля Вам необходимо знать имя этого модуля, его функции, как вызывать этот модуль с диска, когда он потребуется, как взаимодействовать с ним и как от него избавиться, если он больше не нужен. Прежде чем изучить характеристики каждого модуля, давайте посмотрим, каким образом мы можем взаимодействовать с такими модулями.

Д.1. ВЗАИМОСВЯЗЬ ПОЛЬЗОВАТЕЛЯ С ОС RSX-11M

КОМАНДЫ

Команды — это средства, с помощью которых пользователь или программист может указать "штатным сотрудникам", что они должны делать. Чтобы этого достичь, тот, кто приказывает (программист), должен знать язык, понятный тому, кто оказывает услуги (системные программы). Поскольку обычным устройством ввода-вывода, предназначенным для связи с системой, является терминал на ЭЛТ, команды должны

печататься на клавиатуре этого терминала. Конечно, каждой команде соответствует присущие только ей формат и функции (см. руководства для пользователей). Для удобства воспользуемся общим термином COMMAND. Тогда формат для выдачи команды будет следующим:

COMMAND <CR>

Символ <CR> в данном случае означает, что пользователь должен оканчивать команду нажатием на клавишу возврата каретки. Это необходимо потому, что программный модуль (штатный сотрудник) должен знать, с какого момента начинать интерпретацию и выполнение команды. Как только модуль "видит" код ASCII возврата каретки, он начинает выполнять команду, если она ему понятна. Очевидно, что во многих случаях команда очень похожа на мнемоническую инструкцию: для команды тоже требуется, чтобы был указан вместе с командой операнд (тогда модуль будет знать, к чему применять заданную команду). Например, если нужно, чтобы системная программа ассемблера преобразовала исходную программу (в мнемоническом виде), занимающуюся поиском нечетных чисел, в объектную программу (в двоичном машинном коде), то в команде нам придется указать ассемблеру, какую программу он должен преобразовывать. Общепринято, что отдельная программа помещается в файл с уникальным идентификатором.

ФАЙЛЫ

Концепция файла, используемого для идентификации отдельной программы, системной или прикладной, имеет важное значение в любой операционной системе. Имена файлов похожи на номера системы социального страхования, присваиваемые фирмой каждому работнику. Но файл работника будет идентифицироваться не только его номером социального страхования, но и его возрастом, полом, полным именем и т. п. В ОС RSX-11M файл формально определяется следующим образом:

устр: [г, ч] имя файла.тип; n

где **устр**: Δ имя дискового привода, на котором размещен файл;

[г, ч] Δ числовой идентификатор пользователя или программиста; **г** – номер группы пользователей, **ч** – номер члена группы. Используются числа, состоящие из трех цифр. Например, числа [220, 051] означают 220-ю группу и 51-го члена этой группы;

имя файла Δ определенное имя, назначенное программистом и содержащее от 0 до 9 печатных знаков (например, ODDNUM);

.тип Δ от 0 до 3 печатных знаков, назначенных пользователем. Хотя у пользователя есть возможность назначать любые знаки, существуют некоторые стандартные типы, имеющие определенный смысл, и поэтому пользователю не рекомендуется использовать их по другому назначению. Они определяются так:

.MAC – исходный файл микроассемблера;
.OBJ – выходной объектный файл макроассемблера;
.LST – файл листинга макроассемблера;
.TSK – выходной файл построителя задач;
.STB – файл таблицы символов;

; *n* Δ *n*-я версия одного и того же файла.

Для программы поиска нечетных чисел из гл.3 мы можем задать файл следующим образом:

DK0: [100, 101] BIGODD.MAC; 1

Эта запись означает, что первая версия исходной программы с именем BIGODD располагается на диске, установленном в настоящее время на дисководе DK0, и принадле-

жит 101-му члену группы 100. В спецификациях файлов можно использовать величины, заданные по умолчанию. Например, если не будет указан дисковый привод DK0, то по умолчанию система будет искать файл на дисководе 0 системного диска. Если пользователь осуществил вход в систему с терминала с помощью кода [100, 101], то этот код будет значением по умолчанию для идентификатора пользователя до тех пор, пока пользователь не осуществит выход из этой системы.

Д.2. ОСНОВНЫЕ ЭЛЕМЕНТЫ ОС RSX-11M

В этом разделе мы познакомим читателя с основными программными элементами или модулями, существенными при разработке и выполнении программ. Хотя в данном случае мы имеем дело с конкретной операционной системой, модули которой могут иметь свои индивидуальные названия, мы будем сравнивать их функции с аналогичными модулями в других операционных системах. Это позволит Вам в будущем без затруднений переходить к другим операционным системам. Другими словами, все операционные системы должны состоять из модулей, обеспечивающих похожее обслуживание пользователей, хотя индивидуальное имя каждого модуля может совершенно меняться при переходе от одной операционной системе к другой. Давайте рассмотрим модули ОС RSX-11M.

ПРОГРАММА КОНСОЛЬНОГО МОНИТОРА (MCR)

Самый первый модуль или системная программа, с которой программист встретится за терминалом, – MCR. Функции этого модуля заключаются в том, чтобы интерпретировать команды пользователя, напечатанные на клавиатуре терминала, и вырабатывать соответствующие ответы. С клавиатуры пользователь может вызвать любой модуль с помощью модуля MCR. Но сначала пользователь должен убедиться, что модуль MCR активен. О его активности будет свидетельствовать символ ">" на экране терминала, который называется подсказкой модуля MCR. Теперь пользователь может взаимодействовать с модулем MCR, вводя информацию с клавиатуры и читая ответы на ЭЛТ. Для большей ясности мы будем подчеркивать все ответы, чтобы пользователь мог отличить входные сообщения от ответных. Если пользователь хочет, например, узнать, какие из разработанных ранее программ находятся в настоящее время на диске, он может ввести следующее:

```
1) >              ;НА ЭКРАНЕ ВИДНА ПОДСКАЗКА MCR
2) >DIR (CR)      ;ПОЛЬЗОВАТЕЛЬ ВВОДИТ КОМАНДУ DIR (СОКРАЩЕНИЕ СЛОВА DIRECTORY)
                  ;И ЗАКАНЧИВАЕТ ВВОД КОМАНДНОЙ СТРОКИ НАЖАТИЕМ КЛАВИШИ
                  ;ВОЗВРАТА КАРЕТКИ. ПОСЛЕ ЭТОГО НА ЭКРАН БУДЕТ ВЫВЕДЕН
                  ;СПИСОК ФАЙЛОВ, НАХОДЯЩИХСЯ НА ДИСКЕ В НАСТОЯЩЕЕ ВРЕМЯ.
```

Есть целый ряд команд, которыми пользователь может воспользоваться для взаимодействия с модулем MCR. Подробности можно найти в руководствах для пользователей Вашей ЭВМ.

РЕДАКТОР ТЕКСТА (EDI)

EDI – модуль, имеющий два режима работы: командный режим и режим ввода. Командный режим предназначен прежде всего для того, чтобы пользователь мог исправлять неизбежные погрешности, допускаемые им при подготовке программы и при вводе с клавиатуры ее текста. Пользователь может выдавать редактору EDI команды для удаления, вставки или замены текста и т. п. В режиме ввода редактор запоминает в оперативной памяти строку за строкой все, что пользователь набирает на клавиатуре.

Пример:

```
)EDI<CR>          ;MCR ВЫДАЕТ НА ЭКРАН СВОЮ ПОДСКАЗКУ.
                  ;ПОЛЬЗОВАТЕЛЬ ВЫЗЫВАЕТ МОДУЛЬ РЕДАКТОРА ТЕКСТА,
                  ;ПЕЧАТАЯ EDI<CR>.
EDI>              ;НА ЭКРАНЕ - ПОДСКАЗКА РЕДАКТОРА ТЕКСТА
EDI>ADD.MAC<CR>   ;ПОЛЬЗОВАТЕЛЬ ПЕЧАТАЕТ ИМЯ ФАЙЛА ADD.MAC,
                  ;ПОКАЗЫВАЯ ТЕМ САМЫМ,
                  ;ЧТО ОН ХОЧЕТ РЕДАКТИРОВАТЬ ЭТОТ ФАЙЛ
```

(Заметим, что имя файла мы выбрали произвольно лишь в качестве примера.) Мы можем получить два возможных ответа:

1. Редактор находит файл ADD.MAC, загружает его в оперативную память (чтобы пользователь мог его модифицировать) и дает следующий ответ:

```
N LINES READ IN
PAGE 1
*                 ;ЭТО ПОДСКАЗКА РЕДАКТОРА, КОГДА
                  ;ОН РАБОТАЕТ В КОМАНДНОМ РЕЖИМЕ
```

2. Если указанный в команде файл не существует, редактор дает ответ:

```
CREATING NEW FILE
INPUT             ;ИНФОРМАЦИЯ ПОЛЬЗОВАТЕЛЯ О ТОМ, ЧТО
                  ;РЕДАКТОР НАХОДИТСЯ В РЕЖИМЕ ВВОДА
```

Существует целый ряд команд, с помощью которых пользователь может взаимодействовать с редактором текста. (Детальное описание команд можно найти в соответствующем руководстве для пользователей.) Например для перехода из командного режима в режим ввода пользователь может напечатать I <CR>, а для возврата в командный режим – просто нажать клавишу возврата каретки на новой строке. Если пользователь хочет осуществить выход из редактора текста, он может напечатать одну из следующих трех команд:

```
EXIT<CR>
EI<CR>
^Z<CR>            ;ЭТА СИМВОЛИКА ОЗНАЧАЕТ, ЧТО ОДНОВРЕМЕННО
                  ;НАЖИМАЮТСЯ КЛАВИША УПРАВЛЕНИЯ И КЛАВИША Z,
                  ;А ЗАТЕМ - КЛАВИША ВОЗВРАТА КАРЕТКИ
```

Тогда редактор EDI прекратит работу и будет вызван модуль MCR, который выдаст на экран свою подсказку. Работа редактора текста показана на рис. Д.1. Спецификации входного и выходного файлов одинаковы, за исключением того, что номер версии выходного файла равен номеру версии входного файла плюс единица, причем такое увеличение номера версии производится всякий раз, когда пользователь вызывает редактор и затем осуществляет выход из него, независимо от того, выполнялись какие-либо модификации текста или нет.

АССЕМБЛЕР (MAC)

Следующий шаг после редактирования исходного файла – трансляция этого исходного файла в объектный файл, содержащий двоичный машинный код. Трансляция выполняется модулем MAC. Ассемблер не только формирует объектный файл

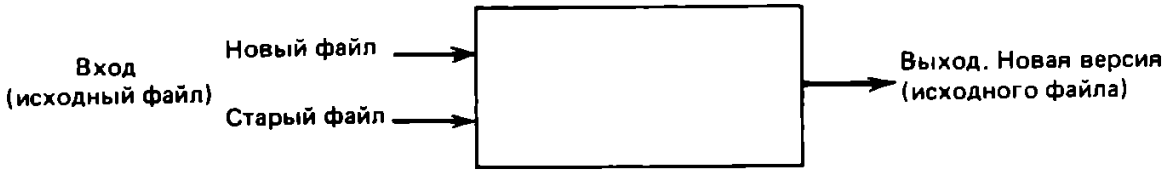


Рис. Д.1. Работа редактора текста

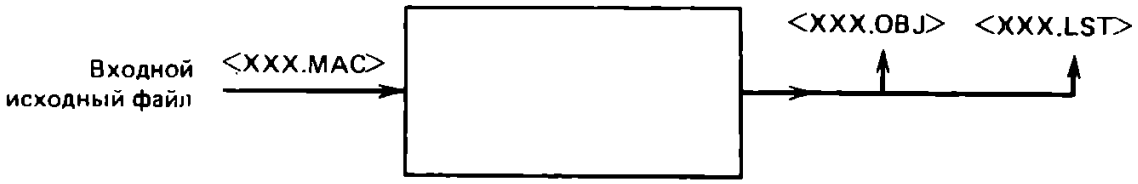


Рис. Д.2. Работа ассемблера

(как показано на рис. Д.2), но и строит два выходных файла: <XXX.OBJ> и <XXX.LST>. Угловые скобки служат для более четкого выделения имен; набирать на клавиатуре их не нужно.

Как и редактор текста, ассемблер MAC может быть вызван с помощью модуля MCR.

Пример 1:

```
)MAC(CR)          ;ВЫЗОВ АССЕМБЛЕРА С ПОМОЩЬЮ КОМАНДЫ МОДУЛЯ MCR
MAC>ADD.OBJ,ADD.LST=ADD.MAC(CR)
```

В данном случае пользователем указан входной файл с именем ADD.MAC (он должен быть исходным файлом) и ему нужно, чтобы ассемблер сгенерировал два выходных файла: объектный файл с именем ADD.OBJ и файл листинга с именем ADD.LST. В ответ на эту команду макроассемблер построит соответствующий объектный файл и запишет его на диск для последующего использования. В то же время он сгенерирует файл листинга, который тоже будет записан на диск. При желании его можно распечатать на АЦПУ, чтобы в будущем можно было обращаться к его содержимому. В файле листинга рядом приводятся объектный код и исходный код, что дает возможность пользователю одновременно видеть оба файла, объектный и исходный. Кроме того, файл листинга отображает таблицу символьных адресов; состояние памяти; ошибки программиста, если они есть (такие, как неправильно напечатанные инструкции, дублированные символьные адреса и т. п., но не ошибки в логике программы).

Если копия листинга на бумаге нам не нужна, можно использовать следующий формат:

MAC > ADD.OBJ, TI: = ADD.MAC <CR>

В этом случае файл листинга будет показан на экране терминала. В большинстве систем возможно использование формата по умолчанию:

≥MAC ADD, ADD = ADD <CR>

Пример 2. Команда

≥MAC ADD = ADD <CR>

построит только объектный файл.

Пример 3. Команда

≥MAC, ADD = ADD <CR>

построит только файл листинга.

ПОСТРОИТЕЛЬ ЗАДАЧ (ТКВ)

После построения объектного файла ассемблером мы должны быть готовы выполнить следующее действие: загрузить его в оперативную память. На практике, однако, не всегда это можно сделать непосредственно. Например, во многих случаях нам бы хотелось для выполнения одной задачи связать вместе несколько объектных файлов. Для этой цели и предназначен модуль построителя задач или компоновщик. В ОС RSX-11M такой модуль принимает на входе один или несколько объектных файлов и порождает выходные файлы трех типов: <XXX.TSK>, <XXX.MAP> и <XXX.STB>,

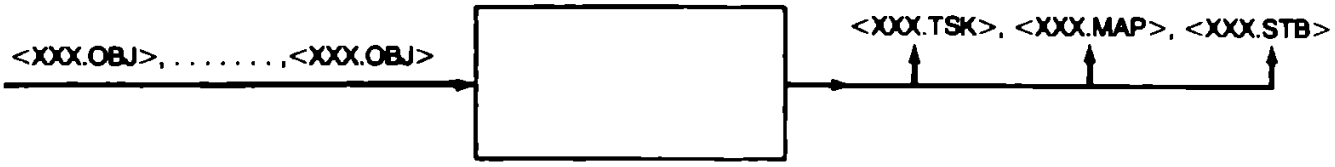


Рис. Д.3. Работа построителя задач

как это показано на рис. Д.3. Причем файл <xxx.TSK> содержит собранную программу, которая может быть загружена в память для выполнения. Его называют также файлом образа памяти. Файл <xxx.MAP> содержит карту распределения памяти, а файл <xxx.STB> – таблицу символов, созданную построителем задач.

Пример:

≥TKB ADD.TSK, ADD.MAP, ADD . STB = ADD.OBJ <CR>

Эта команда создает три выходных файла. Если нам нужен только файл образа памяти и мы хотим воспользоваться форматом по умолчанию, то можем напечатать

≥TKB ADD = ADD <CR>

и в результате получим файл ADD.TSK.

МОДУЛЬ ЗАГРУЗКИ И ВЫПОЛНЕНИЯ (LGO)

Созданный файл <xxx.TSK> тоже запоминается на диске под определенным именем. Для выполнения программы этот файл должен быть загружен с диска в оперативную память, начиная с определенной ячейки. В разных системах для этой цели выделены разные ячейки, но в любом случае загрузка файла образа памяти в нужные ячейки и запуск программы осуществляется с помощью модуля загрузки и выполнения, который вызывается с помощью модуля MCR.

Пример 1. Вызов модуля LGO, загрузка и запуск программы ADD:

≥LGO <CR>
LGO> ADD.TSK <CR>

или просто

LGO >ADD <CR>

Пример 2. В некоторых случаях пользователю нужно разбить процесс загрузки и запуска программы (LGO) на два шага. Это можно сделать следующим образом:

```
>GET<CR>          ;ВЫЗОВ МОДУЛЯ GET
GET>ADD<CR>       ;ПРИКАЗ МОДУЛЮ GET ЗАГРУЗИТЬ ФАЙЛ ADD
[LOADED]          ;МОДУЛЬ ИНФОРМИРУЕТ ПОЛЬЗОВАТЕЛЯ, ЧТО
                  ;ПРОГРАММА ЗАГРУЖЕНА
^Z                ;НАЖАТИЕ КЛАВИШИ УПРАВЛЕНИЯ И КЛАВИШИ Z
                  ;ПРИВОДИТ К ОТКАЗУ ОТ МОДУЛЯ GET
>BREAK            ;НАЖАТИЕ КЛАВИШИ РАЗРЫВА ПРИВОДИТ
                  ;К ВЫЗОВУ ОТЛАДЧИКА ODT
@                 ;ПОДСКАЗКА МОДУЛЯ ODT
```

В этом месте пользователь может вручную выполнить программу, запуская ее с определенной ячейки. Например, если мы хотим запустить программу, начиная с ячейки 1172, то можем сделать следующее:

@1172G <CR>

Буква G в данном случае является командой запуска GO для модуля отладчика ODT, краткое описание которого будет дано ниже.

ПРОГРАММА ОБМЕНА МЕЖДУ ПЕРИФЕРИЙНЫМИ УСТРОЙСТВАМИ (МОДУЛЬ PIP)

Этот модуль может применяться для пересылки программ или манипуляции ими на файловом уровне. С его помощью можно копировать, уничтожать, объединять или распечатывать файлы, а также выполнять многие другие операции, удобные пользователю. Модуль PIP вызывается с помощью модуля MCR. Общий формат командной строки:

≥PIP <спецификация файла> /xx <CR>

где /xx обозначает переключатель, указывающий, что модуль PIP должен выполнить некую необязательную функцию. Примеры использования модуля PIP и переключателей:

```
>PIP(СПЕЦИФИКАЦИЯ_ФАЙЛА)/DE(CR)          ;УНИЧТОЖИТЬ С ДИСКА
                                          ;УКАЗАННЫЙ ФАЙЛ
>PIP(СПЕЦИФИКАЦИЯ_ФАЙЛА)/PU(CR)          ;УНИЧТОЖИТЬ ВСЕ ВЕРСИИ
                                          ;ФАЙЛА КРОМЕ ПОСЛЕДНЕЙ
>PIP /LI(CR)              ;РАСПЕЧАТАТЬ КАТАЛОГ ВСЕХ ФАЙЛОВ,
                          ;СОЗДАННЫХ ДАННЫМ ПОЛЬЗОВАТЕЛЕМ
>PIP(СПЕЦИФИКАЦИЯ_ФАЙЛА)=(СПЕЦИФИКАЦИЯ_ФАЙЛА)(CR)
                          ;СКОПИРОВАТЬ ВХОДНОЙ ФАЙЛ В ВЫХОДНОЙ ФАЙЛ,
                          ;ТАК ЧТО СОДЕРЖИМОЕ ОБОИХ ФАЙЛОВ
                          ;БУДЕТ ОДНИМ И ТЕМ ЖЕ
```

ВОСЬМЕРИЧНЫЙ ОТЛАДЧИК (ODT)

Модуль ODT, известный также как оперативный отладчик, позволяет пользователю вносить временные поправки в его программу. С помощью отладчика можно посмотреть или изменить содержимое любого регистра ЦП и любого слова или байта оперативной памяти в любой заданной ячейке, причем данные приводятся в восьмеричном представлении. Отладчик дает также возможность пользователю осуществлять одношаговое выполнение, т. е. позволяет управлять выполнением программы. Предположим, например, что файл типа TSK был загружен с помощью модуля GET и после нажатия клавиши разрыва BREAK на экране ЭЛТ появилась подсказка отладчика @. Если теперь мы введем число (например, 1172) и нажмем клавишу косой черты (/), то отладчик откроет ячейку памяти 1172 и покажет ее содержимое:

@1172/000000

В данном случае содержимое ячейки оказалось равным нулю. Если мы хотим изменить его на 012345, то должны сделать следующее:

@1172/000000 012345 <CR>

Тогда указанный восьмеричный код (012345) будет записан в ячейку 1172. Мы можем проверить это, вновь открыв ячейку 1172:

@1172/012345

Аналогично можно посмотреть и изменить содержимое любого регистра ЦП. Например:

@R1/034750

(Был открыт регистр R1 и его содержимое оказалось равным 034750.)

СВЯЗЬ МОДУЛЕЙ ОС RSX-11M

Мы познакомились с основными элементами или модулями ОС RSX-11M. Упор при этом был сделан на том, "что" и "почему", но не на подробностях того, "как". Чтобы воспользоваться этими модулями для разработки программ, Вам необходимо изучить соответствующие руководства для пользователей. Как новичка Вас может

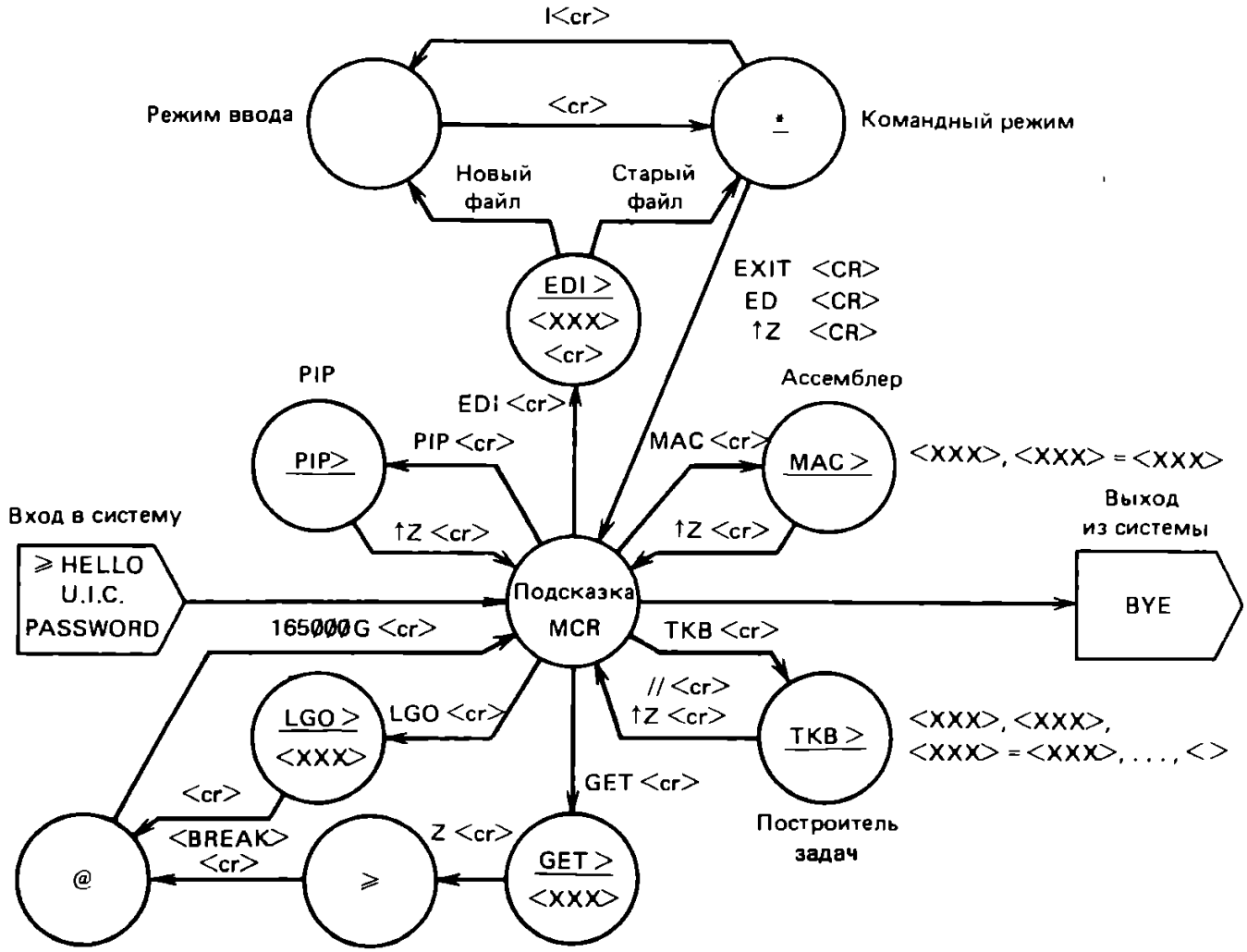


Рис. Д.4. Связь модулей ОС RSX-11M

смутить разнообразие всех этих модулей. На рис. Д.4 показана связь описанных модулей, которая даст Вам общую картину устройства операционной системы и послужит памяткой того, как вызывать каждый модуль и отказываться от его услуг.

Д.3. РАЗРАБОТКА ПРИКЛАДНЫХ ПРОГРАММ В СРЕДЕ ОС RSX-11M

ПРОЦЕДУРА РАЗРАБОТКИ ПРОГРАММЫ

Общая процедура разработки программы показана на рис. Д.5. Обратите внимание, что после блока "блок-схема" мы поместили блок "контроль логических ошибок". Это очень важный шаг, поскольку системные программы не способны обнаруживать логические ошибки в алгоритме программы. Проверив наличие логических ошибок в программе, мы можем начать подготовку первого варианта нашей исходной программы на языке ассемблера. Ниже мы опишем шаги, необходимые для использования модулей EDI, MAC, TKB и LGO или GET и GO. За информацией о том, как осуществить вход в эти модули и выход из них, обращайтесь к рис. Д.4.

Для иллюстрации процедуры разработки программы воспользуемся примером поиска нечетного числа из гл.3. Предположим, что мы имеем дело с первым черновым вариантом программы, записанным на бумаге. Тогда нам потребуется вызвать программу редактора текста с указанием спецификации файла, для чего нужно выполнить следующие шаги.

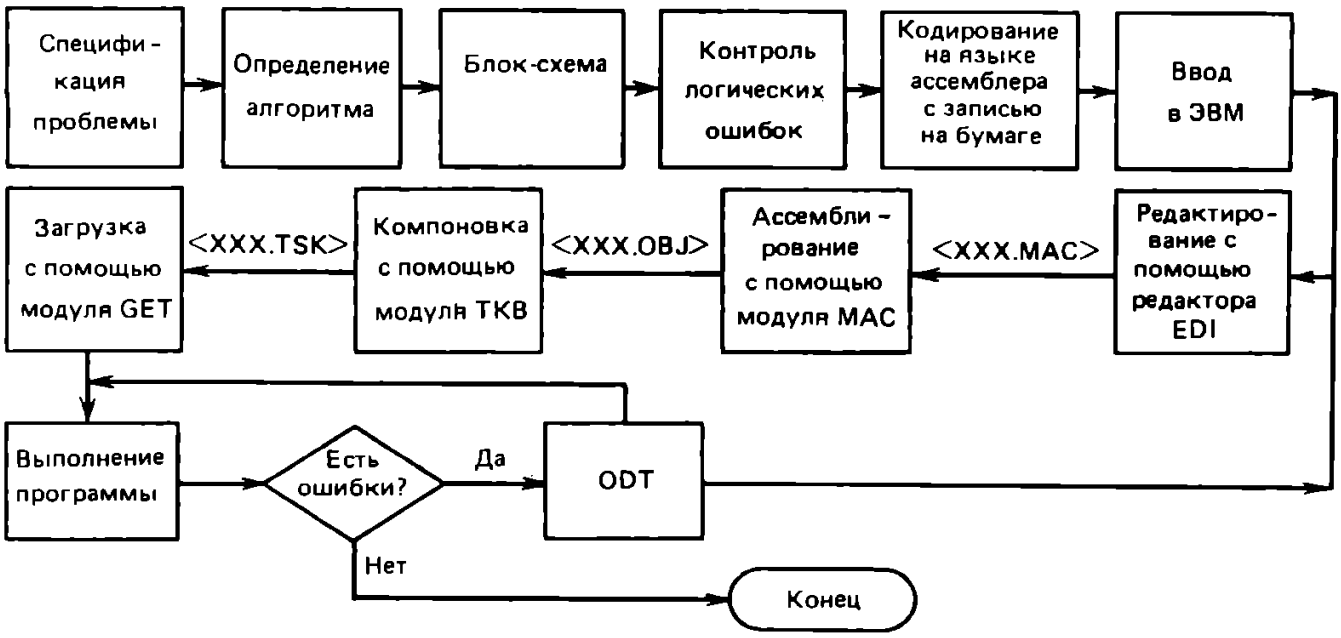


Рис. Д.5. Общая процедура разработки программы

Вхождение в систему. Процедура вхождения в систему может варьироваться от одной системы к другой, обычно она имеет следующий вид:

```
>HELLO<CR>
UIC                 ;ЗАПРОС КОДА ИДЕНТИФИКАЦИИ ПОЛЬЗОВАТЕЛЯ
PASSWORD            ;ЗАПРОС СЕКРЕТНОГО ПАРОЛЯ ПОЛЬЗОВАТЕЛЯ
>
```

Редактирование текста. Вызываем редактор текста

≥EDI BIGODD.MAC <CR>

Конечно, редактор не сможет найти такой файл и поэтому ответит

CREATING NEW FILE
INPUT

Это означает, что редактор готов воспринять все, что будет напечатано на клавиатуре терминала. Предположим, что теперь мы ввели через клавиатуру нашу подготовленную программу. После этого обычно осуществляют ассемблирование, но если мы хотим иметь на бумаге копию того, что напечатали на клавиатуре, то можем воспользоваться модулем PIP:

≥PIP LP:= BIGODD.MAC <CR>

Результат выполнения этой команды показан на рис. Д.6.

Ассемблирование программы. Чтобы сгенерировать объектный файл и файл листинга, нужно напечатать

≥MAC BIGODD.OBJ, BIGODD.LST = BIGODD.MAC <CR>

Распечатка файла листинга показана на рис. Д.7. В ней содержатся объектный (*а*) и исходный (*б*) коды программы, таблица символов и информация об обнаруженных ошибках в исходном коде. Напоминаем, что ассемблер не может обнаружить логических ошибок. Поскольку из распечатки видно, что ошибки не найдены, мы можем перейти к следующему шагу.

```
                ;THE BIG ODD NUMBER SORT
                ;
                ;
START:  MOV     N1,R1           ;(N1) → R1
        MOV     N2,R2           ;(N2) → R2
        MOV     #1,R0           ;R0 IS THE LSB EXAMINATION MASK
EXAM1:  BIT     R0,R1           ;EXAMINE LSB OF N1
        BNE     ODD2            ;IF N1 IS ODD, CHECK N2
EXAM2:  BIT     R0,R2           ;EXAMINE LSB OF N2
        BNE     STON2
        BR      QUIT
ODD2:   BIT     R0,R2
        BNE     COMPR           ;BOTH N1 N2 ARE ODD, SO COMPARE
STON1:  MOV     R1,ODDBIG
        BR      QUIT
COMPR:  CMF     R1,R2
        BGE     STON1
STON2:  MOV     R2,ODDBIG
QUIT:   HALT
ODDBIG: .BLKW 1
N1:     .WORD 145
N2:     .WORD 111
        .END    START
```

Рис. Д.6. Пример получения бумажной копии программы BIGODD с помощью модуля PIP

```
 1                                      ;THE BIG ODD NUMBER SORT
 2                                      ;
 3                                      ;
 4 000000  016701  000050   START:  MOV   N1,R1          ;(N1) → R1
 5 000004  016702  000046            MOV   N2,R2          ;(N2) → R2
 6 000010  012700  000001            MOV   #1,R0          ;R0 IS THE LSB EXAMINATION MASK
 7 000014  030001           EXAM1:   BIT   R0,R1          ;EXAMINE LSB OF N1
 8 000016  001003                    BNE   ODD2           ;IF N1 IS ODD, CHECK N2
 9 000020  030002           EXAM2:   BIT   R0,R2          ;EXAMINE LSB OF N2
10 000022  001010                    BNE   STON2
11 000024  000411                    BR    QUIT
12 000026  030002           ODD2:    BIT   R0,R2
13 000030  001003                    BNE   COMPR          ;BOTH N1 N2 ARE ODD, SO COMPARE
14 000032  010167  000014   STON1:   MOV   R1,ODDBIG
15 000036  000404                    BR    QUIT
16 000040  020102           COMPR:   CMP   R1,R2
17 000042  002373                    BGE   STON1
18 000044  010267  000002   STON2:   MOV   R2,ODDBIG
19 000050  000000           QUIT:    HALT
20 000052                   ODDBIG:  .BLKW 1
21 000054  000145           N1:      .WORD145
22 000056  000111           N2:      .WORD111
23         000000'                   .END  START
```

а)

```
COMPR  000040R    N1     000054R    ODDBIG  000052R    QUIT    000050R
EXAM1  000014R    N2     000056R    ODD2    000026R    START   000000R
EXAM2  000020R    STON1  000032R    STON2   000044R

  ABS.  000000    000
        000060    001
ERRORS DETECTED:  0

        000060    001
   000056R    ODD2
DYNAMIC MEMORY: 3086 WORDS ( 11 PAGES)
ELAPSED TIME: 00:00:04
.BIGODD=BIGODD
```

б)

Рис. Д.7. Файл листинга программы BIGODD

Компоновка программы. Здесь мы используем сгенерированный ассемблером объектный файл в качестве входного файла для построителя задач с целью получения файла образа памяти типа TSK с помощью следующей команды:

≥TKB BIGODD.TSK = BIGODD.OBJ <CR>

или просто

≥TKB BIGODD = BIGODD <CR>

Такой командой создается файл BIGODD.TSK. И мы переходим к следующему шагу.

Загрузка и выполнение. Имея построенный файл типа TSK, мы можем загрузить этот файл в оперативную память и запустить программу

≥LGO BIGODD <CR>

Если в нашем алгоритме нет логических ошибок, то в этом месте программа поместит в ячейку с символьным адресом ODDBIG наибольшее нечетное число 145. Для проверки правильности работы программы нам нужно посмотреть содержимое ячейки ODDBIG. Сделать это можно двумя способами: 1) используя команду отображения для выдачи содержимого ячейки ODDBIG на экран ЭЛТ; 2) используя отладчик ODT. Давайте воспользуемся отладчиком и просмотрим каждую ячейку памяти, включая и ODDBIG.

Отладка программы. Из рис. Д.7 видно, что объектный код начинается с адреса 000000. Однако программа не может загружаться с этого адреса, поскольку область памяти от нулевого адреса до некоторого определенного адреса резервируется для нужд системного программного обеспечения. Поэтому построитель задач ТКВ, ответственный за генерацию образа памяти, должен назначать адреса так, чтобы прикладная программа загружалась после этого определенного адреса, который может варьироваться от одной системы к другой.

Предположим, что в нашей системе область памяти для прикладных программ начинается с адреса 001172. Другими словами, область памяти между адресами 000000 и 001172 зарезервирована под системные операции, а наша программа будет загружаться в область, начинающуюся с адреса 001172. В результате значения символьных адресных меток в программе должны быть сдвинуты на 1172. Тогда например, START = = 1172, EXAM1 = 14 + 1172 = 1206 и т. д. При использовании отладчика ODT для проверки программы мы должны знать абсолютные или фактические адреса инструкций и печатать их на клавиатуре, сопровождая косой чертой. Тогда отладчик будет отображать на экране содержимое нужных нам ячеек памяти. На рис. Д.8 показано содержимое памяти, отведенной для нашей программы, до и после ее выполнения. Обратите внимание, что до выполнения программы содержимое ячейки ODDBIG равно 52 + + 1172, т. е. 000000 или просто "мусор", после выполнения программы это содержимое равно 000145. Рекомендуем вам сравнить это содержимое с содержимым на рис. Д.7.

Документирование программы. Создание программной документации зачастую игнорируется программистом. Обычно программист глубоко поглощен самим процессом разработки программы и увлечен только тем, чтобы заставить ее работать. Когда же наконец программа работает правильно, программист переходит к следующему проекту и за короткое время забывает детали первой программы и даже может не узнать ее. Поэтому крайне важно, чтобы каждая разработанная программа и каждый модуль были хорошо документированы.

Ниже мы приводим простые принципы документирования для примера из гл.3, предлагаемые для всех программистов.

1. Титул. В нем обычно содержатся ключевые слова программы. Для титула может использоваться псевдоинструкция или директива .TITLE. Например, .TITLE BIGODD.
2. Дата разработки: 13 января 1983 г.

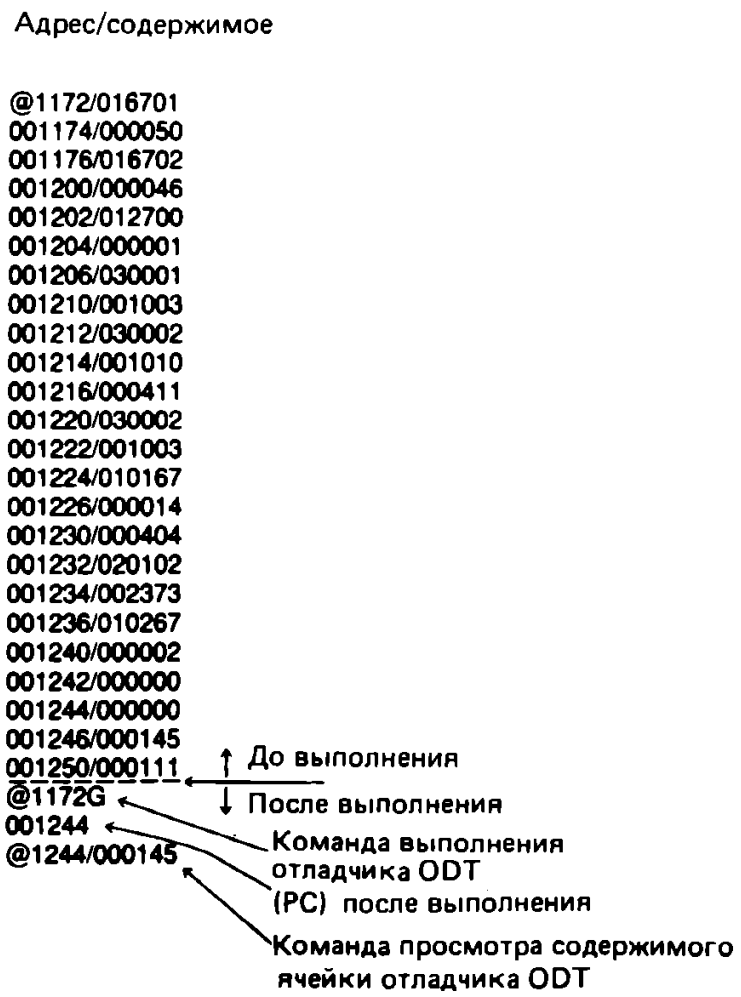


Рис. Д.8. Проверка и выполнение программы BIGODD с помощью отладчика ODT

3. Автор: В. Лин.
4. Дата модификации программы: 14 февраля 1983 г.
5. Функциональное описание: этот программный модуль находит наибольшее нечетное число и помещает его в выделенную для этого ячейку памяти.
6. Входные переменные: символьные адреса N1, N2.
7. Выходные переменные: символьный адрес ODDBIG.
8. Комментированный листинг программы: см. рис. Д.6.

Д.4. УПРАЖНЕНИЯ

1. Напишите простую программу на языке ассемблера ЭВМ PDP-11, которая будет вычислять среднее значение двух чисел, размещенных, начиная с ячейки INTEGER, и помещать его в ячейку с меткой AVERAGE. (*Подсказка*: инструкция арифметического сдвига вправо ASR осуществляет деление числа на 2 при каждом сдвиге.)
2. Используя программу, созданную в предыдущем упражнении, осуществите шаг за шагом процедуру разработки программы, показанную на рис. Д.5. Представьте, что вы работаете на ЭВМ, и опишите те команды, которые вам пришлось бы выдавать, и те ответы, которые вы ожидаете увидеть на экране ЭЛТ. Это позволит вам более четко представить себе шаги, необходимые для разработки и выполнения прикладной программы. Вы можете воспользоваться отладчиком ODT и поменять два целых числа, чтобы убедиться, что программа хорошо работает с любыми целыми числами.

ПРИЛОЖЕНИЕ Е

УКАЗАТЕЛЬ ИНСТРУКЦИЙ ЭВМ VAX-11, СОСТАВЛЕННЫЙ В АЛФАВИТНОМ ПОРЯДКЕ МНЕМОНИК

Мнемоника	Инструкция	Код операции
ACBB	Прибавить, сравнить и перейти (байтовая)	9D
ACBD	Прибавить, сравнить и перейти (двойной точности формата D)	6F
ACBF	Прибавить, сравнить и перейти (формата F чисел с плавающей точкой)	4F
ACBG	Прибавить, сравнить и перейти (формата G чисел с плавающей точкой)	4FFD
ACBH	Прибавить, сравнить и перейти (формата H чисел с плавающей точкой)	6FFD
ACBL	Прибавить, сравнить и перейти (длинного слова)	F1
ACBW	Прибавить, сравнить и перейти (словная)	3D
ADAWI	Прибавить выровненное слово (с блокировкой)	58
ADDB2	Прибавить байт (двухоперандная)	80
ADDB3	Прибавить байт (трехоперандная)	81
ADDD2	Прибавить число с плавающей точкой (формата D, двухоперандная)	60
ADDD3	Прибавить число с плавающей точкой формата D (трехоперандная)	61
ADDF2	Прибавить число с плавающей точкой формата F (двухоперандная)	40
ADDF3	Прибавить число с плавающей точкой формата F (трехоперандная)	41
ADDG2	Прибавить число с плавающей точкой формата G (двухоперандная)	40FD
ADDG3	Прибавить число с плавающей точкой формата G (трехоперандная)	41FD
ADDH2	Прибавить число с плавающей точкой формата H (двухоперандная)	60FD
ADDH3	Прибавить число с плавающей точкой формата H (трехоперандная)	61FD
ADDL2	Прибавить длинное слово (двухоперандная)	C0
ADDL3	Прибавить длинное слово (трехоперандная)	C1
ADDP4	Прибавить упакованные числа (четырехоперандная)	20
ADDP6	Прибавить упакованные числа (шестиоперандная)	21
ADDW2	Прибавить слово (двухоперандная)	A0
ADDW3	Прибавить слово (трехоперандная)	A1
ADWC	Прибавить с переносом	D8
AOBLEQ	Прибавить единицу и перейти, если меньше или равно	F3
AOBLSS	Прибавить единицу и перейти, если меньше	F2
ASHL	Арифметически сдвинуть длинное слово	78
ASHP	Арифметически сдвинуть и округлить упакованное число	F8
ASHQ	Арифметически сдвинуть четырехкратное слово	79
BBC	Ветвление, если бит очищен	E1
BBCC	Ветвление, если бит очищен, и очистка бита	E5

Мнемоника	Инструкция	Код операции
BBCCI	Ветвление, если бит очищен, и очистка бита (с блокировкой)	E7
BBCS	Ветвление, если бит очищен и установка бита	E3
BBS	Ветвление, если бит установлен	E0
BBSC	Ветвление, если бит установлен, и очистка бита	E4
BBSS	Ветвление, если бит установлен, и установка бита	E2
BBSSI	Ветвление, если бит установлен, и установка бита (с блокировкой)	E6
BCC	Ветвление, если бит переноса очищен	1E
BCS	Ветвление, если перенос установлен	1F
BEQL	Ветвление, если равно (с учетом знака)	13
BEQLU	Ветвление, если равно (без учета знака)	13
BGEQ	Ветвление, если больше или равно	18
BGEQU	Ветвление, если больше или равно (без учета знака)	1E
BGTR	Ветвление, если больше	14
BGTRU	Ветвление, если больше (без учета знака)	1A
BICB2	Очистка битовая (байтовая, двухоперандная)	8A
BICB3	Очистка битовая (байтовая, трехоперандная)	8B
BICL2	Очистка битовая (длинного слова, двухоперандная)	CA
BICL3	Очистка битовая (длинного слова, трехоперандная)	CB
BICPSW	Очистка битовая в слове состояния программы	B9
BICW2	Очистка битовая в слове (двухоперандная)	AA
BICW3	Очистка битовая в слове (трехоперандная)	AB
BISB2	Установка битовая в байте (двухоперандная)	88
BISB3	Установка битовая в байте (трехоперандная)	89
BISL2	Установка битовая в длинном слове (двухоперандная)	C8
BISL3	Установка битовая в длинном слове (трехоперандная)	C9
BISPSW	Установка битовая в слове состояния программы	B8
BISW2	Установка битовая в слове (двухоперандная)	A8
BISW3	Установка битовая в слове (трехоперандная)	A9
BITB	Проверка битовая в байте	93
BITL	Проверка битовая в длинном слове	D3
BITW	Проверка битовая в слове	B3
BLBC	Ветвление, если младший бит очищен	E9
BLBS	Ветвление, если младший бит установлен	E8
BLEQ	Ветвление, если меньше или равно	15
BLEQU	Ветвление, если меньше или равно (без учета знака)	1B
BLSS	Ветвление, если меньше	19
BLSSU	Ветвление, если меньше (без учета знака)	1F
BNEQ	Ветвление, если не равно	12
BNEQU	Ветвление, если не равно (без учета знака)	12
BPT	Точка прерывания по сбою	03
BRB	Ветвление с байтовым смещением	11
BRW	Ветвление со словным смещением	31
BSBB	Ветвление к подпрограмме с байтовым смещением	10
BSBW	Ветвление к подпрограмме со словным смещением	30

Мнемоника	Инструкция	Код операции
BUGL	Контроль ошибок в длинном слове	FDFF
BUGW	Контроль ошибок в слове	FEFF
BVC	Ветвление, если бит переполнения очищен	1C
BVS	Ветвление, если бит переполнения установлен	1D
CALLG	Вызов процедуры с общим списком аргументов	FA
CALLS	Вызов процедуры со списком аргументов в стеке	FB
CASEB	Инструкция переключения (байтовая)	8F
CASEL	Инструкция переключения (с длинным словом)	CF
CASEW	Инструкция переключения (со словом)	AF
CHME	Установить режим исполнителя	BD
CHMK	Установить режим ядра	BC
CHMS	Установить режим супервизора	BE
CHMU	Установить режим пользователя	BF
CLRB	Очистить байт	94
CLRD	Очистить число с плавающей точкой (формата D)	7C
CLRF	Очистить число с плавающей точкой (формата F)	D4
CLRG	Очистить число с плавающей точкой (формата G)	7C
CLRH	Очистить число с плавающей точкой (формата H)	7CFD
CLRL	Очистить длинное слово	D4
CLRO	Очистить восьмикратное слово	7CFD
CLRQ	Очистить четырехкратное слово	7C
CLRW	Очистить слово	B4
CMPB	Сравнить байты	91
CMPC3	Сравнить строки символов (трехоперандная)	29
CMPC5	Сравнить строки символов (пятиоперандная)	2D
CMPD	Сравнить числа с плавающей точкой (формата D)	71
CMPF	Сравнить числа с плавающей точкой (формата F)	51
CMPG	Сравнить числа с плавающей точкой (формата G)	51FD
CMPH	Сравнить числа с плавающей точкой (формата H)	71FD
CMPL	Сравнить длинные слова	D1
CMPP3	Сравнить упакованные числа (трехоперандная)	35
CMPP4	Сравнить упакованные числа (четырехоперандная)	37
CMPV	Сравнить битовое поле с целым числом	EC
CMPW	Сравнить слова	B1
CMPZV	Сравнить с целым числом расширенное битовое поле	ED
CRC	Вычислить значение циклического контроля по избыточности	OB
CVTBD	Преобразовать байт в формат D	6C
CVTBF	Преобразовать байт в формат F	4C
CVTBG	Преобразовать байт в формат G	4CFD
CVTBH	Преобразовать байт в формат H	6CFD
CVTBL	Преобразовать байт в длинное слово	98
CVTBW	Преобразовать байт в слово	99
CVTDB	Преобразовать формат D в байт	68
CVTDF	Преобразовать формат D в формат F	76
CVTDH	Преобразовать формат D в формат H	32FD
CVTDL	Преобразовать формат D в длинное слово	6A
CVTDW	Преобразовать формат D в слово	69

Мнемоника	Инструкция	Код операции
CVTFB	Преобразовать формат F в байт	48
CVTF	Преобразовать формат F в формат D	56
CVTFG	Преобразовать формат F в формат G	99FD
CVTFH	Преобразовать формат F в формат H	98FD
CVTFL	Преобразовать формат F в длинное слово	4A
CVTFW	Преобразовать формат F в слово	49
CVTGB	Преобразовать формат G в байт	48FD
CVTGF	Преобразовать формат G в формат F	33FD
CVTGH	Преобразовать формат G в формат H	56FD
CVTGL	Преобразовать формат G в длинное слово	4AFD
CVTGW	Преобразовать формат G в слово	49FD
CVTHB	Преобразовать формат H в байт	68FD
CVTHD	Преобразовать формат H в формат D	F7FD
CVTHF	Преобразовать формат H в формат F	F6FD
CVTHG	Преобразовать формат H в формат G	76FD
CVTHL	Преобразовать формат H в длинное слово	6AFD
CVTHW	Преобразовать формат H в слово	69FD
CVTLB	Преобразовать длинное слово в байт	F6
CVTLD	Преобразовать длинное слово в формат D	6E
CVTLF	Преобразовать длинное слово в формат F	4E
CVTLG	Преобразовать длинное слово в формат G	4EFD
CVTLH	Преобразовать длинное слово в формат H	6EFD
CVTLP	Преобразовать длинное слово в упакованное число	F9
CVTLW	Преобразовать длинное слово в слово	F7
CVTPL	Преобразовать упакованное число в длинное слово	36
CVTTP	Преобразовать оставшуюся числовую строку в упакованное число	26
CVTPT	Преобразовать упакованное число в остаточную числовую строку	24
CVTPS	Преобразовать упакованное число в лидирующую числовую строку	08
CVTRDL	Преобразовать округленное число в формате D в длинное слово	6B
CVTRFL	Преобразовать округленное число в формате F в длинное слово	4B
CVTRGL	Преобразовать округленное число в формате G в длинное слово	4BFD
CVTRHL	Преобразовать округленное число в формате H в длинное слово	6BFD
CVTSP	Преобразовать лидирующую числовую строку в упакованное число	09
CVTWB	Преобразовать слово в байт	33
CVTWD	Преобразовать слово в формат D	6D
CVTWF	Преобразовать слово в формат F	4D
CVTWG	Преобразовать слово в формат G	4DFD
CVTWH	Преобразовать слово в формат H	6DFD
CVTWL	Преобразовать слово в длинное слово	32
DECB	Уменьшить байт	97
DECL	Уменьшить длинное слово	D7

Мнемоника	Инструкция	Код операции
DECW	Уменьшить слово	B7
DIVB2	Разделить на байт (двухоперандная)	86
DIVB3	Разделить на байт (трехоперандная)	87
DIVD2	Разделить на число в формате D (двухоперандная)	66
DIVD3	Разделить числа в формате D (трехоперандная)	67
DIVF2	Разделить числа в формате F (двухоперандная)	46
DIVF3	Разделить числа в формате F (трехоперандная)	47
DIVG2	Разделить числа в формате G (двухоперандная)	46FD
DIVC3	Разделить числа в формате G (трехоперандная)	47FD
DIVH2	Разделить числа в формате H (двухоперандная)	66FD
DIVH3	Разделить числа в формате H (трехоперандная)	67FD
DIVL2	Разделить длинные слова (двухоперандная)	C6
DIVL3	Разделить длинные слова (трехоперандная)	C7
DIVP	Разделить упакованные числа	27
DIVW2	Разделить слова (двухоперандная)	A6
DIVW3	Разделить слова (трехоперандная)	A7
EDITPC	Отредактировать упакованное число в символьную строку	38
EDIV	Разделить числа с расширенной точностью	7B
EMODD	Умножить числа с расширенной точностью и выделить целую часть чисел в формате D	74
EMODF	Умножить числа с расширенной точностью и выделить целую часть в формате F	54
EMODG	Умножить числа с расширенной точностью и выделить целую часть чисел в формате G	54FD
EMODH	Умножить числа с расширенной точностью и выделить целую часть чисел в формате H	74FD
EMUL	Умножить числа с расширенной точностью	7A
EXTV	Извлечь битовое поле	EE
EXTZV	Извлечь расширенное битовое поле	EF
FFC	Найти первый очищенный бит	EB
FFS	Найти первый установленный бит	EA
HALT	Остановить процессор	00
INCB	Увеличить байт	96
INCL	Увеличить длинное слово	D6
INCW	Увеличить слово	B6
INDEX	Вычислить индекс	0A
INSQHI	Вставить в начало очереди (с блокировкой)	5C
INSQTI	Вставить в конец очереди (с блокировкой)	5D
INSQUE	Вставить в очередь	0E
INSV	Вставить поле	F0
JMP	Перейти	17
JSB	Перейти к подпрограмме	16
LDPCTX	Загрузить контекст процессора	06
LOCC	Загрузить символ	3A
MATCHC	Найти подстроку в символьной строке	39
MCOMB	Перенести дополненный байт	92
MCOML	Перенести дополненное длинное слово	D2
MCOMW	Перенести дополненное слово	B2
MFPR	Перенести из привилегированного регистра	DB

Мнемоника	Инструкция	Код операции
MNEGB	Перенести байт, взятый с обратным знаком	8E
MNEGD	Перенести число в формате D, взятое с обратным знаком	72
MNEGF	Перенести число в формате F, взятое с обратным знаком	52
MNEGG	Перенести число в формате G, взятое с обратным знаком	52FD
MNEGH	Перенести число в формате H, взятое с обратным знаком	72FD
MNEGL	Перенести длинное слово, взятое с обратным знаком	CE
MNEGW	Перенести слово, взятое с обратным знаком	AE
MOVAB	Перенести адрес байта	9E
MOVAD	Перенести адрес числа в формате D	7E
MOVAF	Перенести адрес числа в формате F	DE
MOVAG	Перенести адрес числа в формате G	7E
MOVAH	Перенести адрес числа в формате H	7EFD
MOVAL	Перенести адрес длинного слова	DE
MOVAO	Перенести адрес восьмикратного слова	7EFD
MOVAQ	Перенести адрес четырехкратного слова	7E
MOVAW	Перенести адрес слова	3E
MOVB	Перенести байт	90
MOVC3	Перенести символьную строку (трехоперандная)	28
MOVC5	Перенести символьную строку (пятиоперандная)	2C
MOVD	Перенести число в формате D	70
MOVF	Перенести число в формате F	50
MOVG	Перенести число в формате G	50FD
MOVH	Перенести число в формате H	70FD
MOVL	Перенести длинное слово	D0
MOVO	Перенести восьмикратное слово	7DFD
MOVP	Перенести упакованное число	34
MOVPSL	Перенести длинное слово состояния процессора	DC
MOVQ	Перенести четырехкратное слово	7D
MOVTC	Перенести оттранслированную символьную строку	2E
MOVTUC	Перенести оттранслированную символьную строку, обработав последовательность расширения кода	2F
MOVW	Перенести слово	BO
MOVZBL	Перенести целый байт без знака в длинное слово	9A
MOVZBW	Перенести целый байт без знака в слово	9B
MOVZWL	Перенести слово целого числа без знака в длинное слово	3C
MTPR	Перенести в привилегированный регистр	DA
MULB2	Умножить байты (двухоперандная)	84
MULB3	Умножить байты (трехоперандная)	85
MULD2	Умножить числа в формате D (двухоперандная)	64
MULD3	Умножить числа в формате D (трехоперандная)	65
MULF2	Умножить числа в формате F (двухоперандная)	44
MULF2	Умножить числа в формате F (двухоперандная)	44
MULF3	Умножить числа в формате F (трехоперандная)	45

Мнемоника	Инструкция	Код операции
MULG2	Умножить числа в формате G (двухоперандная)	44FD
MULG3	Умножить числа в формате G (трехоперандная)	45FD
MULH2	Умножить числа в формате H (двухоперандная)	64FD
MULH3	Умножить числа в формате H (трехоперандная)	65FD
MULL2	Умножить длинные слова (двухоперандная)	C4
MULL3	Умножить длинные слова (трехоперандная)	C5
MULP	Умножить упакованные числа	25
MULW2	Умножить слова (двухоперандная)	A4
MULW3	Умножить слова (трехоперандная)	A5
NOP	Нет операции	01
POLYD	Вычислить полином в формате D	75
POLYF	Вычислить полином в формате F	55
POLYG	Вычислить полином в формате G	55FD
POLYH	Вычислить полином в формате H	75FD
POPR	Вытолкнуть регистры из стека	BA
PROBER	Опробовать доступ для чтения	0C
PROBEW	Опробовать доступ для записи	0D
PUSHAB	Протолкнуть в стек адрес байта	9F
PUSHAD	Протолкнуть в стек адрес числа в формате D	7F
PUSHAF	Протолкнуть в стек адрес числа в формате F	DF
PUSHAG	Протолкнуть в стек адрес числа в формате G	7F
PUSHAH	Протолкнуть в стек адрес числа в формате H	7FFD
PUSHAL	Протолкнуть в стек адрес длинного слова	DF
PUSHAO	Протолкнуть в стек адрес восьмикратного слова	7FFD
PUSHAQ	Протолкнуть в стек адрес четырехкратного слова	7F
PUSHAW	Протолкнуть в стек адрес слова	3F
PUSHL	Протолкнуть в стек длинное слово	DD
PUSHR	Протолкнуть в стек регистры	BB
REI	Вернуться из прерывания или исключения	02
REMQHI	Удалить из начала очереди (с блокировкой)	5E
REMQTI	Удалить из конца очереди (с блокировкой)	5F
REMQUE	Удалить из очереди	0F
RET	Вернуться из процедуры	04
ROTL	Циклически сдвинуть длинное слово	9C
RSB	Вернуться из подпрограммы	05
SBWC	Вычесть с учетом переноса	D9
SCANC	Найти символы	2A
SKPC	Пропустить символы	3B
SOBGEQ	Вычесть единицу и перейти, если больше или равно	F4
SOBGTR	Вычесть единицу и перейти, если больше	F5
SPANC	Пропустить символы	2B
SUBB2	Вычесть байты (двухоперандная)	82
SUBB3	Вычесть байты (трехоперандная)	83
SUBD2	Вычесть числа в формате D (двухоперандная)	62
SUBD3	Вычесть числа в формате D (трехоперандная)	63
SUBF2	Вычесть числа в формате F (двухоперандная)	42
SUBF3	Вычесть числа в формате F (трехоперандная)	43
SUBC2	Вычесть числа в формате G (двухоперандная)	42FD
SUBC3	Вычесть числа в формате G (трехоперандная)	43FD

Мнемоника	Инструкция	Код операции
SUBH2	Вычесть числа в формате H (двухоперандная)	62FD
SUBH3	Вычесть числа в формате H (трехоперандная)	63FD
SUBL2	Вычесть длинные слова (двухоперандная)	C2
SUBL3	Вычесть длинные слова (трехоперандная)	C3
SUBP4	Вычесть упакованные слова (четырехоперандная)	22
SUBP6	Вычесть упакованные слова (шестиоперандная)	23
SUBW2	Вычесть слова (двухоперандная)	A2
SUB3	Вычесть слова (трехоперандная)	A3
SVPCTX	Сохранить контекст процесса	07
TSTB	Проверить байт	95
TSTD	Проверить число в формате D	73
TSTF	Проверить число в формате F	53
TSTG	Проверить число в формате G	53FD
TSTH	Проверить число в формате H	73FD
TSTL	Проверить длинное слово	D5
TSTW	Проверить слово	B5
XFC	Вызов дополнительной функции	FC
XORB2	ИСКЛЮЧАЮЩЕЕ ИЛИ байт (двухоперандная)	8C
XORB3	ИСКЛЮЧАЮЩЕЕ ИЛИ байт (трехоперандная)	8D
XORL2	ИСКЛЮЧАЮЩЕЕ ИЛИ длинных слов (двухоперандная)	CC
XORL3	Исключающее ИЛИ длинных слов (трехоперандная)	CD
XORW2	Исключающее ИЛИ слов (двухоперандная)	TC
XORW3	Исключающее ИЛИ слов (трехоперандная)	AD
ESCD	Зарезервирована фирмой DEC	FD
ESCE	Зарезервирована фирмой DEC	FE
ESCF	Зарезервирована фирмой DEC	FF

Зарезервированы фирмой DEC следующие коды операций:
57, 59, 5A, 5B, 77, 00FD – 31FD, 34FD – 3FFD, 57FD, 58FD – 5FFD, 77ED, 78FD – 7FFD, 80FD – 97FD, 9AFD – F5FD, F8FD – FCFF.

ПРИЛОЖЕНИЕ Ж

СИМВОЛИКА БЛОК-СХЕМ ПРОГРАММ

Эта символика утверждена Американским национальным институтом стандартов в 1970 г.

СПИСОК ЛИТЕРАТУРЫ

1. Digital Equipment Corporation Handbooks: *PDP-11 Software Handbook*, 1982–83. *PDP-11 Architecture Handbook*, 1983–84. *Microcomputers and Memories*, 1982. *PDP-11 Peripherals Handbook*, 1976.
2. Eckhouse, Richard H., Jr. *Minicomputer Systems Organization and Programming (PDP-11)*. Englewood Cliffs, N.J.: Prentice-Hall, 1975.
3. Eckhouse, Richard H., Jr., and L. Robert Morris. *Minicomputer Systems Organization, Programming and Applications (PDP-11)*. Englewood Cliffs, N.J.: Prentice-Hall, 1979.
4. Frank, Thomas S. *Introduction to the PDP-11 and its Assembly Language*. Englewood Cliffs, N.J.: Prentice-Hall, 1983.
5. Gill, Arthur. *Machine and Assembly Language Programming of the PDP-11*, 2d ed. Englewood Cliffs, N.J.: Prentice-Hall, 1983.
6. Hamacher, V. Carl, Zuonko G. Vranesic, and Safwat G. Zaky. *Computer Organization*. New York: McGraw-Hill, 1978.
7. Kapps, Charles A., and Robert L. Stafford. *Assembly Language for the PDP-11*. Prindle, Weber and Schmidt and CBI, 1981.
8. Lemone, Karen A., and Martin E. Kaliski. *Assembly Language Programming for the VAX-11*. Boston: Little, Brown, 1983.
9. Levy, Henry M., and Richard H. Eckhouse, Jr. *Computer Programming and Architecture: The Vax-11*. Bedford, Mass.: Digital Press, 1980.
10. Lewis, Harry R. *An Introduction to Computer Programming and Data Structures Using MACRO-11*. Reston, Va.: Reston Publishing, 1981.
11. MacEwen, Glenn H. *Introduction to Computer Systems Using the PDP-11 and Pascal*. New York: McGraw-Hill, 1980.
12. McNamara, John E. *Technical Aspects of Data Communication*. Digital Equipment Corporation, 1977.
13. Singer, Michael. *PDP-11 Assembler Language Programming and Machine Organization*. New York: Wiley, 1980.
14. Sloan, M. E. *Introduction to Minicomputers and Microcomputers*. Reading, Mass.: Addison-Wesley, 1980.
15. Sloan, M. E. *An Introduction: Computer Hardware and Organization*, 2d ed. Science Research Associates Inc., 1983.
16. Stone, Harold S., and Siewiorek. *Introduction to Organization and Data Structure: PDP-11 Edition*. New York: McGraw-Hill, 1975.

ОГЛАВЛЕНИЕ

Лин В.

Л59 PDP-11 и VAX-11. Архитектура ЭВМ и программирование на языке ассемблера: Пер. с англ. – М.: Радио и связь, 1989. – 320 с.: ил.

ISBN 5-256-00299-6

В книге американского автора рассмотрены архитектура аппаратного обеспечения ЭВМ семейства PDP-11. Показано взаимодействие аппаратного и программного обеспечения и пользования как единой системы, предназначенной для решения определенной задачи. Приведено большое число наглядных примеров и полезных приложений. В одной из глав, являющейся введением в программирование для ЭВМ VAX-11, кратко описаны структура машины, типы данных, команды, режимы адресации, макросы, подпрограммы и прерывания.

Для широкого круга программистов.

Л $\frac{2404000000\text{-}094}{046(01)\text{-}89}$ 137-89

ББК 32.973

Производственное издание

Лин Вэн

PDP-11 И DAX-11. АРХИТЕКТУРА ЭВМ И ПРОГРАММИРОВАНИЕ НА ЯЗЫКЕ АССЕМБЛЕРА

Заведующая редакцией О. В. Толкачева
Редактор М. Г. Коробочкина
Перплет художника И. В. Тыртычного
Художественный редактор А. С. Широков
Технический редактор А. Н. Золотарева
Корректор Н. В. Козлова

ИБ № 1570

Подписано в печать с оригинала-макета 15.03.89. Формат 70x100/16. Бумага офсетная № 2
Гарнитура "Пресс-роман". Печать офсетная. Усл. печ. л. 26,0. Усл. кр.-отт. 26,0. Уч.-изд. л. 24,83.
Тираж 30 000 экз. Изд. № 22132. Заказ № 2023. Цена 2 р. 10 к.
Издательство "Радио и связь". 101000 Москва, Почтамт, а/я 693

Московская типография № 4 Союзполиграфпрома при Государственном комитете СССР по делам издательств, полиграфии и книжной торговли. 129041 Москва, Б. Переяславская ул., д. 46

УВАЖАЕМЫЙ ЧИТАТЕЛЬ!

В 1988 г. в издательстве "Радио и связь" выйдут книги:

Крутько П. Д., Максимов А. И., Скворцов Л. М. **Алгоритмы и программы проектирования автоматических систем**/Под ред. П. Д. Крутько. – 21 л.: ил. – 1 р. 40 к.

Изложены методики разработки алгоритмов исследования и проектирования линейных и нелинейных систем автоматического управления. Рассмотрены задачи, наиболее часто встречающиеся в инженерной практике: от построения математических моделей систем до расчета параметров управляющих алгоритмов. Представленные алгоритмы и программы на языке Фортран IV предназначены для непосредственного применения пользователем. Приведены пакеты прикладных программ, в которых реализованы эффективные методы анализа и синтеза линейных и нелинейных систем.

Для инженерно-технических работников, занимающихся проектированием и исследованием систем управления..

Черемных С. В., Гиглавый А. В., Поляк Ю. Е. **От микропроцессоров к персональным ЭВМ.** – 21 л.: ил. – 1 р. 40 к.

Рассматриваются микропроцессоры, персональные ЭВМ и их роль в организации инженерного труда. Даются примеры применения программируемых микрокалькуляторов и персональных ЭВМ для решения инженерных задач – от расчетов по формулам до принятия решений, а также для построения графических зависимостей и выпуска отчетов. Приводятся характеристики отечественных и зарубежных профессиональных персональных ЭВМ.

Для инженеров, использующих автоматизированные рабочие места на базе микроЭВМ или занимающихся их разработкой.

В 1989 г. в издательстве "Радио и связь" выйдут книги:

М и р е н к о в Н. Н. **Параллельное программирование для многомодульных вычислительных систем.** – 21 л.: ил. – ISBN 5-256-00196-5 – 1 р. 90 к.

Рассматриваются математические методы и программные средства параллельной обработки данных в многомодульных системах с динамически реконфигурируемой структурой. Базовый модуль таких систем, осуществляющий хранение, переработку и эффективную транспортировку данных, ориентирован на реализацию в виде СБИС. С единых позиций описываются способы построения параллельных алгоритмов, конкретные методы параллельных вычислений, языки, средства анализа, отладки и специальной подготовки параллельных программ. Анализируются операционные системы, алгоритмы планирования и управления, программное обеспечение системы МИНИМАКС.

Для научных работников – специалистов в области вычислительных систем.

А р х а н г е л ь с к и й Б. В., Ч е р н я х о в с к и й В. В. **Поиск устойчивых ошибок в программах.** – 18 л.: ил. – ISBN 5-256-00306-2 – 1 р. 20 к.

Систематизированы устойчивые семантические ошибки программирования, характерные для различных классов задач, разных языков программирования и программистов различной квалификации. Рассматривается новый метод отладки программ с помощью поиска устойчивых ошибок, позволяющий по сравнению с традиционными методами обнаруживать большинство типичных ошибок программирования с меньшими затратами ресурсов. Описывается его реализация в системе отладки КАПКАН-ФОРТРАН. Определяется область применения метода устойчивых ошибок и исследуется его взаимосвязь с другими методами отладки.

Для программистов всех категорий и инженерно-технических работников, связанных с программированием.

Эти книги Вы можете приобрести во всех книжных магазинах, распространяющих научно-техническую литературу.