А. П. Жмакин

АРХИТЕКТУРА ЭВМ

- *Функциональная организация ЭВМ*
- *Машинная арифметика и синтез устройств*
- *Архитектура микропроцессорных систем*
- *Программная модель учебной ЭВМ*
- *Лабораторный практикум*
- *Курсовое проектирование*

+ CD

УЧЕБНОЕ ПОСОБИЕ

bhv®

А. П. Жмакин

АРХИТЕКТУРА ЭВМ

Рекомендовано УМО по образованию в области инновационных междисциплинарных образовательных программ в качестве учебного пособия по специальности «Математическое обеспечение и администрирование информационных систем»- 010503

Санкт-Петербург
«БХВ-Петербург»
2006

УДК 681.3(075.8)
ББК 32.973-02я73
Ж77

Жмакин А. П.

Ж77 Архитектура ЭВМ. — СПб.: БХВ-Петербург, 2006. — 320 с: ил.

ISBN 5-94157-719-2

Пособие объединяет в одном издании теоретическую часть одноименной дисциплины и лабораторный практикум. Рассмотрены базовые вопросы организации ЭВМ: функциональная организация ЭВМ, системы команд и командный цикл. Большое внимание уделено арифметическим основам ЭВМ, принципам построения различных усройств и их взаимодействию. Обсуждаются вопросы построения микропроцессорных систем. Лабораторный практикум проводится на программной модели ЭВМ, представленной на прилагаемом компакт-диске. Также пособие содержит материалы для выполнения курсового проектирования.

Для студентов и преподавателей технических вузов

УДК 681.3(075.8)
ББК 32.973-02я73

Группа подготовки издания:

Главный редактор	*Екатерина Кондукова*
Зам. главного редактора	*Людмила Еремеевская*
Зав. редакцией	*Григорий Добин*
Редактор	*Анна Кузьмина*
Компьютерная верстка	*Ольги Сергиенко*
Корректор	*Зинаида Дмитриева*
Дизайн серии	*Игоря Цырульникова*
Оформление обложки	*Елены Беляевой*
Зав. производством	*Николай Тверских*

Рецензенты:
Терехов А. Н., д.ф.-м. н., профессор,
заведующий кафедрой системного программирования
Санкт-Петербургского государственного университета
Костин В. А., к. ф.-м. н., доцент кафедры информатики
Санкт-Петербургского государственного университета

Лицензия ИД № 02429 от 24.07.00. Подписано в печать 12.12.05.
Формат 70x100 1/16. Печать офсетная. Уел. печ. л. 25,8.
Тираж 3000 экз. Заказ № 4499
"БХВ-Петербург", 194354, Санкт-Петербург, ул. Есенина, 5Б.

Санитарно-эпидемиологическое заключение на продукцию
№ 77.99.02.953.Д.006421.11.04 от 11.11.2004 г. выдано Федеральной службой
по надзору в сфере защиты прав потребителей и благополучия человека.

Отпечатано с готовых диапозитивов
в ГУП "Типография "Наука"
199034, Санкт-Петербург, 9 линия. 12

ISBN 5-94157-719-2

Оглавление

Предисловие

Эта книга создавалась как учебное пособие по архитектуре процессоров и ЭВМ. Материал книги ориентирован на студентов инженерных специальностей, обучающихся в области разработки программного обеспечения и информационных систем, для которых "компьютерное железо" не является основным предметом изучения, но которые хотят (и должны) знать основы построения процессоров, организацию взаимодействия основных устройств ЭВМ, программирование на низком уровне. Книга может быть полезной и студентам педагогических специальностей, в учебных планах которых предусмотрены курсы по изучению архитектуры ЭВМ (физика, математика, информатика).

Читателям необходимо владеть начальными знаниями в области цифровой схемотехники (булева алгебра, логические элементы, триггеры, операционные элементы).

В основу книги положены материалы курсов лекций, читаемых автором на кафедре вычислительной техники Курского государственного технологического университета (КГТУ) и кафедре программного обеспечения и администрирования информационных систем КГУ. Книга объединяет в себе теоретический материал, цикл лабораторных работ и задания на курсовое проектирование.

Пособие состоит из трех частей.

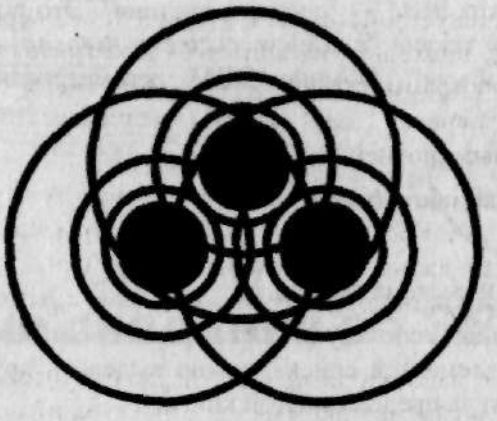
В *части I* рассматриваются общие принципы организации ЭВМ, включая их функциональную и структурную организацию. Достаточно подробно рассмотрены арифметические основы ЭВМ, представление чисел в различных кодах и алгоритмы выполнения арифметических операций. Уделено внимание кодированию десятичных чисел и десятичной машинной арифметике, а также системам счисления в остаточных классах. Далее рассматриваются

принципы построения устройств ЭВМ — концепция операционного и управляющего автоматов, подходы к их синтезу. Рассмотрены управляющие автоматы с жесткой и программируемой логикой. Отдельная глава посвящена организации многоуровневой памяти ЭВМ, вопросам взаимодействия устройств памяти разных уровней.

Часть II посвящена обсуждению базовой архитектуры систем на основе микропроцессоров *x*86. Кроме внутренней структуры микропроцессора и его интерфейса рассматривается организация ввода/вывода, прерываний, прямого доступа в память. Кратко рассмотрена эволюция архитектуры процессоров семейства *x*86.

Часть III включает лабораторный практикум и курсовое проектирование. В лабораторном практикуме описывается структура и система команд разработанной под руководством автора программной модели учебной ЭВМ (прилагается компакт-диск с моделью) и предлагается к выполнению ряд работ с этой моделью. Для каждой работы сформулирована цель, индивидуальные задания, требования к оформлению отчета и контрольные вопросы; для некоторых заданий приведены примеры выполнения. Содержанием курсовой работы является разработка арифметико-логического устройства, реализующего заданный набор операций с учетом ограничений на код выполнения операций и способ построения управляющего автомата. Сформулированы индивидуальные задания, определены этапы выполнения работы и содержание пояснительной записки.

Автор выражает искреннюю признательность заведующему кафедрой системного программирования СПбГУ профессору, докт. физ.-мат. наук А. Н. Терехову и доценту, канд. физ.-мат. наук В. А. Костину за ценные замечания, сделанные при рецензировании книги.



ЧАСТЬ I

Принципы организации ЭВМ

Принято считать, что ЭВМ — "сложная система". Это понятие имеет много трактовок, в т. ч. и такую: *"сложную систему невозможно адекватно описать на одном языке"*. Обычно ЭВМ рассматривают на нескольких уровнях:

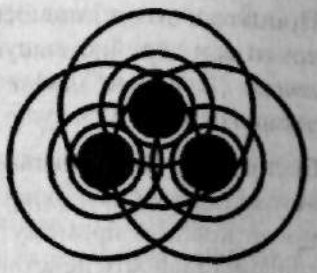
- логические элементы;
- операционные элементы (узлы);
- устройства;
- структура ЭВМ и система команд.

На каждом из уровней используются свои языки описания. И "выше", и "ниже" приведенных элементов списка можно выделить другие уровни, но их рассмотрение лежит за пределами этой книги.

Приступая к изучению вопросов архитектуры ЭВМ, читатель должен иметь представление о логических и операционных элементах цифровой техники (конъюнкторы, инверторы, ..., триггеры, регистры, мультиплексоры, дешифраторы, сумматоры и т. д.).

Центральным в структуре ЭВМ является, несомненно, процессор, а главными устройствами любого процессора можно считать *арифметико-логическое устройство* (АЛУ) и *устройство управления* (УУ). Далее мы подробно рассмотрим принципы и способы организации АЛУ. Поскольку АЛУ разрабатывается для реализации определенных алгоритмов арифметической и логической обработки данных, то неизбежным становится и рассмотрение различных вариантов таких алгоритмов.

ГЛАВА 1



Начальные сведения об ЭВМ

1.1. История развития вычислительной техники

С тех пор, как человечество осознало понятие количества, разрабатывались и применялись различные приспособления для отображения количественных эквивалентов и операций над величинами. Отбросив рассмотрение "доисторических" с точки зрения вычислительной техники средств (кучки камней, счеты и т. д.), рассмотрим кратко историю развития вычислительных машин.

Пожалуй, первой реально созданной машиной для выполнения арифметических действий в десятичной системе счисления можно считать *счетную машину Паскаля*. В 1642 г. Б. Паскаль продемонстрировал ее работу. Машина выполняла суммирование чисел (восьмиразрядных) с помощью колес, которые при добавлении единицы поворачивались на 36° и приводили в движение следующее по старшинству колесо всякий раз, когда цифра 9 должна была перейти в значение 10. Машина Паскаля получила известность во многих странах, было изготовлено более 50 экземпляров машины.

Впрочем, еще до Паскаля машину, механически выполняющую арифметические операции, изобразил в эскизах Леонардо да Винчи (1452—1519). Суммирующая машина по его эскизам выполнена в наши дни и доказала свою работоспособность.

В средние века (расцвет механики) было предложено и выполнено много различных вариантов арифметических машин: Морлэнд (1625—1695), К. Перро (1613—1688), Якобсон, Чебышев и др. Первую машину, с помощью которой можно было не только складывать, но и умножать и делить, разработал Г. Лейбниц (1646—1716). Однако большинство подобных машин изготавливались авторами в единичных экземплярах. Удачное решение инженера

В. Однера, разработавшего колесо с переменным числом зубьев, позволило почти век серийно выпускать арифмометры (например, "Феликс" Курского завода *"Счетмаш"*), *являвшиеся основным средством вычислений вплоть до* эпохи ПЭВМ и калькуляторов.

Все упомянутые выше механизмы обладали одной особенностью — могли автоматически выполнять только отдельные действия над числами, но не могли хранить промежуточные результаты и, следовательно, выполнять последовательность действий.

Первой вычислительной машиной, реализующей автоматическое выполнение последовательности действий, можно считать *разностную машину Ч. Беббеджа* (1792—1871). В 1819 г. он изготовил ее для расчета астрономических и морских таблиц. Машина обеспечивала хранение необходимых промежуточных значений и выполнение последовательности сложений для получения значения функции. В дальнейшем Беббедж предложил т. н. *аналитическую машину*, предназначенную для решения любых вычислительных задач. При желании в аналитической машине Беббеджа можно найти прообразы всех основных устройств современной ЭВМ: арифметическое устройство ("мельница"), память ("склад"), устройство управления (на перфокартах), позволяющее выбирать различные пути решения в зависимости от значений исходных данных и промежуточных результатов. Проект аналитической машины Беббеджа так и не был реализован — из-за несоответствия идеи и элементной базы.

Даже выпускаемые большими сериями электрические *релейные машины Холлерита* (1860—1929) — *табуляторы* — не произвели переворота в средствах обработки информации, хотя и широко использовались для обработки статистической информации вплоть до 70-х годов прошлого века.

Идеи аналитической машины Беббеджа были использованы в релейных машинах, выпускавшихся в 30—40-х годах XX века. Теоретической основой разработки релейно-контактных схем явился аппарат булевой алгебры, который в дальнейшем использовался для синтеза схем ЭЭВМ. Однако и электрические реле как элементная база вычислительной техники не удовлетворяли потребностям этой техники по всем основным параметрам (быстродействие, надежность, потребляемая мощность, стоимость, габариты и др.).

Только освоение электронных схем в качестве элементной базы положило начало действительно массовому внедрению сначала вычислительной, а потом и информационной техники во все сферы человеческой деятельности. Первые электронные цифровые вычислительные машины (ЭЭВМ) были разработаны и выпущены на рубеже 40—50-х годов прошлого века в США, Англии и чуть позднее — в СССР.

1.2. Цифровые и аналоговые вычислительные машины

Все приведенные выше факты относятся к истории т. н. *цифровой* вычислительной техники, в которой информация представлена в дискретной форме (в форме чисел, кодов, знаков). Однако большинство физических величин может принимать значение из непрерывного множества — континуума. Существуют вычислительные устройства, оперирующие непрерывной информацией (пример — логарифмическая линейка, где информация представлена отрезками длины). Существует и целый класс электронных вычислительных машин — т. н. *аналоговые*, информация в которых представляется непрерывными значениями электрического напряжения или тока. Принцип работы таких машин — в построении электрических цепей, процессы в которых описываются теми дифференциальными уравнениями, которые требуется решить.

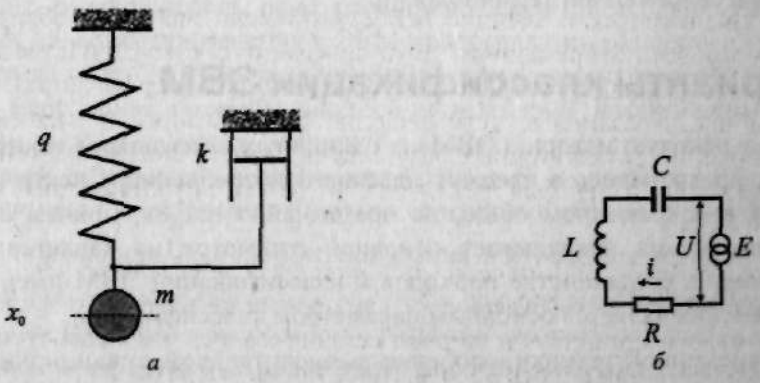


Рис. 1.1. Модель: *а* — механическая система; *б* — "аналогичная" ей электрическая цепь

Классический пример такого подхода показан на рис. 1.1. Например, требуется изучить поведение механической колебательной системы, описываемой дифференциальным уравнением (1.1). Подберем электрическую цепь, процессы в которой описываются тем же дифференциальным уравнением с точностью до обозначений (1.2). Между механическими величинами (рис. 1.1, *а*) и электрическими (рис. 1.1, *б*) существует соответствие (сравните уравнения (1.1) и (1.2)).

$$m\frac{dx}{dt} = mg - \int_0^t x \cdot dt - kx. \qquad (1.1)$$

$$L\frac{di}{dt} = U - \int_0^t i \cdot dt - Ri. \qquad (1.2)$$

Таким образом, для механического устройства можно подобрать электрическую цепь, процессы в которой описываются аналогичными дифференциальными уравнениями. Или, для заданного дифференциального уравнения (системы) построить электрическую цепь, которая описывается этим уравнением.

Существует хорошо отработанная методика синтеза таких цепей и наборы функциональных блоков (АВМ), позволяющие собирать и исследовать синтезированные цепи.

Достоинства АВМ: простота подготовки решения, высокая скорость решения.

Недостатки АВМ: неуниверсальность (предназначены только для решения дифференциальных уравнений) и низкая точность решения.

В настоящее время АВМ находят применение лишь в ограниченных областях технического моделирования. Поэтому в дальнейшем будем употреблять термин "ЭВМ", имея в виду только цифровые вычислительные машины, как это принято в современной терминологии.

1.3. Варианты классификации ЭВМ

За свою полувековую историю ЭВМ из единичных экземпляров инструментов ученых превратились в предмет массового потребления. Спектр применения ЭВМ в современном обществе чрезвычайно широк, причем именно область применения накладывает основной отпечаток на характеристики ЭВМ. Поэтому в большинстве подходов к классификации ЭВМ именно область применения является основным параметром классификации.

Изделия современной техники, особенно вычислительной, традиционно принято делить на поколения (табл. 1.1), причем основным признаком поколения ЭВМ считается ее элементная база. Следует помнить, что любая классификация не является абсолютной. Всегда можно отыскать объект классификации, который по одним параметрам относится к одному классу, а по другим — к другому. Это в большой степени относится и к классификации поколений ЭВМ: некоторые авторы выделяют три поколения ЭВМ (дальнейшее развитие ЭВМ идет как бы вне поколений), другие насчитывают целых шесть.

В рамках *первого поколения* ЭВМ не возникала необходимость в классификации, т. к. машин были считанные единицы и использовались они, как правило, для выполнения научно-технических расчетов. Отдельные машины характеризовались быстродействием (числом выполняемых операций в секунду), объемом памяти, стоимостью, надежностью (наработка на отказ), габаритно-весовыми характеристиками, потребляемой мощностью и другими параметрами.

Таблица 1.1. Поколения ЭВМ

Поколение	Элементная база	Годы существования	Области применения
Первое	Электронные лампы	50—60	Научно-технические расчеты
Второе	Транзисторы, ферритовые сердечники	60—70	Научно-технические расчеты, планово-экономические расчеты
Третье	Интегральные схемы	70—80	Научно-технические расчеты, планово-экономические расчеты, системы управления
Четвертое	СИС, БИС, СБИС и т. д.	80 и по сей день	Все сферы деятельности

Использование транзисторов в качестве элементной базы *второго поколения* привело к улучшению примерно на порядок каждого из основных параметров ЭВМ. Это, в свою очередь, резко расширило сферу применения ЭВМ, причем в разных областях применения к ЭВМ предъявлялись различные требования. Так называемые "научно-технические расчеты" характеризовались относительно небольшим объемом входной и выходной информации, но очень большим числом сложных операций с высокой точностью над входной информацией, а "планово-экономические расчеты"[1] — наоборот, простейшими операциями (сложение, сравнение) над огромными объемами информации.

Соответственно в рамках второго поколения ЭВМ выделялись:

- ЭВМ для *научно-технических расчетов*, характеризующиеся мощным быстродействующим процессором с развитой системой команд (в т. ч. реализующей арифметику с плавающей запятой) и относительно небольшой внешней памятью и номенклатурой устройств ввода/вывода;
- ЭВМ для *планово-экономических расчетов*, характеризующиеся, прежде всего, большой многоуровневой памятью, развитой номенклатурой устройств ввода/вывода (УВВ), но относительно простым и дешевым процессором, система команд которого включает простые арифметические команды (сложение, вычитание) с фиксированной запятой.

Характерно, что и языки программирования "второго поколения" так же разделялись на "математические" (FORTRAN) и "экономические" (COBOL).

Однако по мере расширения сферы применения ЭВМ, улучшения их основных характеристик, появления новых задач, границы между выделенными классами стали размываться. Уже в рамках второго поколения стали выде-

[1] Здесь используется терминология, принятая в годы существования второго поколения ЭВМ.

лять т. н. ЭВМ *общего назначения*, одинаково хорошо приспособленные для решения разнообразных задач. Такие машины объединяли в себе достоинства "научно-технических" и "планово-экономических" ЭВМ: мощный процессор, большую память, широкую номенклатуру УВВ (в то время это уже можно было себе позволить). Такие машины могли решать задачи, недоступные предыдущим моделям. Но для решения более простых задач их ресурсы являлись избыточными и, следовательно, решение этих задач — экономически не оправдано. Поэтому ЭВМ общего назначения (универсальные ЭВМ) стали выпускать различной вычислительной мощности (и, следовательно, стоимости): *большие*, *средние* и *малые*.

В рамках ЭВМ *третьего поколения* стал усиленно развиваться новый класс — *управляющие ЭВМ*. К ЭВМ, работающим в контуре управления объектом или технологическим процессом, предъявляются специфические требования: прежде всего, высокая надежность, способность работать в экстремальных внешних условиях (перепады температуры, давления, питающих напряжений, высокий уровень электромагнитных помех и т. п.), быстрая реакция на изменения состояния внешней среды, малые габариты и вес, простота обслуживания. В то же время к таким характеристикам, как быстродействие процессора, мощность системы команд, объем памяти, часто не предъявлялись слишком высоких требований, зато решающим становился фактор стоимости. Эти особенности привели к появлению класса т. н. *мини-ЭВМ*, а затем и *микроЭВМ*, хотя в дальнейшем и мини- и микроЭВМ использовались не только в качестве управляющих. Иногда эти классы объединяли понятием *проблемно-ориентированные ЭВМ*.

Наряду с упомянутыми классами ЭВМ широкого применения всегда выпускались машины, которые можно было считать *специализированными*. Это, во-первых, т. н. *суперЭВМ*, выпускаемые в единичных экземплярах и предназначенные для решения задач, недоступных для серийной вычислительной техники. Для ряда применений создавались специализированные ЭВМ, архитектура и структура которых оптимизировалась под решение конкретной задачи. Ту же задачу можно было решить и на универсальной ЭВМ подходящего класса, но со значительно более низкими показателями качества. В то же время решение других задач на специализированной ЭВМ было либо невозможно, либо крайне неэффективно. Одна из возможных классификаций ЭВМ на рубеже 3—4 поколений показана на рис. 1.2.

Еще одним важным явлением, проявившимся при развитии третьего поколения ЭВМ, стало появление *семейств ЭВМ*. В рамках одного семейства, объединенного общими архитектурными, структурными, а иногда — и конструктивными решениями, выпускались несколько (иногда — более десятка) классов ЭВМ: малые, средние, "полусредние", большие, очень большие и т. д.

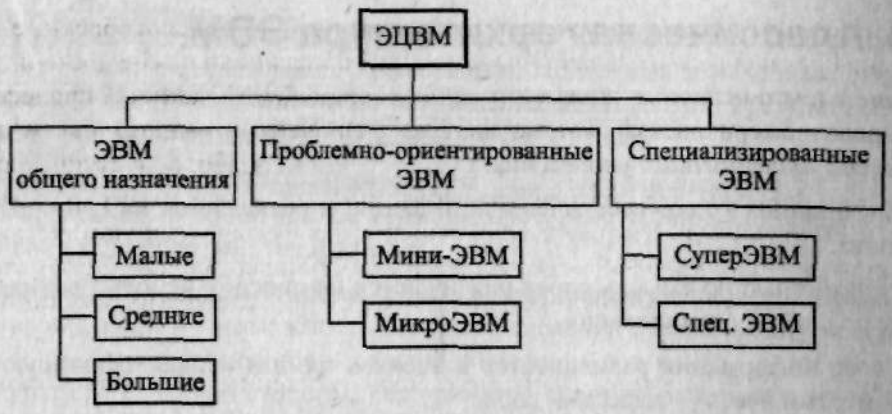


Рис. 1.2. Вариант классификации ЭВМ

Общими для большинства семейств являются:

- внутренний язык, что позволяет осуществлять совместимость программ на уровне машинных кодов (IBM-360, ЕС ЭВМ) либо системы команд, обладающие совместимостью "снизу вверх" (PDP-11), когда старшие представители семейства реализуют все команды младших моделей плюс еще некоторые команды;
- форматы данных;
- форматы записи на внешний носитель;
- интерфейс, что позволяет иметь единую номенклатуру внешних устройств для всех представителей семейства;
- преемственность программного обеспечения (как правило, та же совместимость "снизу вверх").

Для решения конкретной задачи пользователь подбирал соответствующий экземпляр семейства, а по мере усложнения задачи осуществлялся переход на более старшие модели семейства, причем уже отлаженные на младших моделях программы, как правило, не требовали доработки.

Наиболее известными примерами семейств ЭВМ могут служить:

- семейство универсальных ЭВМ третьего поколения IBM-360 и его советский аналог — ЕС ЭВМ, включающее малые машины ЕС-1010 и ЕС-1020, средние ЕС-1022, ЕС-1030, ЕС-1035 и др., большие ЕС-1050, ЕС-1060, ЕС-1065;
- семейство мини-ЭВМ PDP-11 и его советский аналог — СМ ЭВМ (лишь часть представителей семейства — СМ-3, СМ-4, СМ-1420);
- семейство микроЭВМ LXI-11 (Электроника-60 и ее модификации);
- семейство микропроцессоров i80x86.

1.4. Классическая архитектура ЭВМ

Считается, что основные идеи построения современных ЭВМ в 1945 г. сформулировал американский математик Дж. фон Нейман, определив их как *принципы программного управления*:

1. Информация кодируется в двоичной форме и разделяется на единицы — слова.
2. Разнотипные по смыслу слова различаются по способу использования, но не по способу кодирования.
3. Слова информации размещаются в ячейках памяти и идентифицируются номерами ячеек — адресами слов.
4. Алгоритм представляется в форме последовательности управляющих слов, называемых *командами*. Команда определяет наименование операции и слова информации, участвующие в ней. Алгоритм, записанный в виде последовательности команд, называется *программой*.
5. Выполнение вычислений, предписанных алгоритмом, сводится к последовательному выполнению команд в порядке, однозначно определенном программой.

Поэтому классическую архитектуру современных ЭВМ, представленную на рис. 1.3, часто называют "архитектурой фон Неймана".

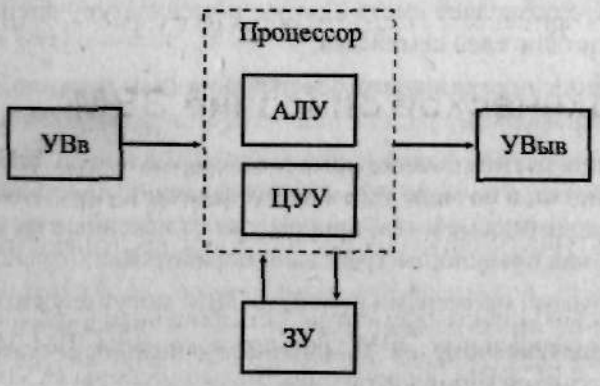


Рис. 1.3. Классическая архитектура ЭВМ

Программа вычислений (обработки информации) составляется в виде последовательности команд и загружается в память машины — *запоминающее устройство* (ЗУ). Там же хранятся исходные данные и промежуточные результаты обработки. *Центральное устройство управления* (ЦУУ) последовательно извлекает из памяти команды программы и организует их выполне-

ние. *Арифметико-логическое устройство* (АЛУ) предназначено для реализации операций преобразования информации. Программа и исходные данные вводятся в память машины через *устройства ввода* (УВв), а результаты обработки предъявляются на *устройства вывода* (УВыв).

Характерной особенностью архитектуры фон Неймана является то, что память представляет собой единое адресное пространство, предназначенное для хранения как программ, так и данных.

Такой подход, с одной стороны, обеспечивает большую гибкость организации вычислений — возможность перераспределения памяти между программой и данными, возможность самомодификации программы в процессе ее выполнения. С другой стороны, без принятия специальных мер защиты снижается надежность выполнения программы, что особенно недопустимо в управляющих системах.

Действительно, поскольку и команды программы, и данные кодируются в ЭВМ двоичными числами, теоретически возможно как разрушение программы (при обращении в область программы как к данным), так и попытка "выполнения" области данных как программы (при ошибочных переходах программы в область данных).

Альтернативной фон-неймановской является т. н. *гарвардская архитектура*. ЭВМ, реализованные по этому принципу, имеют два непересекающихся адресных пространства — для программы и для данных, причем программу нельзя разместить в свободной области памяти данных и наоборот. Гарвардская архитектура применяется главным образом в управляющих ЭВМ.

1.5. Иерархическое описание ЭВМ

ЭВМ как сложная система может быть адекватно описана на нескольких уровнях с применением различных языков описания на каждом из уровней.

Принципы структурного описания предполагают введение следующих понятий:

- *система* — совокупность элементов, объединенных в одно целое для достижения определенных целей. Для полного описания системы следует определить ее функции и структуру;
- *структура системы* — фиксированная совокупность элементов системы и связей между ними;
- *элемент* — неделимая часть системы, структура которого не рассматривается, а определяются только его функции.

Функции системы стремятся описывать в математической форме, иногда — в словесной (содержательной форме). Структура системы может быть задана

в виде графа или эквивалентных ему математических форм (матриц). Инженерной формой задания структуры является схема (отличается от графа только формой). Различным уровням представления систем соответствуют различные виды схем.

Свойства системы не являются простой суммой свойств входящих в нее элементов; за счет организации связей между элементами приобретается новое качество, отсутствующее в элементах. Например, радиокомпоненты → логические элементы → сумматор.

Для сложных систем характерно, что функция, реализуемая системой, не может быть представлена как композиция функций, реализуемых наименьшими элементами системы (иначе говоря, функцию сложной системы нельзя адекватно описать на одном языке). Действительно, функционирование ЭВМ нельзя описать лишь на языке электрических процессов, в ней происходящих. Функции ЭВМ как системы выявляются лишь при рассмотрении информационных и логических аспектов ее работы.

Поэтому в описании сложных систем используют несколько форм описания (языков) функций и структуры — иерархию функций и структуры. Иерархический подход к описанию сложных систем предполагает, что на высшем уровне иерархии система рассматривается как один элемент, имеющий входы и выходы для связи с внешней средой. В этом случае функция не может быть задана подробно и представляется как отображение состояний входов на состояние выходов системы.

Чтобы раскрыть устройство и порядок функционирования системы, глобальная функция и сама система разделяются на части — функции и структурные элементы следующего более низкого уровня иерархии и т. д. до тех пор, пока функции и структура системы не будут раскрыты полностью, с необходимой степенью детализации.

В этом случае элемент — это, прежде всего, удобное понятие, а не физическое свойство, т. к. один и тот же физический объект может рассматриваться как элемент на одном уровне иерархии и как система — на другом (более низком) уровне. В табл. 1.2 представлены основные уровни ЭВМ и языки описания этих уровней.

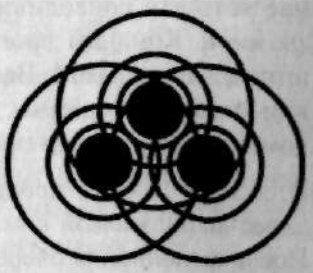
Таблица 1.2. Уровни описания ЭВМ

Уровень описания	Объект	Структурный базис	Язык описания
Электрические схемы	Логические и запоминающие элементы	Электронные и радиокомпоненты — транзисторы, резисторы и др.	Соотношения теории электрических цепей

Таблица 1.2 *(окончание)*

Уровень описания	Объект	Структурный базис	Язык описания
Логические схемы	Операционные элементы (счетчики, сумматоры, дешифраторы, регистры и т. д.) микропрограммные автоматы	Логические и запоминающие элементы	Булева алгебра, теория конечных автоматов
Операционные схемы	Операционные устройства: (арифметико-логическое устройство, устройство управления, запоминающее устройство и др.)	Операционные элементы, микропрограммные автоматы	Языки описания микроопераций
Структурные схемы	ЭВМ и системы	Операционные устройства	Языки машинных команд, микропрограмм
Программный уровень	Операционные системы, вычислительный процесс	Команды и операторы	Алгоритмические языки

ГЛАВА 2



Функциональная организация ЭВМ

Термин *"функциональная организация ЭВМ"* часто используют в качестве синонима (в некотором смысле) более широкого термина — *"архитектура ЭВМ"*, который, в свою очередь, трактуется разными авторами несколько в различных смыслах. Наиболее близким к трактовке автора может служить определение термина "архитектура ЭВМ", данное в [8]. Приведем это определение.

Архитектура ЭВМ — это абстрактное представление ЭВМ, которое отражает ее структурную, схемотехническую и логическую организацию. Понятие архитектуры ЭВМ является комплексным и включает в себя:

- структурную схему ЭВМ;
- средства и способы доступа к элементам структурной схемы;
- организацию и разрядность интерфейсов ЭВМ;
- набор и доступность регистров;
- организацию и способы адресации памяти;
- способы представления и форматы данных ЭВМ;
- набор машинных команд ЭВМ;
- форматы машинных команд;
- обработку нештатных ситуаций (прерываний).

В рамках данной книги мы, в основном, будем рассматривать перечисленные выше вопросы.

2.1. Командный цикл процессора

Командой называется элементарное действие, которое может выполнить процессор без дальнейшей детализации. Последовательность команд, выполне-

ние которых приводит к достижению определенной цели, называется *программой*. Команды программы кодируются двоичными словами и размещаются в памяти ЭВМ. Вся работа ЭВМ состоит в последовательном выполнении команд программы. Действия по выбору из памяти и выполнению одной команды называются *командным циклом*.

В составе любого процессора имеется специальная ячейка, которая хранит адрес выполняемой команды — *счетчик команд* или *программный счетчик*. После выполнения очередной команды его значение увеличивается на единицу (если код одной команды занимает несколько ячеек памяти, то содержимое счетчика команд увеличивается на длину команды). Таким образом осуществляется выполнение последовательности команд. Существуют специальные команды (передачи управления), которые в процессе своего выполнения модифицируют содержимое программного счетчика, обеспечивая переходы по программе. Сама выполняемая команда помещается в *регистр команд* — специальную ячейку процессора.

Во время выполнения командного цикла процессор реализует следующую последовательность действий:

1. Извлечение из памяти содержимого ячейки, адрес которой хранится в программном счетчике, и размещение этого кода в регистре команд (чтение команды).
2. Увеличение содержимого программного счетчика на единицу.
3. Формирование адреса операндов.
4. Извлечение операндов из памяти.
5. Выполнение заданной в команде операции.
6. Размещение результата операции в памяти.
7. Переход к п. 1.

Пункты 1, 2 и 7 обязательно выполняются в каждом командном цикле, остальные могут не выполняться в некоторых командах. Если длина кода команды составляет несколько машинных слов, то пп. 1 и 2 повторяются.

Фактически вся работа процессора заключается в циклическом выполнении пунктов 1—7 командного цикла. При запуске машины в счетчик команд аппаратно помещается фиксированное значение — начальный адрес программы (часто 0 или последний адрес памяти; встречаются и более экзотические способы загрузки начального адреса). В дальнейшем содержимое программного счетчика модифицируется в командном цикле. Прекращение выполнения командных циклов может произойти только при выполнении специальной команды "СТОП".

2.2. Система команд процессора

Разнообразие типов данных, форм их представления и действий, которые необходимы для обработки информации и управления ходом вычислений, порождает необходимость использования различных команд — набора команд. Каждый процессор имеет собственный вполне определенный набор команд, называемый *системой команд процессора*. Система команд должна обладать двумя свойствами — *функциональной полнотой* и *эффективностью*.

Функциональная полнота — это достаточность системы команд для описания любого алгоритма. Требование функциональной полноты не является слишком жестким. Доказано, что свойством функциональной полноты обладает система, включающая всего *три* команды (система Поста): *присвоение 0*, *присвоение 1*, *проверка на 0*. Однако составление программ в такой системе команд крайне неэффективно.

Эффективность системы команд — степень соответствия системы команд назначению ЭВМ, т. е. классу алгоритмов, для выполнения которых предназначается ЭВМ, а также требованиям к производительности ЭВМ. Очевидно, что реализация развитой системы команд связана с большими затратами оборудования и, следовательно, с высокой стоимостью процессора. В то же время ограниченный набор команд приводит к снижению производительности и повышенным требованиям к памяти для размещения программы. Даже простые и дешевые современные микропроцессоры поддерживают систему команд, содержащую несколько десятков (а с модификациями — сотен) команд.

Система команд процессора характеризуется тремя аспектами: форматами, способами адресации и системой операций.

2.2.1. Форматы команд

Под *форматом команды* следует понимать длину команды, количество, размер, положение, назначение и способ кодировки ее полей.

Команды, как и любая информация в ЭВМ, кодируются двоичными словами, которые должны содержать в себе следующие виды информации:

- тип операции, которую следует реализовать в данной команде (КОП);
- место в памяти, откуда следует взять первый операнд (А1);
- место в памяти, откуда следует взять второй операнд (А2);
- место в памяти, куда следует поместить результат (А3).

Каждому из этих видов информации соответствует своя часть двоичного слова — поле, а совокупность полей (их длины, расположение в командном сло-

ве, способ кодирования информации) называется форматом команды. В свою очередь, некоторые поля команды могут делиться на подполя. Формат команды, поля которого перечислены выше, называется *трехадресным* (рис. 2.1, *а*).

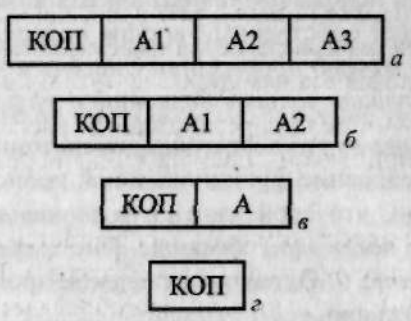


Рис. 2.1. Форматы команд: *а* — трехадресный; *б* — двухадресный; *в* — одноадресный; *г* — безадресный

Команды трехадресного формата занимают много места в памяти, в то же время далеко не всегда поля адресов используются в командах эффективно. Действительно, наряду с двухместными операциями (сложение, деление, конъюнкция и др.) встречаются и одноместные (инверсия, сдвиг, инкремент и др.), для которых третий адрес не нужен. При выполнении цепочки вычислений часто результат предыдущей операции используется в качестве операнда для следующей. Более того, нередко встречаются команды, для которых операнды не определены (СТОП) или подразумеваются самим кодом операций (DAA, десятичная коррекция аккумулятора).

Поэтому в системах команд реальных ЭВМ трехадресные команды встречаются редко. Чаще используются *двухадресные команды* (рис. 2.1, *б*), в этом случае в бинарных операциях результат помещается на место одного из операндов.

Для реализации одноадресных форматов (рис. 2.1, *в*) в процессоре предусматривают специальную ячейку — *аккумулятор*. Первый операнд и результат всегда размещаются в аккумуляторе, а второй операнд адресуется полем А.

Реальная система команд обычно имеет команды нескольких форматов, причем тип формата определяется в поле КОП.

2.2.2. Способы адресации

Способ адресации определяет, каким образом следует использовать информацию, размещенную в поле адреса команды.

Не следует думать, что во всех случаях в поле адреса команды помещается адрес операнда. Существует пять основных способов адресации операндов в командах.

- *Прямая* — в этом случае в адресном поле располагается адрес операнда. Разновидность — *прямая регистровая* адресация, адресующая не ячейку памяти, а РОН. Поле адреса регистра имеет в команде значительно меньшую длину, чем поле адреса памяти.
- *Непосредственная* — в поле адреса команды располагается не адрес операнда, а сам операнд. Такой способ удобно использовать в командах с константами.
- *Косвенная* — в поле адреса команды располагается адрес ячейки памяти, в которой хранится адрес операнда ("адрес адреса"). Такой способ позволяет оперировать адресами как данными, что облегчает организацию циклов, обработку массивов данных и др. Его основной недостаток — потеря времени на двойное обращение к памяти — сначала за адресом, потом — за операндом. Разновидность — *косвенно-регистровая* адресация, при которой в поле команды размещается адрес РОН, хранящего адрес операнда. Этот способ, помимо преимущества обычной косвенной адресации, позволяет обращаться к большой памяти с помощью коротких команд и не требует двойного обращения к памяти (обращение к регистру занимает гораздо меньше времени, чем к памяти).
- *Относительная* — адрес формируется как сумма двух слагаемых: базы, хранящейся в специальном регистре или в одном из РОН, и смещения, извлекаемого из поля адреса команды. Этот способ позволяет сократить длину команды (смещение может быть укороченным, правда в этом случае не вся память доступна в команде) и/или перемещать адресуемые массивы информации по памяти (изменяя базу). Разновидности — *индексная* и *базово-индексная* адресации. Индексная адресация предполагает наличие индексного регистра вместо базового. При каждом обращении содержимое индексного регистра автоматически модифицируется (обычно увеличивается или уменьшается на 1). Базово-индексная адресация формирует адрес операнда как сумму трех слагаемых: базы, индекса и смещения.
- *Безадресная* — поле адреса в команде отсутствует, а адрес операнда или не имеет смысла для данной команды, или подразумевается по умолчанию. Часто безадресные команды подразумевают действия над содержимым аккумулятора. Характерно, что безадресные команды нельзя применить к другим регистрам или ячейкам памяти.

  Одной из разновидностей безадресного обращения является использование т. н. магазинной памяти или *стека*. Обращение к такой памяти напоминает

обращение с магазином стрелкового оружия. Имеется фиксированная ячейка, называемая *верхушкой стека*. При чтении слово извлекается из верхушки, а все остальное содержимое "поднимается вверх" подобно патронам в магазине, так что в верхушке оказывается следующее по порядку слово. Одно слово нельзя прочитать из стека дважды. При записи новое слово помещается в верхушку стека, а все остальное содержимое "опускается вниз" на одну позицию. Таким образом, слово, помещенное в стек первым, будет прочитано последним. Говорят, что стек поддерживает дисциплину LIFO — Last In First Out (последний пришел — первый ушел). Реже используется безадресная память типа *очередь* с дисциплиной FIFO — First In First Out (первый пришел — первый ушел).

2.2.3. Система операций

Все операции, выполняемые в командах ЭВМ, принято делить на пять классов.

- *Арифметико-логические* и *специальные* — команды, в которых выполняется собственно преобразование информации. К ним относятся арифметические операции сложение, вычитание, умножение и деление (с фиксированной и плавающей занятой), команды десятичной арифметики, логические операции конъюнкции, дизъюнкции, инверсии и др., сдвиги, преобразование чисел из одной системы счисления в другую и такие экзотические, как извлечение корня, решение системы уравнений и др. Конечно, очень редко встречаются ЭВМ, система команд которых включает все эти команды.
- *Пересылки* и *загрузки* — обеспечивают передачу информации между процессором и памятью или между различными уровнями памяти (СОЗУ ↔ ОЗУ). Разновидность — *загрузка регистров и ячеек константами*.
- *Ввода/вывода* — обеспечивают передачу информации между процессором и внешними устройствами. По структуре они очень похожи на команды предыдущего класса. В некоторых ЭВМ принципиально отсутствует различие между ячейками памяти и регистрами внешних устройств (единое адресное пространство) и класс команд ввода/вывода не выделяется, все обмены осуществляются в рамках команд пересылки и загрузки.
- *Передачи управления* — команды, которые изменяют естественный порядок выполнения команд программы. Эти команды меняют содержимое программного счетчика, обеспечивая переходы по программе. Существуют команды безусловной и условной передачи управления. В последнем случае передача управления происходит, если выполняется заданное в коде команды условие, иначе выполняется следующая по порядку команда.

В качестве условий обычно используются признаки результата предыдущей операции, которые хранятся в специальном регистре признаков (флажков). Чаще всего формируются и проверяются признаки нулевого результата, отрицательного результата, наличия переноса из старшего разряда, четности числа единиц в результате и др. Различают три разновидности команд передачи управления:

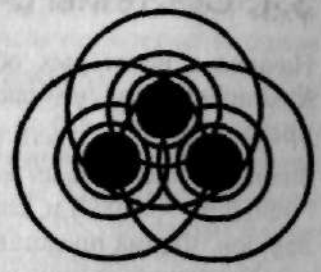
- переходы;
- вызовы подпрограмм;
- возвраты из подпрограмм.

Команды *переходов* помещают в программный счетчик содержимое своего адресного поля — адрес перехода. При этом старое значение программного счетчика теряется. В микроЭВМ часто для экономии длины адресного поля команд условных переходов адрес перехода формируется как сумма текущего значения программного счетчика и относительно короткого знакового смещения, размещаемого в команде. В крайнем случае, в командах условных переходов можно и вовсе обойтись без адресной части — при выполнении условия команда "перепрыгивает" через следующую команду, которой обычно является безусловный переход.

Команда *вызова* подпрограммы работает подобно команде безусловного перехода, но старое значение программного счетчика предварительно сохраняется в специальном регистре или в стеке. Команда возврата передает содержимое верхушки стека или специального регистра в программный счетчик. Команды *вызова* и *возврата* работают "в паре". Подпрограмма, вызываемая командой вызова, должна заканчиваться командой возврата, что обеспечивает по окончании работы подпрограммы передачу управления в точку вызова. Хранение адресов возврата в стеке обеспечивает возможность реализации вложенных подпрограмм.

- *Системные* — команды, выполняющие управление процессом обработки информации и внутренними ресурсами процессора. К таким командам относятся команды управления подсистемой прерывания, команды установки и изменения параметров защиты памяти, команда останова программы и некоторые другие. В простых процессорах класс системных команд немногочисленный, а в сложных мультипрограммных системах предусматривается большое число системных команд.

ГЛАВА 3



Арифметические основы ЭВМ

Безусловно, одним из основных направлений применения компьютеров были и остаются разнообразные вычисления. Обработка числовой информации ведется и при решении задач, на первый взгляд не связанных с какими-то расчетами, например, при использовании компьютерной графики или звука.

В связи с этим встает вопрос о выборе *оптимального представления чисел в компьютере*. Безусловно, можно было бы использовать 8-битное (байтовое) кодирование отдельных цифр, а из них составлять числа. Однако такое кодирование не будет оптимальным, что легко увидеть из простого примера. Пусть имеется двузначное число 13. При 8-битном кодировании отдельных цифр в кодах ASCII его представление выглядит следующим образом: 0011000100110011, т. е. код имеет длину 16 битов; если же определять это число просто в двоичном коде, то получим 4-битную цепочку 1101.

Важно, что представление определяет не только способ записи данных (букв или чисел), но и допустимый набор операций над ними; в частности, буквы могут быть только помещены в некоторую последовательность (или исключены из нее) без изменения их самих; над числами же возможны *операции, изменяющие само число, например, извлечение корня или сложение с другим* числом.

Представление чисел в компьютере имеет две особенности:

- числа записываются в двоичной системе счисления (в отличие от привычной десятичной);
- для записи и обработки чисел отводится конечное количество разрядов (в "некомпьютерной" арифметике такое ограничение отсутствует).

Следствия, к которым приводят эти отличия, и рассматриваются в данной главе.

3.1. Системы счисления

Начнем с некоторых общих замечаний относительно понятия *число* [12]. Можно считать, что любое число имеет значение (содержание) и форму представления.

Значение числа задает его отношение к значениям других чисел ("больше", "меньше", "равно") и, следовательно, порядок расположения чисел на числовой оси. Форма представления, как следует из названия, определяет порядок записи числа с помощью предназначенных для этого знаков. При этом значение числа является инвариантом, т. е. не зависит от способа его представления. Это означает также, что число с одним и тем же значением может быть записано по-разному, т. е. отсутствует взаимно однозначное соответствие между представлением числа и его значением.

В связи с этим возникают вопросы, во-первых, о формах представления чисел и, во-вторых, о возможности и способах перехода от одной формы к другой.

Способ представления числа определяется *системой счисления*.

Определение

Система счисления — это правило записи чисел с помощью заданного набора специальных знаков — цифр.

Людьми использовались различные способы записи чисел, которые можно объединить в несколько групп: унарная, непозиционные и позиционные.

Унарная — это система счисления, в которой для записи чисел используется только один знак — | (вертикальная черта, палочка). Следующее число получается из предыдущего добавлением новой палочки: их количество (сумма) равно самому числу. Унарная система важна в теоретическом отношении, поскольку в ней число представляется наиболее простым способом и, следовательно, просты операции с ним. Кроме того, именно унарная система определяет значение целого числа количеством содержащихся в нем единиц, которое, как было сказано, не зависит от формы представления.

Из *непозиционных* наиболее распространенной можно считать римскую систему счисления. В ней некоторые базовые числа обозначены заглавными латинскими буквами: 1 — I, 5 — V, 10 — X, 50 — L, 100 — C, 500 — D, 1000 — M. Все другие числа строятся комбинаций базовых в соответствии со следующими правилами:

- если цифра меньшего значения стоит справа от большей цифры, то их значения суммируются; если слева — то меньшее значение вычитается из большего;

- цифры I, X, C и M могут следовать подряд не более трех раз каждая;
- цифры V, L и D могут использоваться в записи числа не более одного раза.

Например, запись XIX соответствует числу 19, MDXLIX — числу 1549. Запись чисел в такой системе громоздка и неудобна, но еще более неудобным оказывается выполнение в ней даже самых простых арифметических операций. Отсутствие нуля и знаков для чисел больше M не позволяют римскими цифрами записать любое число (хотя бы натуральное). По указанным причинам теперь римская система используется лишь для нумерации.

В настоящее время для представления чисел применяют, в основном, *позиционные системы счисления*.

Определение

Позиционными называются системы счисления, в которых значение каждой цифры в изображении числа определяется ее положением (позицией) в ряду других цифр.

Наиболее распространенной и привычной является система счисления, в которой для записи чисел используется 10 цифр: 0, 1, 2, 3, 4, 5, 6, 7, 8 и 9. Число представляет собой краткую запись многочлена, в который входят степени некоторого другого числа — основания системы счисления. Например:

$$575{,}15 = 5\cdot 10^2 + 7\cdot 10^1 + 5\cdot 10^0 + 1\cdot 10^{-1} + 5\cdot 10^{-2}.$$

В данном числе цифра 5 встречается трижды, однако значение этих цифр различно и определяется их положением (позицией) в числе. Количество цифр для построения чисел, очевидно, равно основанию системы счисления. Также очевидно, что максимальная цифра на 1 меньше основания. Причина широкого распространения именно десятичной системы счисления понятна — она происходит от унарной системы с пальцами рук в качестве "палочек".

Однако в истории человечества имеются свидетельства использования и других систем счисления — пятеричной, шестеричной, двенадцатиричной, двадцатиричной и даже шестидесятиричной. Общим для унарной и римской систем счисления является то, что значение числа в них определяется посредством операций сложения и вычитания базисных цифр, из которых составлено число, *независимо от их позиции в числе*. Такие системы получили название *аддитивных*.

В отличие от них позиционное представление следует считать *аддитивно-мультипликативным*, поскольку значение числа определяется операциями умножения и сложения. Главной же особенностью позиционного представле-

ния является то, что в нем посредством конечного набора знаков (цифр, разделителя десятичных разрядов и обозначения знака числа) можно записать неограниченное количество различных чисел. Кроме того, в позиционных системах гораздо легче, чем в аддитивных, осуществляются операции умножения и деления. Именно эти обстоятельства обуславливают доминирование позиционных систем при обработке чисел как человеком, так и компьютером.

По принципу, положенному в основу десятичной системы счисления, очевидно, можно построить системы с иным основанием. Пусть p — основание системы счисления. Тогда любое число Z (пока ограничимся только целыми числами), удовлетворяющее условию $Z < p^k$ ($k > 0$, целое), может быть представлено в виде многочлена со степенями (при этом, очевидно, максимальный показатель степени будет равен $k-1$):

$$Z_p = a_{k-1} \cdot p^{k-1} + a_{k-2} \cdot p^{k-2} + \ldots + a_1 \cdot p^1 + a_0 \cdot p^0 = \sum_{j=1}^{k-1} a_j \cdot p^j . \qquad (3.1)$$

Из коэффициентов a_j при степенях основания строится сокращенная запись числа:

$$Z_p = (a_{k-1} a_{k-2} \ldots a_1 a_0).$$

Индекс p числа Z указывает, что оно записано в системе счисления с основанием p: общее число цифр числа равно k. Все коэффициенты a_j — целые числа, удовлетворяющие условию: $0 < a_j < p-1$.

Уместно задаться вопросом: каково минимальное значение p? Очевидно, $p = 1$ невозможно, поскольку тогда все $a_j = 0$ и форма (3.1) теряет смысл. Первое допустимое значение $p = 2$ — оно и является минимальным для позиционных систем.

Система счисления с основанием 2 называется *двоичной*. Цифрами двоичной системы являются 0 и 1, а форма (3.1) строится по степеням 2. Интерес именно к этой системе счисления связан с тем, что, как указывалось выше, любая информация в компьютерах представляется с помощью двух состояний — 0 и 1, которые легко реализуются технически.

Наряду с двоичной в компьютерах используются восьмеричная и шестнадцатеричная системы счисления — причины будут рассмотрены далее.

3.2. Представление чисел в различных системах счисления

Очевидно, что значение целого числа, т. е. общее количество входящих в него единиц, не зависит от способа его представления и остается одинаковым во всех системах счисления; различаются только *формы представления* одного и того же количественного содержания числа.

Например: $|||||_1 = 5_{10} = 101_2 = 5_{16}$.

Поскольку одно и то же число может быть записано в различных системах счисления, встает вопрос о переводе представления числа из одной системы в другую.

3.2.1. Перевод целых чисел из одной системы счисления в другую

Обозначим преобразование числа Z, представленного в p-ричной системе счисления в представление в q-ричной системе как $Z_p \to Z_q$. Теоретически *возможно произвести его при любых* q *и* p. *Однако подобный прямой* перевод будет затруднен тем, что придется выполнять операции по правилам арифметики недесятичных систем счисления (полагая в общем случае, что p, $q \neq 10$).

По этой причине более удобными с практической точки зрения оказываются *варианты преобразования с промежуточным переводом* $Z_p \to Z_r \to Z_q$ с основанием r, для которого арифметические операции выполнить легко. Такими удобными основаниями являются $r = 1$ и $r = 10$, т. е. перевод осуществляется через унарную или десятичную систему счисления.

Преобразование $Z_p \to Z_1 \to Z_q$

Идея алгоритма перевода предельно проста: положим начальное значение $Z_q := 0$; из числа Z_p вычтем 1 по правилам вычитания системы p, т. е. $Z_p := Z_p - 1$, и добавим ее к Z_q по правилам сложения системы q, т. е. $Z_q := Z_q + 1$. Будем повторять эту последовательность действий, пока не достигнем $Z_p = 0$. Правила сложения с 1 (инкремента) и вычитания 1 (декремента) могут быть записаны так, как представлено в табл. 3.1.

Таблица 3.1. Правила сложения и вычитания 1

Для системы p	Для системы q
$(p-1)-1=p-2$	$0+1=1$
$(p-2)-1=p-3$	$1+1=2$
...	...
$1-1=0$	$(q-2)+1=q-1$
$0-1=\pi(p-1)$	$(q-1)+1=\pi.0$

Примечание: π — перенос в случае инкремента или заем в случае декремента.

Промежуточный переход к унарной системе счисления в данном случае осуществляется неявно — используется упоминавшееся выше свойство независимости значения числа от формы его представления. Рассмотренный алгоритм перевода может быть легко реализован программным путем.

Преобразование $Z_p \to Z_w \to Z_q$

Очевидно, первая и вторая часть преобразования не связаны друг с другом, что дает основание рассматривать их по отдельности. Алгоритмы перевода $Z_w \to Z_q$ вытекают из следующих соображений. Многочлен (3.1) для Z_q может быть представлен в виде:

$$Z_q = \sum_{j=0}^{m-1} b_j \cdot q^j = ((\ldots(b_{m-1} \cdot q + b_{m-2}) \cdot q + b_{m-3}) \cdot q + \ldots + b_1) \cdot q + b_0, \qquad (3.2)$$

где m — число разрядов в записи Z_p, а $b_j\,(j=0,\ldots,m-1)$ — цифры числа Z_q.

Разделим число Z_q на две части по разряду номер i. Число, включающее $m-i$ разрядов с $(m-i)$-го по i-й, обозначим γ_i, а число с i разрядами с $(i-1)$-го по 0-й — δ_i. Очевидно, $i \in [0, m-1]$, $\gamma_0 = \delta_{m-1} = Z_q$.

$$Z_q = (\underbrace{b_{m-1}b_{m-2}\ldots b_i}_{\gamma_i}\underbrace{b_{i-1}\ldots b_1 b_0}_{\delta_i}).$$

Позаимствуем из языка PASCAL обозначение двух операций: `div` — результат целочисленного деления двух целых чисел и `mod` — остаток от целочисленного деления (`13 div 4 = 3; 13 mod 4 = 1`).

Теперь если принять $\gamma_{m-1} = b_{m-1}$, то в (3.2) усматривается следующее рекуррентное соотношение: $\gamma_i = \gamma_{i+1} + b_i$, из которого, в свою очередь, получаются выражения:

$$\gamma_{i+1} = \gamma_i \text{ div } q\,;\ b_i = \gamma_i \bmod q\,. \tag{3.3}$$

Аналогично, если принять $\delta_0 = b_0$, то для правой части числа будет справедливо другое рекуррентное соотношение: $\delta_i = \delta_{i-1} + b_i q^i$, из которого следуют:

$$b_i = \delta_i \text{ div } q^i\,;\ \delta_{i-1} = \delta_i \bmod q^i\,. \tag{3.4}$$

Из соотношений (3.3) и (3.4) непосредственно вытекают два способа перевода целых чисел из десятичной системы счисления в систему с произвольным основанием q.

Способ 1 является следствием соотношений (3.3), предполагающий *следующий алгоритм перевода:*

1. Целочисленно разделить исходное число (Z_{10}) на основании новой системы счисления (q) и найти остаток от деления — это будет цифра 0-го разряда числа Z_q.
2. Частное от деления снова целочисленно разделить на q с выделением остатка; процедуру продолжать до тех нор, пока частное от деления не окажется меньше q.
3. Образовавшиеся остатки от деления, поставленные в порядке, обратном порядку их получения, и представляют Z_q.

Пример 3.1

Выполнить преобразование $123_{10} \to Z_5$. Результат — на рис. 3.1.

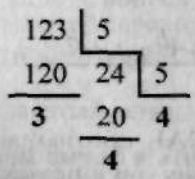


Рис. 3.1. Результат выполнения примера 3.1

Остатки от деления (3, 4) и результат последнего целочисленного деления (4) образуют обратный порядок цифр нового числа. Следовательно, $123_{10} = 443_5$.

Необходимо заметить, что полученное число нельзя читать как "четыреста сорок три", поскольку десятки, сотни, тысячи и прочие подобные обозначения чисел относятся только к десятичной системе счисления. Прочитывать число следует простым перечислением его цифр с указанием системы счисления ("число четыре, четыре, три в пятеричной системе счисления").

Способ 2 вытекает из соотношения (3.4), действия производятся в соответствии со следующим алгоритмом:

1. Определить $m-1$ — максимальный показатель степени в представления числа по форме (3.1) для основания q.
2. Целочисленно разделить исходное число (Z_{10}) на основание новой системы счисления в степени $m-1$ (т. е. q^{m-1}) и найти остаток от деления; результат деления определит первую цифру числа Z_q.
3. Остаток от деления целочисленно разделить на g^{m-2}, результат деления принять за вторую цифру нового числа; найти остаток; продолжать эту последовательность действий, пока показатель степени q не достигнет значения 0.

Продемонстрируем действие алгоритма на той же задаче, что была рассмотрена выше.

Определить $m-1$ можно либо путем подбора ($5^0 = 1 < 123$; $5^1 = 5 < 123$; $5^2 = 25 < 123$; $5^3 = 125 > 123$, следовательно, $m-1 = 2$), либо логарифмированием с оставлением целой части логарифма ($\log_5 123 = 2{,}99$, т. е. $m-1 = 2$).

Далее:

$b_2 = 123 \text{ div } 5^2 = 4$ $\quad \delta_1 = 23 \bmod 5^2 = 23$ $\quad i = 2-1 = 1$

$b_1 = 23 \text{ div } 5^1 = 4$ $\quad \delta_0 = 23 \bmod 5^1 = 3$ $\quad i = 0$

Алгоритмы перевода $Z_g \to Z_w$ явно вытекают из представлений (3.1) или (3.2): необходимо Z_p представить в форме многочлена и выполнить все операции по правилам десятичной арифметики.

Пример 3.2

Выполнить преобразование $443_5 \to Z_{10}$.

Решение:

$$443_5 = 4 \cdot 5^2 + 4 \cdot 5^1 + 3 \cdot 5^0 = 4 \cdot 25 + 4 \cdot 5 + 3 \cdot 1 = 123_{10}.$$

Необходимо еще раз подчеркнуть, что приведенными алгоритмами удобно пользоваться при переводе числа из десятичной системы в какую-то иную или наоборот. Они работают и для перевода между любыми иными системами счисления, однако преобразование будет затруднено тем, что все арифметические операции необходимо осуществлять *по правилам исходной* (в первых алгоритмах) или *конечной* (в последнем алгоритме) системы счисления.

По этой причине переход, например, $Z_3 \to Z_8$ проще осуществить через промежуточное преобразование к десятичной системе $Z_3 \to Z_{10} \to Z_8$. Ситуация, однако, значительно упрощается, если основания исходной и конечной систем счисления оказываются связанными соотношением $p = q^r$, где r — целое число (естественно, большее 1) или $r = 1/n$ ($n > 1$, целое) — эти случаи будут рассмотрены далее.

3.2.2. Перевод дробных чисел из одной системы счисления в другую

Вещественное число, в общем случае содержащее целую и дробную часть, всегда можно представить в виде суммы целого числа и правильной дроби. Поскольку в предыдущем разделе проблема записи натуральных чисел в различных системах счисления уже была решена, можно ограничить рассмотрение только алгоритмами перевода правильных дробей.

Введем следующие обозначения: правильную дробь в исходной системе счисления p будем записывать в виде $0,Y_p$, дробь в системе q — $0,Y_q$, а преобразование — в виде $0,Y_p \to 0,Y_q$.

Последовательность рассуждений весьма напоминает проведенную ранее для натуральных чисел. В частности, это касается рекомендации осуществлять преобразование через промежуточный переход к десятичной системе, чтобы избежать необходимости производить вычисления в "непривычных" системах счисления, т. е. $0,Y_p \to 0,Y_{10} \to 0,Y_q$.

Это, в свою очередь, разбивает задачу на две составляющие: преобразование $0,Y_p \to 0,Y_{10}$ и $0,Y_{10} \to 0,Y_q$, каждое из которых может рассматриваться независимо.

Алгоритмы перевода $0,Y_{10} \to 0,Y_q$ выводятся путем следующих рассуждений. Если основание системы счисления q, простая дробь содержит n цифр, и b_k — цифры дроби ($1 < b_k < n$, $0 < b_k < n-1$), то она может быть представлена в виде суммы:

$$0,Y_q = \sum_{k=1}^{n} b_k \cdot q^{-k} = \frac{1}{q}(b_1 + \frac{1}{q}(b_2 + \ldots + \frac{1}{q}(b_{n-1} + \frac{1}{q}b_n)\ldots), \quad (3.5)$$

$$0,Y_q = (0,b_1 b_1 \ldots \underbrace{b_i b_{i+1} \ldots b_n}_{\varepsilon_i}).$$

Часть дроби от разряда i до ее конца обозначим ε_i и примем $\varepsilon_n = b_n/q$. Очевидно, $\varepsilon_1 = 0,Y_q$, тогда в (3.5) легко усматривается рекуррентное соотношение:

$$\varepsilon_i = \frac{1}{q}(b_i + \varepsilon_{i+1}). \quad (3.6)$$

Если вновь позаимствовать в PASCAL обозначение функции — на этот раз `trunc`, производящей округление целого вещественного числа путем отбрасывания его дробной части, то следствием (3.6) будут соотношения, позволяющие находить цифры новой дроби:

$$b_i = \text{trunc}(q \cdot \varepsilon_i);\ \varepsilon_{i+1} = q \cdot \varepsilon_i - \text{trunc}(q \cdot \varepsilon_i). \quad (3.7)$$

Соотношения (3.7) задают алгоритм преобразования: $0,Y_{10} \to 0,Y_q$:

1. Умножить исходную дробь в десятичной системе счисления на q, выделить целую часть — она будет первой (старшей) цифрой новой дроби; отбросить целую часть.
2. Для оставшейся дробной части операцию умножения с выделением целой и дробных частей повторять, пока в дробной части не окажется 0 или не будет достигнута желаемая точность конечного числа; появляющиеся при этом целые будут цифрами новой дроби.
3. Записать дробь в виде последовательности цифр после нуля с разделителем в порядке их появления в пп. 1 и 2.

Пример 3.3

Выполнить преобразование $0{,}375_{10} \to 0,Y_2$. Результат — на рис. 3.2.

Таким образом, $0{,}375_{10} \to 0{,}011_2$.

0,375 × 2 =	**0,**	750
0,75 × 2 =	**1,**	50
0,5 × 2 =	**1,**	0

Рис. 3.2. Результат выполнения примера 3.3

Перевод $0,Y_p \to 0,Y_{10}$, как и в случае натуральных чисел, сводится к вычислению значения формы (3.5) в десятичной системе счисления. Например:

$$0,011_2 = 0 \cdot 2^{-1} + 1 \cdot 2^{-2} + 1 \cdot 2^{-3} = 0 + 0,25 + 0,125 = 0,375_{10}.$$

Следует сознавать, что после перевода дроби, которая была конечной в исходной системе счисления, она может оказаться бесконечной в новой системе. Соответственно, рациональное число в исходной системе может после перехода превратиться в иррациональное. Справедливо и обратное утверждение: число иррациональное в исходной системе счисления в иной системе может оказаться рациональным.

Пример 3.4

Выполнить преобразование $5,3(3)_{10} \to Y_8$.

Перевод целой части, очевидно, дает: $5_{10} = 12_3$. Перевод дробной части: $0,3(3)_{10} \to 0,1$. Окончательно: $5,3(3)_{10} \to 12,1_3$.

Как уже было сказано, значение целого числа не зависит от формы его представления и выражает количество входящих в него единиц. Простая дробь имеет смысл доли единицы, и это "дольное" содержание также не зависит от выбора способа представления. Другими словами, треть пирога остается третью в любой системе счисления.

3.2.3. Перевод чисел между системами счисления 2 ↔ 8 ↔ 16

Интерес к двоичной системе счисления вызван тем, что именно она используется для представления чисел в компьютере. Однако двоичная запись оказывается громоздкой, поскольку содержит много цифр и, кроме того, плохо воспринимается и запоминается человеком из-за зрительной однородности (все число состоит из нулей и единиц). Поэтому в нумерации ячеек памяти компьютера, записи кодов команд, нумерации регистров и устройств и пр. используются системы счисления с основаниями 8 и 16. Выбор именно этих

систем счисления обусловлен тем, что переход от них к двоичной системе и обратно осуществляется, как будет показано далее, весьма простым образом.

Двоичная система счисления имеет основанием 2 и, соответственно, две цифры: 0 и 1.

Восьмеричная система счисления имеет основание 8 и цифры 0, 1, ..., 7.

Шестнадцатеричная система счисления имеет основание 16 и цифры 0, 1, ..., 9, A, B, C. D, E, F. При этом знак A является шестнадцатеричной цифрой, соответствующей числу 10 в десятичной системе, $B_{16} = 11_{10}$, $C_{16} = 12_{10}$, $D_{16} = 13_{10}$, $E_{16} = 14_{10}$ и $F_{16} = 15_{10}$. Другими словами, в данном случае A, ..., F — это не буквы латинского алфавита, а *цифры шестнадцатеричной системы счисления*.

Докажем две теоремы [12].

Теорема 1. Для преобразования целого числа $Z_p \to Z_q$ в том случае, если системы счисления связаны соотношением $q = p^r$, где r — целое число, большее 1, достаточно Z_p разбить справа налево на группы по r цифр и каждую из них независимо перевести в систему q.

Доказательство. Пусть максимальный показатель степени в записи числа p по форме (3.1) равен $k-1$, причем, $2r > k-1 > r$.

$$Z_p = (a_{k-1} \ldots a_1 a_0) = a_{k-1} \cdot p^{k-1} + a_{k-2} \cdot p^{k-2} + \ldots a_1 \cdot p^1 + a_0 \cdot p^0.$$

Вынесем множитель p^r из всех слагаемых, у которых $j \geq r$. Получим:

$$Z_p = (a_{k-1} \cdot p^{k-1-r} + a_{k-2} \cdot p^{k-2-r} + \ldots a_{r+1} \cdot p^1 + a_r \cdot p^0) \cdot p^r +$$
$$+ (a_{r-1} \cdot p^{r-1} + a_{r-1} \cdot p^{r-2} + \ldots a_1 \cdot p^1 + a_0 \cdot p^0) \cdot p^0 = b_1 \cdot q^1 + b_0 \cdot q^0,$$

где

$$b_1 = a_{k-1} \cdot p^{k-1-r} + a_{k-2} \cdot p^{k-2-r} + \ldots a_{r+1} \cdot p^1 + a_r \cdot p^0 = (a_{k-1} \ldots a_r)_p,$$
$$b_0 = a_{r-1} \cdot p^{r-1} + a_{r-1} \cdot p^{r-2} + \ldots a_1 \cdot p^1 + a_0 \cdot p^0 = (a_{r-1} \ldots a_0)_p.$$

Таким образом, r-разрядные числа системы с основанием p оказываются записанными как цифры системы с основанием q. Этот результат можно обобщить на ситуацию произвольного $k-1 > r$ — в этом случае выделятся не две, а больше (m) цифр числа с основанием q. Очевидно, $Z_q = (b_m \ldots b_0)_q$.

Теорема 2. Для преобразования целого числа $Z_p \to Z_q$ в том случае, если системы счисления связаны соотношением $p = q^r$, где r — целое число, большее 1, достаточно каждую цифру Z_p заменить соответствующим r-разрядным числом в системе счисления q, дополняя его при необходимости незначащими нулями слева до группы в r цифр.

Доказательство. Пусть исходное число содержит две цифры, т. е.

$$Z_p = (a_1 a_0)_p = a_1 \cdot p^1 + a_0 \cdot p^0.$$

Для каждой цифры справедливо: $0 \le a_i \le p-1$ и поскольку $p = q^r$, $0 \le a_i \le q^r - 1$, то в представлении этих цифр в системе счисления q максимальная степень многочленов (3.1) будет не более $r-1$ и эти многочлены будут содержать по r цифр:

$$a_1 = b_{r-1}^{(1)} \cdot q^{r-1} + b_{r-2}^{(1)} \cdot q^{r-2} + \ldots + b_1^{(1)} \cdot q^1 + b_0^{(1)} \cdot q^0;$$

$$a_1 = b_{r-1}^{(0)} \cdot q^{r-1} + b_{r-2}^{(0)} \cdot q^{r-2} + \ldots + b_1^{(0)} \cdot q^1 + b_0^{(0)} \cdot q^0.$$

Тогда

$$Z_p = (a_1 a_0)_p = (b_{r-1}^{(1)} \cdot q^{r-1} + b_{r-2}^{(1)} \cdot q^{r-2} + \ldots + b_1^{(1)} \cdot q^1 + b_0^{(1)} \cdot q^0) \cdot q^r + (b_{r-1}^{(0)} \cdot q^{r-1} +$$
$$+ b_{r-2}^{(0)} \cdot q^{r-2} + \ldots + b_1^{(0)} \cdot q^1 + b_0^{(0)} \cdot q^0) \cdot q^0 = b_{r-1}^{(1)} \cdot q^{2r-1} + b_{r-2}^{(1)} \cdot q^{2r-2} +$$
$$+ \ldots + b_1^{(1)} \cdot q^{r+1} + b_0^{(1)} \cdot q^r + b_{r-1}^{(0)} \cdot q^{r-1} + b_{r-2}^{(0)} \cdot q^{r-2} + \ldots +$$
$$+ b_1^{(0)} \cdot q^1 + b_0^{(0)} \cdot q^0 = (b_{r-1}^{(1)} b_{r-2}^{(1)} \ldots b_1^{(0)} b_0^{(0)}) = Z_q,$$

причем число Z_q содержит $2r$ цифр. Доказательство легко обобщается на случай произвольного количества цифр в числе Z_p.

3.2.4. Понятие экономичности системы счисления

Число в системе счисления с k разрядами, очевидно, будет иметь наибольшее значение в том случае, если все цифры числа окажутся максимальными, т. е. равными $p-1$. Тогда

$$(Z_p)^{\max} = \underbrace{< p-1 > \ldots < p-1 >}_{k \text{ цифр}} = p^k - 1. \qquad (3.8)$$

Количество разрядов числа при переходе от одной системы счисления к другой в общем случае меняется.

Очевидно, если $p = q^{\sigma}$ (σ — не обязательно целое), то

$$(Z_p)^{\max} = p^k - 1 = q^{\sigma k} - 1,$$

т. е. количество разрядов числа в системах счисления p и q будут различаться в σ раз, причем

$$\sigma = \frac{\log p}{\log q}. \qquad (3.9)$$

При этом основание логарифма никакого значения не имеет, поскольку σ определяется отношением логарифмов. Сравним количество цифр в числе 99_{10} и его представлении в двоичной системе счисления: $99_{10} = 1100011_2$, т. е. двоичная запись требует 7 цифр вместо 2 в десятичной, $\sigma = \log 10/\log 2 = 3{,}322$; следовательно, количество цифр в десятичном представлении нужно умножить на 3,322 и округлить в большую сторону: $2 \cdot 3{,}322 = 6{,}644 \approx 7$.

Введем понятие *экономичности представления числа* в данной системе счисления [12].

Определение

Под *экономичностью* системы счисления будем понимать то количество чисел, которое можно записать в данной системе с помощью определенного количества цифр.

Речь в данном случае идет не о количестве разрядов, а об общем количестве сочетаний цифр, которые интерпретируются как различные числа. Поясним на примере: пусть в распоряжении имеется 12 цифр. Можно разбить их на 6 групп по 2 цифры ("0" и "1") и получить шестиразрядное двоичное число; общее количество таких чисел, как уже неоднократно обсуждалось, равно 2^6. Можно разбить заданное количество цифр на 4 группы по три цифры и воспользоваться троичной системой счисления — в этом случае общее количество различных их сочетаний составит 3^4. Аналогично можно произвести другие разбиения; при этом число групп определит разрядность числа, а количество цифр в группе — основание системы счисления. Результаты различных разбиений можно проиллюстрировать табл. 3.2.

Из приведенных оценок видно, что наиболее экономичной оказывается *троичная система счисления*, причем результат будет тем же, если исследовать случаи с другим исходным количеством сочетаний цифр.

Точное расположение максимума экономичности может быть установлено путем следующих рассуждений. Пусть имеется n знаков для записи чисел, а

Таблица 3.2. Результаты разбиения цифр на группы

Параметр	Значения				
Основание системы счисления (p)	2	3	4	6	12
Разрядность числа (k)	6	4	3	2	1
Общее количество различных чисел (N)	$2^6 = 64$	$3^4 = 81$	$4^3 = 64$	$6^2 = 36$	$12^1 = 12$

основание системы числения равно p. Тогда количество разрядов числа $k = n/p$, а общее количество чисел N, которые могут быть составлены, равно:

$$N = p^{\frac{n}{p}}. \tag{3.10}$$

Если считать $N(p)$ *непрерывной функцией*, то можно найти такое значение p_m, при котором N принимает максимальное значение. Для нахождения положения максимума нужно найти производную функции $N(p)$, приравнять ее к нулю и решить полученное уравнение относительно p.

$$\frac{dN}{dp} = -\frac{n}{p^2} \cdot p^{\frac{n}{p}} \cdot \ln p + \frac{n}{p} \cdot p^{\frac{n}{p}-1} = n \cdot p^{\frac{n}{p}-2} \cdot (1 - \ln p). \tag{3.11}$$

Приравнивая полученное выражение к нулю, получаем $\ln p = 1$, или $p_m = e$, где $e = 2{,}71828...$ — основание натурального логарифма. Ближайшее к e целое число, очевидно, 3 — по этой причине троичная система счисления оказывается самой экономичной для представления чисел, однако следующей по экономичности оказывается двоичная система счисления.

Таким образом, простота технических решений — не единственный аргумент в пользу применения двоичной системы в компьютерах.

3.3. Представление информации в ЭВМ. Прямой код

В современных ЭВМ используются, в основном, два способа представления двоичных чисел — с фиксированной и с плавающей запятой, причем в формате с фиксированной запятой (ФЗ) используется как *беззнаковое* представ-

ление чисел ("целое без знака"), так и представление чисел со знаком. В последнем случае знак также кодируется двоичной цифрой — обычно плюсу соответствует 0, а минусу — 1. Под код знака обычно отводится старший разряд a_0 двоичного вектора $a_0a_1a_2...a_n$, называемый *знаковым*.

Запятая может быть фиксирована после любого разряда двоичного числа, однако чаще всего используются два формата ФЗ: *целые числа*, когда запятая фиксируется после младшего разряда a_n, а диапазон представления лежит в пределах

$$|A| \le 2^n - 1, \tag{3.12}$$

и *дробные числа* — запятая фиксирована после a_0, а диапазон

$$|A| \le 1 - 2^{-n}. \tag{3.13}$$

Далее, если не сделано специальных оговорок, будем рассматривать дробные двоичные *числа со знаком*, запятая в которых фиксирована после знакового разряда a_0:

$$a_0, a_1a_2a_3 \ldots a_n. \tag{3.14}$$

Очевидно, если двоичное число $A = 0, a_1a_2a_3 \ldots a_n > 0$, то оно будет представлено в форме (3.14) как $0, a_1a_2a_3 \ldots a_n$, а если $A = 0, a_1a_2a_3 \ldots a_n < 0$, то как $1, a_1a_2a_3 \ldots a_n$. Приведенное кодирование дробных двоичных чисел со знаком принято называть *прямым кодом числа* (обозначается как $[A]_д$). Итак

$$[A]_д = \begin{cases} A, & \text{если } A \ge 0; \\ 1 + |A|, & \text{если } A < 0. \end{cases} \tag{3.15}$$

3.4. Алгебраическое сложение/вычитание в прямом коде

Сформулируем правила выполнения операций сложения и вычитания чисел со знаками (такие операции принято называть *алгебраическими*). Во-первых, алгебраическое вычитание всегда можно свести к алгебраическому сложению, изменив знак второго операнда. Далее следует сравнить знаки слагаемых. *При одинаковых знаках складывают модули слагаемых и результату присваивают* знак любого слагаемого (они одинаковые). Если знаки слагаемых разные, то из большего модуля слагаемого *вычитают* меньший модуль и *присваивают* результату знак слагаемого, имеющего больший модуль.

Введем обозначения:

$$A = a_0 a_1 a_2 a_3 ... a_n,$$

$$B = b_0 b_1 b_2 b_3 ... b_n,$$

$$C = A + B = c_0 c_1 c_2 .. c_n,$$

где:

- a_0, b_0 — знаковые разряды слагаемых;
- c_0 — код знака результата;
- a_i, b_i, c_i, $i \in \{0, 1, 2, ..., n\}$ — двоичные переменные;
- f — тип выполняемой операции: $f = 0$ — сложение, $f = 1$ — вычитание;
- OV — признак переполнения,

и выразим сформулированный выше алгоритм алгебраического сложения/вычитания в форме *граф-схемы алгоритма* (ГСА), приведенной на рис. 3.3.

Отдельно следует рассмотреть проблему обнаружения факта переполнения разрядной сетки данных с фиксированной запятой. Это может произойти, если

$$|C| = |A| + |B| \geq 1. \qquad (3.16)$$

Очевидно, при сложении чисел с разными знаками переполнение невозможно. Если знаки слагаемых одинаковы, признаком переполнения может служить перенос, возникающий при сложении старших разрядов модулей $a_1 + b_1$. При отсутствии этого переноса сложение двух любых *одинаковых* знаковых разрядов даст в результате $c_0 = 0$, а при появлении переноса из первого разряда $c_0 = 1$. Таким образом, после сложения чисел с одинаковыми знаками значение знакового разряда суммы можно рассматривать как признак переполнения OV.

Характерно, что полученное в знаковом разряде c_0 значение не является знаком результата (алгебраической суммы). Истинное значение знака образуется не в процессе арифметической операции над знаковыми разрядами, а формируется искусственно.

Рассмотрим случай сложения чисел с *разными знаками*. Он сводится к вычитанию модулей слагаемых, причем уменьшаемым должен стать больший модуль. Чтобы избежать дополнительной модульной операции сравнения, можно произвести "наугад" вычитание $A - B$. Признаком того, что $|A| > |B|$ будет отсутствие заема из нулевого в первый разряд. Поскольку рассматривается

случай разных знаков слагаемых, то при отсутствии заема значение знакового разряда разности определится как $0-1=1-0=1$, а при наличии заема $0-1-1=1-0-1=0$. Таким образом, если при вычитании $A-B$ получим $c_0=1$, это будет означать, что $|A|>|B|$, и результату следует присвоить знак числа A ($c_0 := a_0$). Если окажется $c_0=0$, то $|A|<|B|$, и следует осуществить вычитание $B-A$, присвоив результату знак числа B ($c_0 := b_0$).

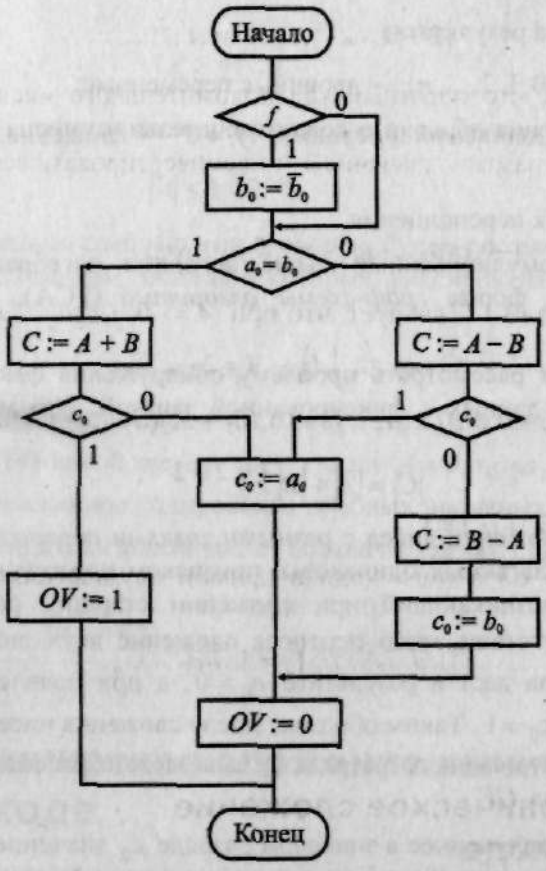


Рис. 3.3. Граф алгоритма алгебраического сложения-вычитания

3.5. Обратный код и выполнение алгебраического сложения в нем

При выполнении алгебраического сложения в прямом коде приходится, во-первых, не только складывать, но и вычитать двоичные коды; во-вторых, код

знака результата формируется искусственно, т. е. знаковые разряды обрабатываются по правилам, отличным от правил обработки разрядов числа. Для устранения отмеченных недостатков в ЭВМ широко используются специальные представления двоичных чисел — т. н. *обратный* и *дополнительный коды*.

Представление *обратного кода* определяется следующим соотношением:

$$[A]_i = \begin{cases} A, & \text{если } A \geq 0; \\ 2 + A - 2^{-n}, & \text{если } A \leq 0. \end{cases} \qquad (3.17)$$

Из (3.17) следует, что обратный код положительного числа равен самому числу! Для получения обратного кода отрицательного числа достаточно присвоить знаковому разряду значение 1 и проинвертировать все остальные разряды числа:

$$[-0, a_1 a_2 \ldots a_n]_i = 1, \overline{a}_1 \overline{a}_2 \ldots \overline{a}_n. \qquad (3.18)$$

Действительно, из (3.17) следует, что при $A = -0, a_1 a_2 a_3 \ldots a_n$ обратный код числа $[A]_i = 2 + A - 2^{-n}$. Откуда $[A]_i - A = 2 - 2^{-n}$.

$$1, \overline{a}_1 \overline{a}_2 \ldots \overline{a}_n + 0, a_1 a_2 \ldots a_n = (1+0), (\overline{a}_1 + a_1)(\overline{a}_2 + a_2) \ldots (\overline{a}_n + a_n) =$$
$$= 1, 11 \ldots 1 = 2 - 2^{-n},$$

учитывая, что $(\overline{a}_i + a_i) = 1$.

Для перехода из обратного кода в прямой осуществляется следующее преобразование:

$$[1, \overline{a}_1 \overline{a}_2 \ldots \overline{a}_n]_i = 1, \overline{\overline{a}}_1 \overline{\overline{a}}_2 \ldots \overline{\overline{a}}_n,$$

т. е. $[[A]_i]_i = [A]_d$.

3.5.1. Алгебраическое сложение в обратном коде

Очевидно, что при отсутствии переполнения возможны четыре случая сочетания знаков и модулей слагаемых [11].

❐ Случай 1.

$$A > 0,\ B > 0,\ A + B < 1.$$

Этот случай соответствует обычному сложению прямых кодов чисел: $[A > 0]_i + [B > 0]_i = A + B$.

- Случай 2.

$$A>0,\ B<0,\ A+B>0.$$

$[A>0]_{\text{д}}+[B<0]_{\text{д}}=A+2+B-2^{-n}$. Назовем этот результат предварительным. Истинное значение результата в рассматриваемом случае (сумма положительна) будет $A+B$. Следовательно, предварительный результат нуждается в *коррекции* путем вычитания 2 и добавления 2^{-n}.

- Случай 3.

$$A>0,\ B<0,\ A+B<0.$$

$[A>0]_{\text{д}}+[B<0]_{\text{д}}=A+2+B-2^{-n}$. Этот результат соответствует правильному, поскольку рассматривается случай отрицательной суммы.

- Случай 4.

$$A<0,\ B<0,\ |A+B|<1.$$

$[A<0]_{\text{д}}+[B<0]_{\text{д}}=2+A-2^{-n}+2+B-2^{-n}$. Здесь предварительный результат нуждается в *коррекции* путем вычитания 2 и добавления 2^{-n}, как и в случае 2, поскольку истинное значение отрицательной суммы, представленной в обратном коде, $A+B+2-2^{-n}$.

Заметим, что в случаях 2 и 4 требуется одинаковая коррекция: $-2+2^{-n}$, причем только в этих двух случаях возникает перенос из знакового разряда. Действительно, в случае 4 оба знаковых разряда равны 1, а в случае 2 знак результата — 0, что при разных знаках слагаемых может получиться только при появлении переноса из первого разряда в нулевой (знаковый), а следовательно, обязательно будет перенос и из знакового разряда. Вес знакового разряда соответствует 2^0, а вес переноса из него — 2^1. Таким образом, игнорируя перенос из знакового разряда, мы *вычитаем из результата* 2, что соответствует первому члену корректирующего выражения. Для учета второго члена следует добавить 1 к младшему разряду суммы, вес которого составляет 2^{-n}.

В случаях 1 и 3 переноса из знакового разряда не возникает и коррекция результата не требуется.

Таким образом, для выполнения алгебраического сложения двоичных чисел, представленных в обратном коде, достаточно, не анализируя соотношение знаков и модулей, произвести сложение чисел, включая знаковые разряды, по правилам двоичной арифметики, причем возникающий в знаковом разряде перенос должен быть добавлен к младшему разряду результата, осуществляя тем самым коррекцию предварительной суммы. Полученный код является алгебраической суммой слагаемых, представленной в обратном коде.

Рассмотрим несколько примеров.

Пример 3.5

Сложить два числа в обратном коде. Результат — на рис. 3.4.

$$
\begin{array}{lll}
A = +0{,}1101 & [A]_d = 0{,}1101 & [A]_i = 0{,}1101 \\
B = -0{,}0011 & [B]_d = 1{,}0011 & [B]_i = 1{,}1100 \\
 & & \hline
 & & \mathbf{1} \leftarrow 0{,}1001 \\
 & & \hline
C = 0{,}1010 \Leftarrow & [C]_d = 0{,}1010 \Leftarrow & [C]_i = 0{,}1010
\end{array}
$$

Рис. 3.4. Результат выполнения примера 3.5

Приведенный пример соответствует рассмотренному выше случаю 2; перенос, возникающий в знаковом разряде, циклически передается в младший разряд предварительного результата.

Пример 3.6

Сложить два числа в обратном коде (случай 3). Результат — на рис. 3.5.

$$
\begin{array}{lll}
A = -0{,}1101 & [A]_d = 1{,}1101 & [A]_i = 1{,}0010 \\
B = +0{,}0011 & [B]_d = 0{,}0011 & [B]_i = 0{,}0011 \\
 & & \hline
C = -0{,}1010 \Leftarrow & [C]_d = 1{,}1010 \Leftarrow & [C]_i = 1{,}0101
\end{array}
$$

Рис. 3.5. Результат выполнения примера 3.6

Пример 3.7

Сложить два числа в обратном коде (случай 4). Результат — на рис. 3.6.

$$
\begin{array}{lll}
A = -0{,}0101 & [A]_d = 1{,}0101 & [A]_i = 1{,}1010 \\
B = -0{,}0110 & [B]_d = 1{,}0110 & [B]_i = 1{,}1001 \\
 & & \hline
 & & \mathbf{1} \leftarrow 1{,}0011 \\
 & & \hline
C = -0{,}1011 \Leftarrow & [C]_d = 1{,}1011 \Leftarrow & [C]_i = 1{,}0100
\end{array}
$$

Рис. 3.6. Результат выполнения примера 3.7

Пример 3.8

Сложить два числа в обратном коде (одинаковые модули, но разные знаки). Результат — на рис. 3.7.

$$\begin{array}{lll} A=-0{,}0101 & [A]_{об}=1{,}0101 & [A]_{об}=1{,}1010 \\ B=+0{,}0101 & [B]_{об}=0{,}0101 & [B]_{об}=0{,}0101 \\ C=-0{,}0000 \Leftarrow & [C]_{об}=1{,}0000 \Leftarrow & [C]_{об}=1{,}1111 \end{array}$$

Рис. 3.7. Результат выполнения примера 3.8

Таким образом, ноль в обратном коде бывает "положительный" и "отрицательный" (обратите внимание на выражение (3.17)), причем добавление к числу "отрицательного" нуля, как и "положительного", дает в результате значение первого слагаемого.

Пример 3.9

Сложить два числа в обратном коде: $3+(-0)$. Результат — на рис. 3.8.

$$\begin{array}{lll} A=+0{,}0011 & [A]_{об}=0{,}0011 & [A]_{об}=0{,}0011 \\ B=-0{,}0000 & [B]_{об}=1{,}0000 & [B]_{об}=1{,}1111 \\ & & \overline{\mathbf{1} \leftarrow 0{,}0010} \\ & & \phantom{\mathbf{1} \leftarrow} \quad 1 \\ C=+0{,}0011 \Leftarrow & [C]_{об}=0{,}0011 \Leftarrow & [C]_{об}=0{,}0011 \end{array}$$

Рис. 3.8. Результат выполнения примера 3.9

Теперь рассмотрим случаи, когда $|A+B| \geq 1|$, что соответствует переполнению разрядной сетки. Очевидно, учитывая, что $|A|<1$ и $|B|<1$, переполнение возможно только при сложении чисел с одинаковыми знаками. Рассмотрим примеры.

Пример 3.10

Сложить два числа в обратном коде: $13/16+5/16=18/16$. Результат — на рис. 3.9.

$$\begin{array}{lll} A=+0{,}1101 & [A]_d=0{,}1101 & [A]_i=0{,}1101 \\ B=+0{,}0101 & [B]_d=0{,}0101 & [B]_i=0{,}0101 \\ C=-0{,}1101 \Leftarrow & [C]_d=1{,}1101 \Leftarrow & [C]_i=1{,}0010 \end{array}$$

Рис. 3.9. Результат выполнения примера 3.10

Пример 3.11

Сложить два числа в обратном коде: $(-11/16)+(-8/16)=(-19/16)$. Результат — на рис. 3.10.

$$\begin{array}{lll} A=-0{,}1011 & [A]_d=1{,}1011 & [A]_i=1{,}0100 \\ B=-0{,}1000 & [B]_d=1{,}1000 & [B]_i=1{,}0111 \\ & & \overline{1 \leftarrow 0{,}1011} \\ & & \phantom{1 \leftarrow 0{,}}1 \\ C=+0{,}1100 \Leftarrow & [C]_d=0{,}1100 \Leftarrow & [C]_i=0{,}1100 \end{array}$$

Рис. 3.10. Результат выполнения примера 3.11

Таким образом, признаком переполнения в обратном коде можно считать *знак результата, противоположный одинаковым знакам слагаемых*:

$$OV=\bar{a}_0\bar{b}_0c_0 \vee a_0b_0\bar{c}_0 . \qquad (3.19)$$

Пример 3.12

Сложить числа в обратном коде ($A>B$, $B>0$, $|A+B|=1$). Результат — на рис. 3.11.

$$\begin{array}{lll} A=+0{,}0111 & [A]_d=0{,}0111 & [A]_i=0{,}0111 \\ B=+0{,}1001 & [B]_d=0{,}1001 & [B]_i=0{,}1001 \\ C=-0{,}1111 \Leftarrow & [C]_d=1{,}1111 \Leftarrow & [C]_i=1{,}0000 \end{array}$$

Рис. 3.11. Результат выполнения примера 3.12

Пример 3.13

Сложить числа в обратном коде ($A<0$, $B<0$, $|A+B|=1$). Результат — на рис. 3.12.

$$A = -0{,}0111 \qquad [A]_д = 1{,}0111 \qquad [A]_і = 1{,}1000$$
$$B = -0{,}1001 \qquad [B]_д = 1{,}1001 \qquad [B]_і = 1{,}0110$$
$$\overline{\mathbf{1} \leftarrow 0{,}1110}$$
$$\overline{\phantom{\mathbf{1} \leftarrow 0{,}1110}\ 1}$$
$$C = +0{,}1111 \Leftarrow [C]_д = 0{,}1111 \Leftarrow [C]_і = 0{,}1111$$

Рис. 3.12. Результат выполнения примера 3.13

Переполнение в соответствии с (3.19) обнаруживается и в этих случаях.

Итак, использование обратного кода в операциях алгебраического сложения/вычитания позволяет:

- использовать только действие арифметического сложения двоичных кодов;
- получать истинное значение знака результата, выполняя над знаковыми разрядами операндов те же действия, что и над разрядами чисел;
- обнаруживать переполнение разрядной сетки.

Еще одним достоинством применения обратного кода можно считать простоту взаимного преобразования прямого и обратного кода.

Однако использование обратного кода имеет один существенный недостаток — коррекция предварительной суммы требует добавления единицы к ее младшему разряду и может вызвать (в некоторых случаях) распространение переноса по всему числу, что, в свою очередь, приводит к увеличению вдвое времени суммирования. Для преодоления этого недостатка можно использовать вместо обратного *дополнительный код*.

3.6. Дополнительный код и арифметические операции в нем

Связь между числом и его изображением в дополнительном коде определяется соотношениями

$$[A]_c = \begin{cases} A, & \text{если } A \geq 0; \\ 2 + A, & \text{если } A < 0. \end{cases} \qquad (3.20)$$

Таким образом, и дополнительный код положительного числа равен самому числу (как обратный и прямой). Дополнительный код отрицательного числа *дополняет* исходное число до основания системы счисления.

Дополнительный код отрицательного числа образуется в соответствии со следующим выражением:

$$[-0,a_1a_2\ldots a_n]_c = 1,\overline{a}_1\overline{a}_2\ldots\overline{a}_n + 2^{-n} = [-0,a_1a_2\ldots a_n]_i + 2^{-n}. \qquad (3.21)$$

Действительно, из (3.20) следует, что для *отрицательного* числа $A = -0,a_1a_2a_3\ldots a_n$ дополнительный код $[A]_c = 2 + A$, откуда $[A]_c - A = 2$ или $[A]_c + A = 2$. Тогда

$$1,\overline{a}_1\overline{a}_2\ldots\overline{a}_n + 2^{-n} + 0,a_1a_2\ldots a_n = (1+0),(\overline{a}_1 + a_1)(\overline{a}_2 + a_2)\ldots(\overline{a}_n + a_n + 2^{-n}) = \\ = 10,00\ldots0 = 2,$$

учитывая, что $(\overline{a}_i + a_i) = 1$.

Таким образом, для преобразования отрицательного двоичного числа в дополнительный код следует преобразовать его сначала в обратный код (установив знаковый разряд в 1 и проинвертировав все остальные разряды числа) и добавить единицу к младшему разряду обратного кода.

Другой способ перевода прямого кода отрицательного двоичного числа в дополнительный (приводящий, разумеется, к такому же результату) определяется следующим правилом: оставить без изменения все младшие нули и одну младшую единицу, остальные разряды (кроме знакового!) проинвертировать.

Пример 3.14

Преобразовать числа в дополнительный код. Результат — на рис. 3.13.

Число		Прямой код		Обратный код		Дополнительный код
$+0,0111$	$\Rightarrow$	$[A]_d = 0,0111$	$\Rightarrow$	$[A]_i = 0,0111$	$\Rightarrow$	$[A]_c = 0,0111$
$-0,0111$	$\Rightarrow$	$[A]_d = 1,0111$	$\Rightarrow$	$[A]_i = 1,1000$	$\Rightarrow$	$[A]_c = 1,1001$
$-0,1000$	$\Rightarrow$	$[A]_d = 1,1000$	$\Rightarrow$	$[A]_i = 1,0111$	$\Rightarrow$	$[A]_c = 1,1000$
$-0,0101$	$\Rightarrow$	$[A]_d = 1,0101$	$\Rightarrow$	$[A]_i = 1,1010$	$\Rightarrow$	$[A]_c = 1,1011$

Рис. 3.13. Результат выполнения примера 3.14

3.6.1. Алгебраическое сложение в дополнительном коде

Рассмотрим те же четыре случая сочетания знаков и модулей операндов, что и при рассмотрении сложения в обратном коде в *разд. 3.5.1*:

- Случай 1.

$$A>0,\ B>0,\ A+B<1.$$

Этот случай соответствует обычному сложению прямых кодов чисел: $[A>0]_c+[B>0]_c=A+B$.

- Случай 2.

$$A>0,\ B<0,\ A+B>0.$$

$[A>0]_c+[B<0]_c=A+2+B$. Истинное значение результата в рассматриваемом случае (сумма положительна) будет $A+B$ и коррекция заключается в вычитании 2.

- Случай 3.

$$A>0,\ B<0,\ A+B<0.$$

$[A>0]_c+[B<0]_c=A+2+B$. Этот результат соответствует правильному, поскольку рассматривается случай отрицательной суммы.

- Случай 4.

$$A<0,\ B<0,\ |A+B|<1.$$

$[A<0]_c+[B<0]_c=2+A+2+B$. Здесь предварительный результат, как и в случае 2°, нуждается в *коррекции* путем вычитания 2, поскольку истинное значение отрицательной суммы, представленной в дополнительном коде $A+B+2$.

Как и в обратном коде, коррекция требуется только в случаях 2 и 4, причем в дополнительном коде коррекция заключается просто в игнорировании переноса, возникающего из знакового разряда.

Рассмотрим несколько примеров.

Пример 3.15

Сложить два числа в дополнительном коде: $(+13/16)+(-3/16)=(+10/16)$. Результат — на рис. 3.14.

$A=+0{,}1101$		$[A]_d=0{,}1101$		$[A]_c=0{,}1101$
$B=-0{,}0011$		$[B]_d=1{,}0011$		$[B]_c=1{,}1101$
$C=+0{,}1010$	$\Leftarrow$	$[C]_d=0{,}1010$	$\Leftarrow$	$[C]_c=+1{,}1010$

Рис. 3.14. Результат выполнения примера 3.15

Пример 3.16

Сложить два числа в дополнительном коде (случай 3). Результат — на рис. 3.15.

$$\begin{array}{lll} A=-0{,}1101 & [A]_d=1{,}1101 & [A]_c=1{,}0011 \\ B=+0{,}0011 & [B]_d=0{,}0011 & [B]_c=0{,}0011 \\ C=-0{,}1010 \Leftarrow & [C]_d=1{,}1010 \Leftarrow & [C]_c=1{,}0110 \end{array}$$

Рис. 3.15. Результат выполнения примера 3.16

Пример 3.17

Сложить два числа в дополнительном коде (случай 4). Результат — на рис. 3.16.

$$\begin{array}{lll} A=-0{,}0101 & [A]_d=1{,}0101 & [A]_c=1{,}1011 \\ B=-0{,}0110 & [B]_d=1{,}0110 & [B]_c=1{,}1010 \\ C=-0{,}1011 \Leftarrow & [C]_d=1{,}1011 \Leftarrow & [C]_c=\text{!}1{,}0101 \end{array}$$

Рис. 3.16. Результат выполнения примера 3.17

Пример 3.18

Сложить два числа в дополнительном коде (одинаковые модули, но разные знаки). Результат — на рис. 3.17.

$$\begin{array}{lll} A=-0{,}0101 & [A]_d=1{,}0101 & [A]_c=1{,}1011 \\ B=+0{,}0101 & [B]_d=0{,}0101 & [B]_c=0{,}0101 \\ C=+0{,}0000 \Leftarrow & [C]_d=0{,}0000 \Leftarrow & [C]_c=\text{!}0{,}0000 \end{array}$$

Рис. 3.17. Результат выполнения примера 3.18

Из примера 3.18 видно, что "ноль" в дополнительном коде имеет единственное "положительное" представление.

Теперь рассмотрим случаи, когда $|A+B|>1$, что соответствует переполнению разрядной сетки.

Пример 3.19

Сложить два числа в дополнительном коде: $13/16 + 5/16 = 18/16$. Результат — на рис. 3.18.

$$\begin{array}{lll} A = +0{,}1101 & [A]_д = 0{,}1101 & [A]_с = 0{,}1101 \\ B = +0{,}0101 & [B]_д = 1{,}0101 & [B]_с = 0{,}0101 \\ C = -0{,}1110 \Leftarrow & [C]_д = 1{,}1110 \Leftarrow & [C]_с = 1{,}0010 \end{array}$$

Рис. 3.18. Результат выполнения примера 3.19

Пример 3.20

Сложить два числа в дополнительном коде: $(-11/16) + (-8/16) = (-19/16)$. Результат — на рис. 3.19.

$$\begin{array}{lll} A = -0{,}1011 & [A]_д = 1{,}1011 & [A]_с = 1{,}0101 \\ B = -0{,}0011 & [B]_д = 1{,}1000 & [B]_с = 1{,}1000 \\ C = +0{,}1101 \Leftarrow & [C]_д = 0{,}1101 \Leftarrow & [C]_с = +0{,}1101 \end{array}$$

Рис. 3.19. Результат выполнения примера 3.20

Очевидно, для дополнительного кода, как и для обратного, справедливо выражение (3.19). Теперь рассмотрим случаи $|A + B| = 1$. Для положительных слагаемых пример 3.12 может относиться как к обратным, так и к дополнительным кодам, но преобразование результата — дополнительного кода в прямой приведет к другому значению. Действительно,

$$[C]_с = 1{,}0000 \rightarrow [[C]_с]_д + 1 = 1{,}1111 + 1 = 1{,}0000 = -0{,}0000.$$

Для случая $A < 0$, $B < 0$, $|A + B| = 1$ имеем следующее.

Пример 3.21

Сложить два числа в дополнительном коде: $(-11/16) + (-5/16) = (-16/16)$. Результат — на рис. 3.20.

Переполнение по признакам выражения (3.19) не обнаружено! Однако результат операции — "отрицательный ноль", который не может использоваться

$$A = -0{,}1011 \qquad [A]_д = 1{,}1011 \qquad [A]_с = 1{,}0101$$
$$B = -0{,}0101 \qquad [B]_д = 1{,}0101 \qquad [B]_с = 1{,}1011$$
$$C = -0{,}0000 \Leftarrow [C]_д = 1{,}0000 \Leftarrow [C]_с = \overline{1}0{,}0000$$

Рис. 3.20. Результат выполнения примера 3.21

в дополнительном коде. Действительно, сложение в дополнительном коде любого числа с "отрицательным нулем" 1,00...0 меняет знак этого числа. Итак, *при* $A<0$, $B<0$, $|A+B|=1$ *признаком переполнения служит не выражение* (3.19), а код результата 1, 00...0.

Таким образом, значение признака переполнения в дополнительном коде можно получить в соответствии со следующим выражением:

$$OV = \overline{a}_0\overline{b}_0c_0 \vee a_0b_0\overline{c}_0 \vee c_0\overline{c}_1\overline{c}_2 \ldots \overline{c}_n. \qquad (3.22)$$

Подведем итоги. Применение дополнительного кода, по сравнению с обратным, имеет одно существенное преимущество — коррекция результата сводится просто к отбрасыванию переноса из знакового разряда и *не требует дополнительных затрат времени*. К недостаткам применения дополнительного кода можно отнести, во-первых, более сложную процедуру взаимного преобразования ПК ↔ ДК, требующую дополнительных затрат времени, и, во-вторых, проблемы с обнаружением переполнения. Для того чтобы минимизировать влияние первого недостатка, данные в памяти часто хранят в дополнительном коде. В этом случае преобразования ПК ↔ ДК выполняются относительно редко — только при вводе и выводе.

3.6.2. Модифицированные обратный и дополнительный коды

Для определения переполнения используют выражение (3.19) — булеву функцию трех переменных. С целью более удобного обнаружения переполнения в обратном и дополнительном кодах можно применить т. н. "модифицированные" их представления:

$$[A]_i^m = \begin{cases} A, & \text{если } A \geq 0; \\ 4 + A - 2^{-n}, & \text{если } A \leq 0. \end{cases} \qquad (3.23)$$

$$[A]_c^m = \begin{cases} A, & \text{если } A \geq 0; \\ 4 + A, & \text{если } A < 0. \end{cases} \qquad (3.24)$$

Нетрудно показать, что модифицированные коды отличаются от соответствующих обычных наличием дополнительного знакового разряда: "плюс" кодируется 00, а "минус" — 11. Эта своеобразная избыточность, сохраняя все качества обычных обратных и дополнительных кодов, позволяет фиксировать факт переполнения по *неравнозначности знаковых разрядов результата*. Заметим, что использование модифицированного дополнительного кода не решает проблемы обнаружения переполнения в случаях $A<0$, $B<0$, $|A+B|=1$.

3.7. Алгоритмы алгебраического сложения в обратном и дополнительном коде

В *разд. 3.5.1* и *3.6* подробно обсуждалось, как выполнить операцию алгебраического сложения чисел, уже представленных соответственно в обратном или дополнительном коде. Для этого достаточно выполнить арифметическое сложение двоичных векторов, получив истинное значение результата в коде представления операндов. При операции в обратном коде возникающий из знакового разряда перенос следует добавить к младшему разряду суммы. Переполнение обнаруживается согласно выражению (3.19).

В случае если слагаемые представлены в прямом коде, а операция выполняется в обратном или дополнительном, их следует сначала преобразовать в соответствующий код, затем выполнить сложение и сумму вновь преобразовать в прямой код — код результата всегда должен соответствовать коду исходных данных. На рис. 3.21 приведен пример алгоритма алгебраического сложения в *обратном коде* чисел, представленных в *прямом коде*, а на рис. 3.22 — алгебраическое сложение/вычитание чисел в *дополнительном коде*.

При рассмотрении алгоритмов использованы те же обозначения, которые были введены в *разд. 3.4* для рис. 3.1. Дополнительно введем обозначения:

- $A'=a_1a_2...a_n$, $B'=b_1b_2...b_n$, $C'=c_1c_2...c_n$ — модули чисел;
- c_{-1} — перенос из знакового разряда;
- $\alpha^*=\overline{a}_0\overline{b}_0c_0 \vee a_0b_0\overline{c}_0 \vee c_0\overline{c}_1\overline{c}_2 \ldots \overline{c}_n$ — ситуации переполнения в дополнительном коде.

В алгоритме рис. 3.22 можно отметить один недостаток. При выполнении вычитания ($f=1$) необходимо получить дополнение второго операнда: $B' \coloneqq B'+1$, что является *арифметической операцией* и требует времени, достаточного для прохождения переноса по всем разрядам числа. Для исключения дополнительной арифметической операции можно в первой операторной

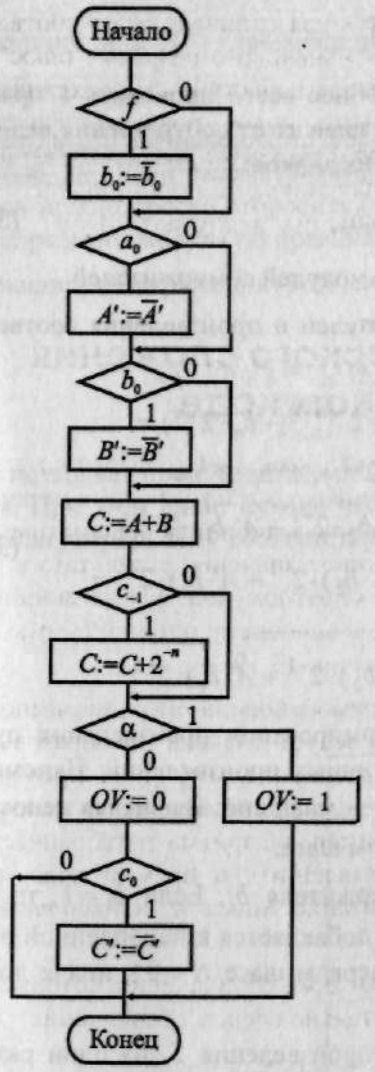


Рис. 3.21. Алгоритм алгебраического сложения в обратном коде

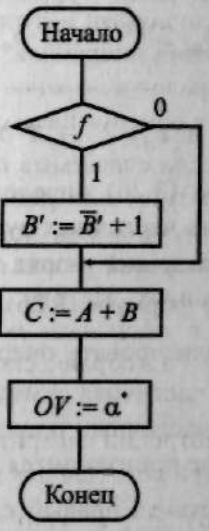


Рис. 3.22. Алгоритм алгебраического сложения/вычитания в дополнительном коде

вершине осуществить только инверсию (*логическую операцию*, которая выполняется быстро), а недостающую "единицу" к младшему разряду добавить, если это необходимо, в качестве входного переноса младшего разряда в момент суммирования слагаемых. Таким образом, в двух первых *операторных* вершинах алгоритма рис. 3.22 следует поместить такие операторы:

$$B' := \overline{B}';$$

$$C := A + B + f.$$

3.8. Алгоритмы умножения

Умножение двоичных чисел со знаком удобнее всего проводить в прямом коде. Действительно, знак произведения не зависит от соотношения величин модулей сомножителей, а зависит только от их знаков:

$$c_0 = a_0 \oplus b_0 = a_0\overline{b}_0 \vee \overline{a}_0 b_0, \tag{3.25}$$

а модуль произведения равен произведению модулей сомножителей.

Обозначим A', B', C' — модули сомножителей и произведения соответственно. Тогда

$$\begin{aligned}
C' &= A' \cdot B' = A' \cdot (b_1 \cdot 2^{-1} + b_2 \cdot 2^{-2} + \ldots + b_{n-1} \cdot 2^{-(n-1)} + b_n \cdot 2^{-n}) = \\
&= A' \cdot b_n \cdot 2^{-n} + A' \cdot b_{n-1} \cdot 2^{-(n-1)} + \ldots + A' \cdot b_2 \cdot 2^{-2} + A' \cdot b_1 \cdot 2^{-1} = \\
&= (A' \cdot b_n \cdot 2^{-(n-1)} + A' \cdot b_{n-1} \cdot 2^{-(n-2)} + \ldots + A' \cdot b_2 \cdot 2^{-1} + A' \cdot b_1) \cdot 2^{-1} = \\
&= ((A' \cdot b_n \cdot 2^{-(n-2)} + A' \cdot b_{n-1} \cdot 2^{-(n-3)} + \ldots + A' \cdot b_2) \cdot 2^{-1} + A' \cdot b_1) \cdot 2^{-1} = \\
&\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots \\
&(\ldots(0 + A' \cdot b_n) \cdot 2^{-1} + A' \cdot b_{n-1}) \cdot 2^{-1} + \ldots + A' \cdot b_2) \cdot 2^{-1} + A' \cdot b_1) \cdot 2^{-1}.
\end{aligned} \tag{3.26}$$

Выражение (3.26) определяет процесс формирования произведения путем вычисления частичных сумм и суммы частичных произведений. Напомним, что b_1 — старший разряд множителя, а b_n — младший. Вычисляя непосредственно по формуле (3.26), следует на каждом шаге:

1. Проанализировать очередную цифру множителя b_i. Если $b_i = 1$, то очередная частичная сумма равна A, и она добавляется к накопленной ранее сумме частичных произведений S (на первом шаге $S = 0$), иначе добавления не производится.
2. Осуществить правый сдвиг частичного произведения S на один разряд. Умножение $S \cdot 2^{-1}$ соответствует делению на 2, что в двоичной системе счисления равносильно *сдвигу числа на один разряд вправо*.
3. Пункты 1 и 2 повторяются до тех пор, пока не будут исчерпаны все цифры множителя.

Очевидно, число шагов при использовании приведенного выше метода равно разрядности модуля множителя. Алгоритм умножения чисел, представленных в прямом коде, приведен на рис. 3.23.

В изображенном на рис. 3.23 алгоритме, в отличие от алгоритмов сложения/вычитания, значение $OV = 0$ устанавливается безусловно. Дейст-

вительно, A и B — дробные числа; очевидно при $|A|<1$ и $|B|<1$ всегда $|A\times B|<1$.

В результате вычисления по формуле (3.26) получается произведение разрядностью $2n$. Если рассматривать сомножители как дроби, то младшие n разрядов можно просто отбросить (округление с недостатком) или округлить до n-разрядного модуля по правилам округления.

Очевидно, из выражения (3.26) легко получить

$$C' = A' \cdot B' = \\ = (\ldots(0 + 2^{-n} \cdot A' \cdot b_1) \cdot 2 + 2^{-n} \cdot A' \cdot b_2) \cdot 2 + \ldots + \\ + 2^{-n} \cdot A' \cdot b_{n-1}) \cdot 2 + 2^{-n} \cdot A' \cdot b_n) \cdot 2, \qquad (3.27)$$

что позволяет производить умножение, начиная со старших разрядов множителя. При этом сдвиг суммы частичных произведений осуществляется *влево на один разряд*, чему соответствует умножение двоичного числа на 2.

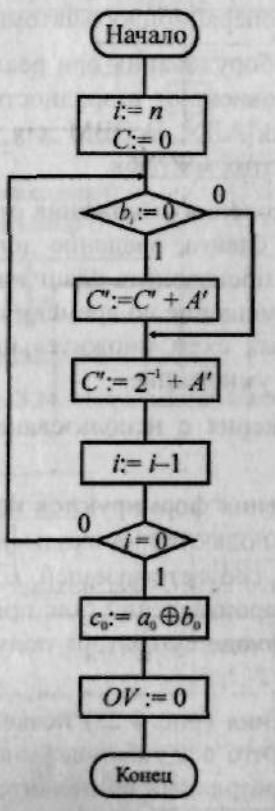


Рис. 3.23. Умножение в прямом коде

3.8.1. Умножение в дополнительном коде

Если числа поступают на обработку уже представленные в дополнительном коде, то для умножения их можно перевести в прямой код или умножать сразу в дополнительном коде. В последнем случае в умножении участвуют и знаковые разряды сомножителей, причем знак произведения получается в том же цикле, что и разряды модуля произведения. Однако в некоторых случаях требуется коррекция предварительного результата. Мы не будем рассматривать здесь случаи умножения в дополнительном коде. Любознательным рекомендуем соответствующую литературу, например [11].

3.8.2. Методы ускорения умножения

Методы ускорения умножения принято делить [8, 11] на аппаратные и логические. Как те, так и другие требуют дополнительных затрат оборудования. При использовании аппаратных методов дополнительные затраты оборудования прямо пропорциональны числу разрядов в операндах. Эти методы вызывают усложнение схемы операционного автомата АЛУ.

Дополнительные затраты оборудования при реализации логических методов ускорения умножения не зависят от разрядности операндов. Усложняется в основном схема управления АЛУ. В ЭВМ для ускорения умножения часто используются комбинации этих методов.

К аппаратным методам ускорения умножения относятся ускорение выполнения операций сложения и сдвига, введение дополнительных цепей сдвига, позволяющих за один такт производить сдвиг информации в регистрах сразу на несколько разрядов, совмещение во времени операций сложения и сдвига, построение комбинационных схем множительных устройств, реализующих "табличное" и "матричное" умножение.

Пример реализации умножения с использованием n-входового сумматора показан на рис. 3.24.

Здесь частичные произведения формируются на схемах n-разрядных конъюнкторов одновременно и подаются на входы n-входового сумматора, причем в сумматоре за счет соответствующей коммутации цепей осуществляются сдвиги частичных произведений (как при выполнении умножения на бумаге "в столбик"). На выходе сумматора получается $2n$-разрядное произведение.

Метод табличного умножения (рис. 3.25) позволяет получить произведение за один такт при условии, что вся таблица умножения (результаты умножения всевозможных пар n-разрядных сомножителей!) будет размещена в памяти. Очевидно, для этого понадобится запоминающее устройство объемом

2^{2n} $2n$-разрядных слов (точно таким же способом можно выполнять и другие "длинные" операции — деление, вычисление функций). Так, для организации 8-разрядного умножителя потребуется память объемом $2^{16} \times 16$ бит = = 128 Кбайт, что для современного уровня развития интегральной технологии не кажется чрезмерным.

Однако для 16-разрядного АЛУ умножитель "потянет" уже на $2^{32} \times 32$ бит = = 16 Гбайт! Что касается современных 32-разрядных процессоров, то к расчету потребности в памяти для таких умножителей даже страшно приступать.

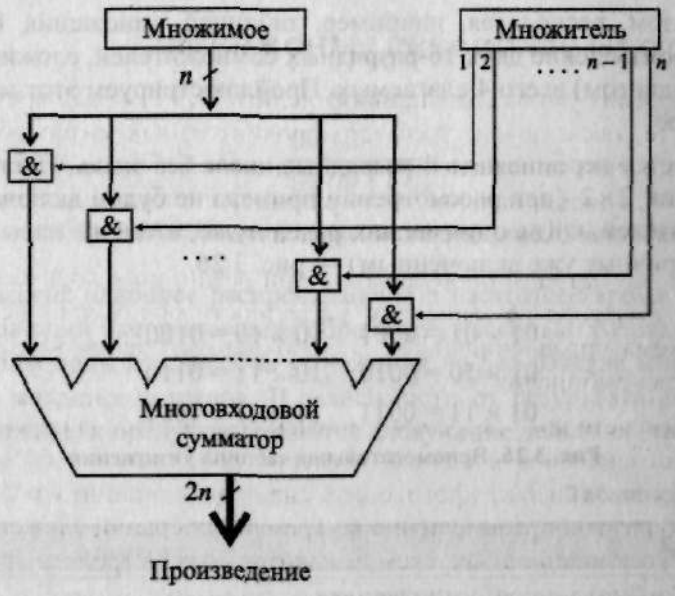


Рис. 3.24. Матричное умножение

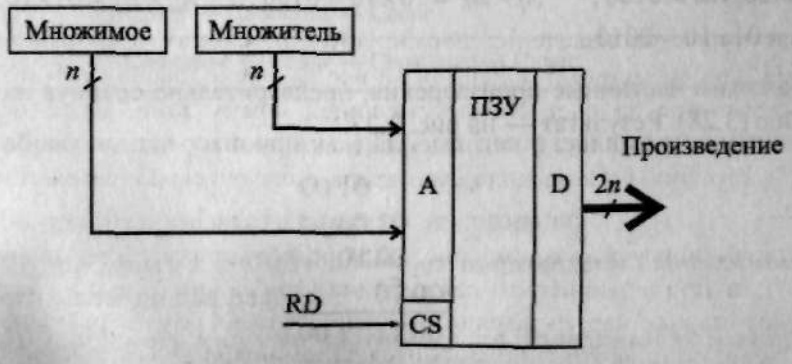


Рис. 3.25. Табличное умножение

В этом случае можно воспользоваться таблицей умножения меньшей разрядности, получая с ее помощью частичные произведения, а потом просуммировать их, предварительно сдвинув на соответствующее число разрядов.

Рассмотрим этот способ умножения подробнее. Пусть n — четное. Тогда каждый из двух сомножителей можно представить конкатенацией двух полей одинаковой разрядности $n/2$: $A = A_h A_l$, $B = B_h B_l$. В этом случае произведение можно представить следующим выражением:

$$A \times B = A_l \cdot B_l + 2^{\frac{n}{2}} \cdot A_h \cdot B_l + 2^{\frac{n}{2}} \cdot A_l \cdot B_h + 2^n \cdot A_h \cdot B_h. \quad (3.28)$$

Таким образом, располагая, например, таблицей умножения 8×8, можно получить произведение двух 16-разрядных сомножителей, сложив (с соответствующим сдвигом) всего 4 слагаемых. Проиллюстрируем этот метод на простом примере.

Пусть требуется перемножать 4-разрядные числа без знака. Построим таблицу умножения 2×2 (при рассмотрении примера не будем включать в нее пары сомножителей, когда один из них равен нулю, а так же пары сомножителей, симметричные уже включенным) — рис. 3.26.

$01 \times 01 = 0001 \quad 10 \times 10 = 0100$
$01 \times 10 = 0010 \quad 10 \times 11 = 0110$
$01 \times 11 = 0011 \quad 11 \times 11 = 1001$

Рис. 3.26. Вспомогательная таблица умножения

Пример 3.22

Выполним умножение $6 \times 10 = 60$ или в двоичном коде $01.10 \times 10.10 = 00111100$. Из таблицы получаем частичные произведения: $A_l \times B_l = 10 \times 10 = 0100$, $A_l \times B_h = 10 \times 10 = 0100$, $A_h \times B_l = 01 \times 10 = 0010$, $A_h \times B_h = 01 \times 10 = 0010$.

Теперь сложим частичные произведения, предварительно сдвинув их в соответствии с (3.28). Результат — на рис. 3.27.

```
      01 00
+     01 00
      00 10
   00 10
-------------
   00 11 11 00
```

Рис. 3.27. Результат выполнения примера 3.22

Пример 3.23

Выполним умножение $7 \times 11 = 77$ или в двоичном коде $01.11 \times 10.11 = 01001101$.

Из таблицы получаем частичные произведения: $A_l \times B_l = 11 \times 11 = 1001$, $A_l \times B_h = 11 \times 10 = 0110$, $A_h \times B_l = 01 \times 11 = 0011$, $A_h \times B_h = 01 \times 01 = 0001$.

Теперь сложим частичные произведения, предварительно сдвинув их в соответствии с (3.28). Результат — на рис. 3.28.

$$\begin{array}{r} 10\,01 \\ +\quad 01\,10 \\ \quad\;\; 00\,11 \\ 00\,10 \\ \hline 01\,00\,11\,01 \end{array}$$

Рис. 3.28. Результат выполнения примера 3.23

Среди *логических* наиболее распространены в настоящее время методы, позволяющие за один шаг умножения обработать несколько разрядов множителя. Рассмотрим один из способов умножения на два разряда множителя, начиная с его младших разрядов. В зависимости от результата анализа пары разрядов множителя предусматриваются следующие действия (табл. 3.3).

Таблица 3.3. Действия

Комбинация	Действие	Добавлено
00	Сдвиг — Сдвиг	0
01	Сложение — Сдвиг — Сдвиг	A
10	Сдвиг — Сложение — Сдвиг	$2A$
11	Сложение — Сдвиг — Сложение — Сдвиг	$3A = 4A - A$

Таким образом, для умножения сразу на два разряда множителя достаточно:

- при 00 просто произвести сдвиг на два разряда;
- при 01 прибавить к сумме частичных произведений множимое и произвести сдвиг на два разряда;
- при 10 прибавить к сумме частичных произведений удвоенное множимое и произвести сдвиг на два разряда;

- при 11 вычесть из суммы частичных произведений множимое (или добавить обратный (дополнительный) код множимого), произвести сдвиг на два разряда и добавить 1 к следующей (старшей) паре цифр множителя.

При классическом методе умножения двоичных n-разрядных чисел согласно выражению (3.26) потребуется n сдвигов суммы частичных произведений и $n/2$ (в среднем) сложений множимого с суммой частичных произведений. *Один из методов ускорения операции умножения — анализ сразу двух разрядов* множителя. Это позволит получить результат, применяя $n/2$ сдвигов и (в среднем) $3n/8$ сложений/вычитаний.

3.9. Алгоритмы деления

Знак частного, как и знак произведения, не зависит от соотношения модулей операндов и определяется в зависимости от знаков операндов по выражению (3.25). Поэтому рассмотрим сначала процесс *деления модулей* двоичных чисел.

Пусть A — делимое, B — делитель, C — частное, W — остаток.

Очевидно, при представлении чисел с фиксированной запятой как дробных, должно соблюдаться условие

$$|A| < |B|, \tag{3.29}$$

иначе $C \geq 1$, что соответствует переполнению разрядной сетки.

Процесс деления двоичных чисел может быть сведен к последовательности вычитаний и анализа знаков получающихся остатков. Сформулируем словесный алгоритм деления следующим образом:

1. Вычетают *из делимого делитель. Если знак разности* 0, *то деление* невозможно в силу нарушения условия (3.29), и следует, установив $OV = 1$, завершить операцию; иначе в разряд целой части частного записывают 0 (в конце операции в этот разряд помещается знак частного).
2. *Так как остаток* (разность $A - B$) *оказался отрицательным, восстанавливают* остаток путем добавления делителя к остатку.
3. Сдвигают восстановленный остаток влево на один разряд.
4. Вычитают из сдвинутого остатка делитель; если полученная разность положительна, то очередной цифрой частного становится 1, и следует перейти к п. 3; иначе очередная цифра частного — 0 и переходят к п. 2.

Пункты 2—4 повторяют столько раз, сколько цифр требуется получить в частном.

Пример 3.24

Деление дробных положительных чисел $+(3/16):+(12/16)=+(1/4)=+(4/16)$ приведено на рис. 3.29.

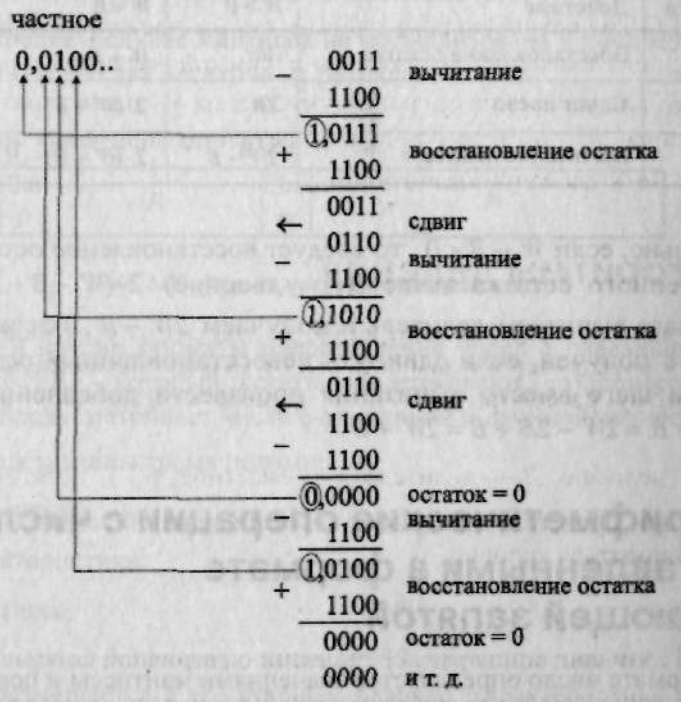


Рис. 3.29. Пример деления

3.9.1. Деление без восстановления остатка

Приведенный выше метод деления называется методом деления с *восстановлением остатка*. При получении отрицательного остатка на очередном шаге деления необходимо перед левым сдвигом восстановить остаток путем добавления к нему делителя. При этом для получения n-разрядного частного требуется в среднем $1{,}5n$ циклов сложения/вычитания.

Существует алгоритм деления *без восстановления остатка*, позволяющий корректировать отрицательные остатки без дополнительного цикла сложения. Рассмотрим действия, производимые с остатками в цикле деления в зависимости от полученного знака остатка (табл. 3.4).

Видно, что если на очередном шаге остаток получился отрицательный, его можно не восстанавливать, но на следующем шаге в этом случае нужно вме-

сто вычитания делителя из сдвинутого остатка добавить делитель к сдвинутому остатку.

Таблица 3.4. Действия, производимые с остатками в цикле деления

Номер шага	Действие	$W>0$	$W<0$
1	Восстановление остатка	Нет	$W+B$
2	Сдвиг влево	$2W$	$2\cdot(W+B)$
3	Вычитание делителя	$2W-B$	$2\cdot(W+B)-B=2W+B$

Действительно, если $W-B<0$, то следует восстановление остатка и сдвиг восстановленного остатка влево (его удвоение) $2\cdot(W-B+B)$. На следующем шаге вычитаем делитель и получаем $2W-B$. Тот же результат может быть получен, если сдвинуть невосстановленный остаток, но на следующем шаге вместо вычитания произвести добавление делителя: $2\cdot(W-B)+B=2W-2B+B=2W+B$.

3.10. Арифметические операции с числами, представленными в формате *с плавающей запятой*

В таком формате число определяется значениями мантиссы и порядка:

$$N=m\cdot q^{p}, \tag{3.30}$$

где m — мантисса числа, p — порядок, q — основание.

Мантисса и порядок могут иметь свои знаки, причем знак мантиссы соответствует знаку числа. Основание q может не совпадать с основанием системы счисления. При операциях с двоичными числами часто для расширения диапазона представления чисел выбирают $q=2^{k}$, например, $q=16$.

В машинном представлении формат числа с плавающей запятой (рис. 3.30) задается двумя полями — полем мантиссы m и полем порядка p, причем каждое поле имеет свой разряд знака. Значение порядка в формате числа не указывается — оно подразумевается одинаковым для всех чисел.

Мантисса и порядок представляются в формате с фиксированной запятой, причем обычно порядок — целое число со знаком (запятая фиксирована после младшего разряда), а мантисса — правильная дробь (запятая фиксирована

между знаковым разрядом и старшим разрядом модуля). С целью увеличения точности представления числа в заданном формате мантиссу представляют в *нормализованной* форме, когда старший разряд модуля мантиссы — не ноль (для прямых кодов). Действительно,

$$0{,}2364 \cdot 10^4 \approx 0{,}0024 \cdot 10^6,$$

однако в последнем случае мантисса не нормализована и точность представления числа — всего два десятичных разряда.

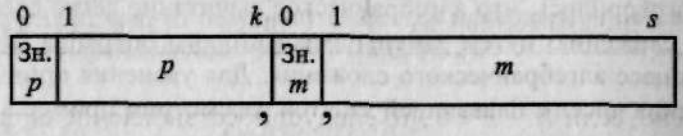


Рис. 3.30. Формат числа с плавающей запятой

Существуют и другие форматы представления чисел с плавающей запятой. Так, стандарт IEEE[1], который, кстати, поддерживают (со)процессоры семейства *x*86(87) предусматривает числа с одинарной и двойной точностью.

Форматы представлены тремя полями:

- s — знак числа;
- e — характеристика;
- m — мантисса.

Формат с *одинарной точностью* занимает 32-разрядное двоичное слово, причем знак s размещается в его старшем разряде, характеристика e — в следующих 8 разрядах и, наконец, 23 младших разряда занимает мантисса m.

Порядок p, под который отводится один байт, может принимать значения в диапазоне ±127. Характеристика в стандарте IEEE получается как порядок *с избытком 127*: $e = p + 127$. При этом характеристика всегда положительна, что упрощает выполнение арифметических операций.

Мантисса числа в стандарте IEEE нормализована и лежит в диапазоне $1 \le m < 2$. Целая часть мантиссы всегда равна 1, поэтому значение целой части не хранится в формате числа, а подразумевается (т. н. "скрытая единица"). Дробная часть мантиссы хранится в 23 младших разрядах формата.

---

[1] Institute of Electrical and Electronics Engineers — институт инженеров по электротехнике и электронике.

Формат с *двойной точностью* отличается длиной полей характеристики (11 битов *с избытком 1023*) и мантиссы (52 бита) и размещается в 64-разрядном двоичном слове.

Попробуйте самостоятельно оценить диапазон представления чисел в форматах IEEE. Подробности о выполнении операций с этими форматами можно посмотреть в [3, 12].

3.10.1. Сложение и вычитание

Ранее мы договорились, что алгебраическое вычитание легко свести к алгебраическому сложению путем замены знака второго операнда. Поэтому рассмотрим процесс алгебраического сложения. Для уяснения принципа выполнения сложения чисел с плавающей запятой рассмотрим пример в десятичной системе.

Пример 3.25

Сложить два числа, представленные в формате с плавающей запятой:

$A = 0{,}315290 \cdot 10^{-2}$, $B = 0{,}114082 \cdot 10^{+2}$.

Обратите внимание, мантиссы чисел нормализованы. Очевидно, прежде чем складывать мантиссы, требуется преобразовать числа таким образом, чтобы они имели *одинаковые порядки*. Выравнивание порядков можно выполнить двумя способами — уменьшением большего порядка до меньшего или увеличением меньшего до большего (рис. 3.31).

$A =$	$0{,}315290 \cdot 10^{-2}$		$A =$	$0{,}000031$	$5290 \cdot 10^{+2}$
$B = 1140$	$0{,}820000 \cdot 10^{-2}$		$B =$	$0{,}114082$	$\cdot 10^{+2}$
$C =$	$1{,}135290 \cdot 10^{-2}$		$C =$	$0{,}114114$	$\cdot 10^{+2}$
	а			*б*	

Рис. 3.31. Выравнивание порядков

В первом случае за разрядную сетку выходят старшие разряды сдвигаемой мантиссы и результат сложения оказывается неверным. Во втором случае при сдвиге теряются младшие разряды мантиссы, что не влияет на точность результата. Поэтому при выравнивании порядков всегда следует *увеличивать меньший порядок до большего* при соответствующем уменьшении мантиссы.

Для выравнивания порядков следует определить разность порядков слагаемых и сдвинуть мантиссу числа с меньшим порядком вправо на величину

этой разности. Если разность порядков превышает разрядность поля мантиссы, то значение слагаемого с меньшим порядком может быть принята за 0, а результат суммирования будет равен слагаемому с большим порядком.

После выравнивания порядков следует сложить мантиссы и определить в качестве порядка результата порядок любого из слагаемых (после выравнивания порядки слагаемых равны). Если при сложении мантисс возникает переполнение, то результат может быть исправлен путем сдвига мантиссы суммы на один разряд вправо и добавления единицы к порядку результата.

Однако если в результате этого добавления произойдет переполнение разрядной сетки порядков, то результат окажется неверным — имеет место т. н. *положительное переполнение*: $OV := 1$ (рис. 3.32).

$$A = 0{,}96502 \cdot 10^{+2} \qquad A = 0{,}96502 \cdot 10^{+2}$$
$$B = 0{,}73005 \cdot 10^{+1} \qquad B = 0{,}07300 \cdot 10^{+2}$$
$$C = 1{,}03802 \cdot 10^{+2} \quad \text{— переполнение мантисс!}$$
$$C = 1{,}0380 \cdot 10^{+3} \quad \text{— правильный результат}$$

Рис. 3.32. Положительное переполнение

В результате алгебраического сложения мантисс результат может оказаться *ненормализованным*. *Для нормализации результата необходимо сдвигать* мантиссу результата влево до тех нор, пока в старшем значащем разряде не окажется цифра, отличная от 0 (в двоичной системе это 1), сопровождая каждый сдвиг уменьшением на 1 порядка результата (рис. 3.33). Этот процесс называется *нормализацией результата.*

$$A = 0{,}24512 \cdot 10^{-8}$$
$$B = -0{,}24392 \cdot 10^{-8}$$
$$C = 0{,}00120 \cdot 10^{-8} \quad = 0{,}12000 \cdot 10^{-10}$$

Рис. 3.33. Положительное переполнение

В процессе уменьшения порядка при нормализации может оказаться, что модуль порядка превысил максимальную величину, размещаемую в поле порядка. Этот случай называют *отрицательным переполнением*. Его можно избежать, оставив результат ненормализованным, однако принято считать, что сохранять ненормализованный результат в памяти недопустимо. Поэтому в случае отрицательного переполнения результат принимает значение "машинный ноль".

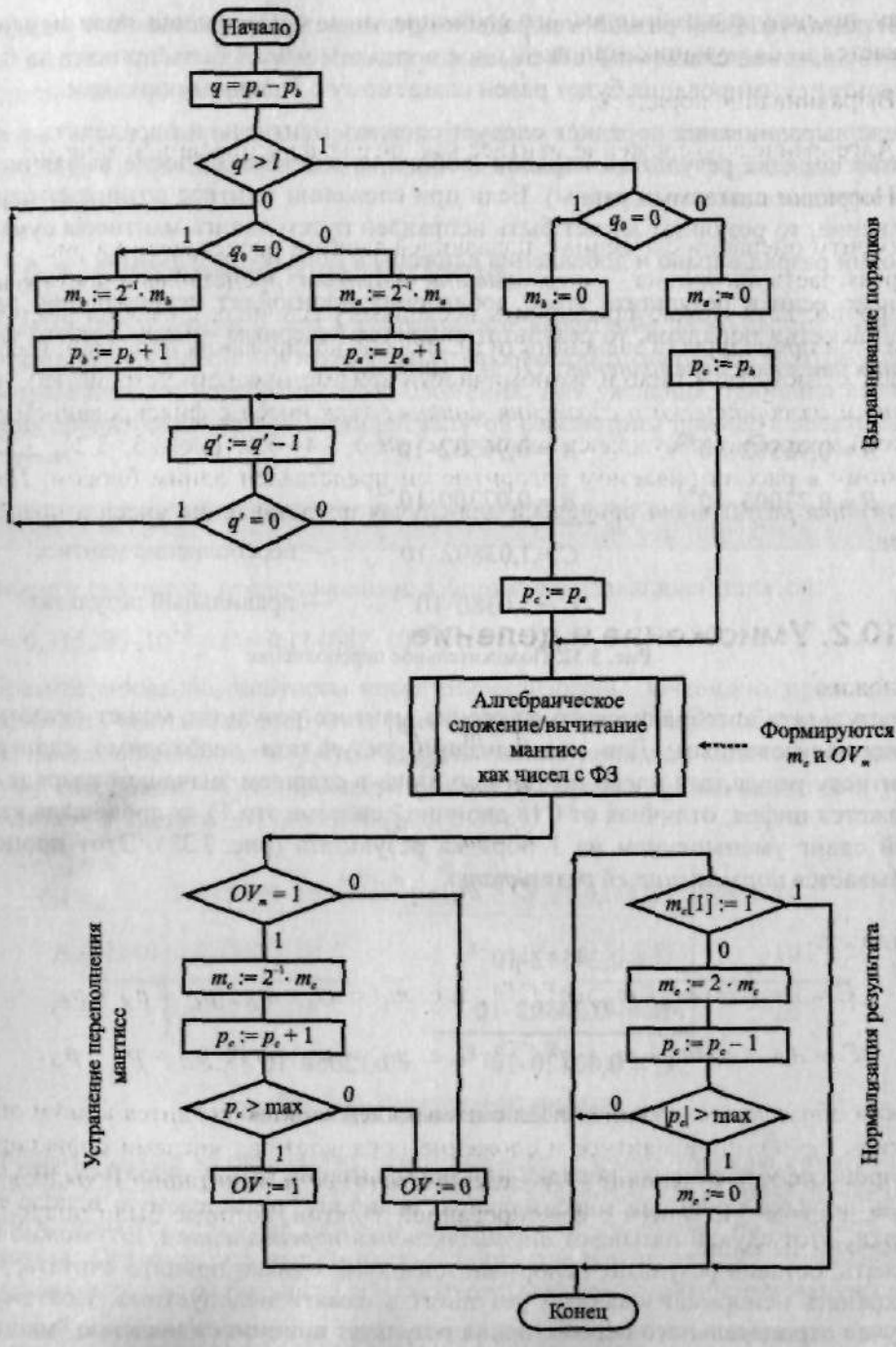


Рис. 3.34. Алгоритм операции сложения чисел с плавающей запятой

Итак, процедура алгебраического сложения чисел с плавающей запятой складывается из следующих этапов:

1. Выравнивание порядков.
2. Алгебраическое сложение мантисс как чисел с фиксированной запятой.
3. Нормализация результата.

Алгоритм операции сложения с плавающей запятой представлен на рис. 3.34. Первая часть алгоритма — *выравнивание порядков*, представлена достаточно подробно, хотя можно предложить несколько различных способов реализации этой процедуры (в зависимости от способа кодирования порядков, требования к быстродействию и экономичности арифметического устройства). *Алгоритм алгебраического сложения мантисс* (как чисел с фиксированной запятой) подробно обсуждался выше (*см. разд. 3.4—3.6*, рис. 3.3, 3.21, 3.22), поэтому в рассматриваемом алгоритме он представлен одним блоком. *Нормализация результата* приведена для случая представления чисел в прямом коде.

3.10.2. Умножение и деление

Положим

$$A = m_A \cdot q^{p_A};$$
$$B = m_B \cdot q^{p_B};$$
$$C = m_C \cdot q^{p_C};$$
$$D = m_D \cdot q^{p_D}.$$

Тогда

$$C = A \times B = (m_A \times m_B) \cdot q^{p_A + p_B}, \text{ т. е. } m_C = m_A \times m_B,\ p_C = p_A + p_B.$$

$$D = A \div B = (m_A \div m_B) \cdot q^{p_A - p_B}, \text{ т. е. } m_D = m_A \div m_B,\ p_D = p_A - p_B.$$

Таким образом, умножение чисел с плавающей запятой сводится к двум операциям (умножение мантисс и сложение порядков) над числами с фиксированной запятой, а деление — к делению мантисс и вычитанию порядков — также к двум операциям с фиксированной запятой, которые были подробно рассмотрены выше.

3.11. Арифметические операции над десятичными числами

3.11.1. Кодирование десятичных чисел

При использовании в ЦВМ десятичные числа кодируются группой двоичных разрядов. Учитывая, что

$$\log_2 10 \approx 3{,}32, \qquad (3.31)$$

для представления одной десятичной цифры требуется не менее четырех двоичных разрядов. Соответствие между десятичной цифрой и ее двоичным представлением называют *двоичным кодом десятичной цифры*. Наиболее естественным представляется кодирование десятичных цифр позиционными двоичными кодами с естественными весами разрядов. Такой код принято называть *кодом "8421"*.

Ясно, что это далеко не единственный способ кодирования десятичных цифр. Используя только четырехразрядные двоичные коды, следует выбрать 10 из шестнадцати возможных комбинаций для представления цифр. Количество способов, которыми могут быть выбраны 10 комбинаций из 16, равно числу сочетаний из 16 по 10:

$$C_{16}^{10} = \frac{16!}{10!\cdot 6!}.$$

После того как выбор комбинаций сделан, можно $P_{10} = 10!$ способами сопоставить комбинацию десятичной цифре. Таким образом, общее число различных четырехразрядных кодов десятичных цифр составляет

$$A_{16}^{10} = C_{16}^{10} \cdot P_{10} = \frac{16!}{10!\cdot 6!} \cdot 10! = \frac{16!}{6!} \approx 3 \cdot 10^{10}.$$

Практически лишь 5—6 различных кодов используют в ЦВМ для представления десятичных цифр.

Основной недостаток кодирования десятичных цифр в коде "8421" состоит в несоответствии веса десятичного и шестнадцатеричного переносов. Действительно, перенос из тетрады шестнадцатеричной цифры имеет вес 16, а десятичный перенос — 10.

Для устранения этого противоречия можно выбрать другие способы кодирования десятичных цифр. Например, код "8421+3" (иногда его называют *код с избытком три*) позволяет при сложении получать сумму "с избытком 6", при этом вес переноса соответствует десятичному.

Можно подобрать такие веса двоичных разрядов при кодировании десятичных цифр, чтобы их сумма равнялась 10. Например, код "5211" обладает

именно таким свойством. При этом, однако, нарушается свойство функциональности соответствия десятичных цифр и их двоичного представления. Например, цифра 7 может быть представлена как 1100 или как 1011. Для преодоления этого недостатка достаточно договориться, чтобы в подобных ситуациях всегда сначала заполнялись *младшие* разряды кода.

В табл. 3.5 приведены упомянутые двоичные коды десятичных цифр.

Таблица 3.5. Двоичные коды десятичных цифр

Цифры	Код "8421"	Код "8421+3"	Код "5211"
0	0000	0011	0000
1	0001	0100	0001
2	0010	0101	0011
3	0011	0110	0101
4	0100	0111	0111
5	*0101*	*1000*	*1000*
6	0110	1001	1001
7	0111	1010	1011
8	1000	1011	1101
9	1001	1100	1111

Арифметические операции над десятичными числами можно выполнять как на специальных десятичных сумматорах (в этом случае можно применять любую кодировку десятичных цифр), так и на обычных двоичных сумматорах. В последнем случае десятичные числа обрабатываются *по правилам двоичной арифметики*, и десятичный результат операции, естественно, нуждается в коррекции. В этом случае сложность коррекции и длительность ее реализации существенно зависят от выбранного кода.

3.11.2. Арифметические операции над десятичными числами

Рассмотрим выполнение операции сложения десятичных чисел в коде "8421" по правилам двоичной арифметики.

Пример 3.26

Результат — на рис. 3.35.

$$
\begin{array}{l}
A = 4754 = 0100\ \ 0111\ 0101\ 0100 \\
B = 2917 = 0010\ \ 1001\ 0001\ 0111 \\
\hline
C = 7671 = 0111\ \leftarrow 0000\ 0110\ 1011 \\
\qquad\qquad\quad 7\ \ \boxed{0}\ \ 6\ \ \boxed{11}
\end{array}
$$

Рис. 3.35. Результат выполнения примера 3.26

Из рассмотренного примера видно, что тетрады результата (назовем его предварительным), обрамленные рамкой, нуждаются в коррекции. Действительно, если сумма тетрад, представляющих соответствующие десятичные разряды слагаемых, больше 9, но не превышает 15, двоичный перенос в следующую тетраду не формируется. В этом случае требуется выработать искусственный перенос и удалить из тетрады 10, что соответствует добавлении шестерки (0110) и передаче обязательно возникающего при этом переноса в следующий старший разряд.

Если из тетрады был двоичный перенос (в примере он отмечен знаком ←), то он "унес" из тетрады 16, в то время как десятичный перенос должен "уносить" только 10. Следовательно, в такие тетрады, из которых был перенос, следует добавить шестерку (0110).

Таким образом, коррекция предварительной двоичной суммы при использовании кода "8421" заключается в добавлении кода 0110 ко всем тетрадам предварительной суммы, значение которых превышает 9 или из которых был двоичный перенос. Возникающие при коррекции переносы должны обязательно передаваться в следующую старшую тетраду. Выполним операции примера 3.26 с учетом сформулированных правил коррекции.

Пример 3.27

Результат — на рис. 3.36.

$$
\begin{array}{l}
A = 4754 = 0100 \leftarrow 0111\ 0101\ \ 0100 \\
B = 2917 = 0010\ \ 1001\ 0001\ \ 0111 \\
\hline
C = 7671\ \ 0111\ \ 0000\ 0110 \leftarrow 1011 \\
\qquad\qquad\quad 0110\ \qquad\qquad 0110 \\
\hline
\qquad = 0111\ 0110\ \ 0111\ 0001 \\
\qquad\quad 7\ \ \ 6\ \ \ 7\ \ \ 1
\end{array}
$$

Рис. 3.36. Результат выполнения примера 3.27

Рассмотрим еще один пример арифметического сложения в коде "8421".

Пример 3.28

Результат — на рис. 3.37.

$$
\begin{array}{lllll}
A = 3852 = 0011 & 1000 & 0101 & 0010 \\
B = 5179 = 0101 & 0001 & 0111 & 1001 \\
\hline
C = 9031 \quad 1000 & 1001 & \leftarrow 1100 & \leftarrow 1011 \\
 & & 0110 & 0110 \\
\hline
 & 1000 & \leftarrow 1010 & 0011 & 0001 \\
 & & 0110 & & \\
\hline
 & = 1001 & 0000 & 0011 & 0001 \\
 & 9 & 0 & 3 & 1
\end{array}
$$

Рис. 3.37. Результат выполнения примера 3.28

Из примера 3.28 видно, что межтетрадные переносы, возникающие в процессе коррекции предварительной суммы, могут таким образом изменить старшие тетрады, что их также потребуется корректировать. В худшем случае количество последовательных коррекций будет равно разрядности слагаемых (рассмотрите пример сложения 9999 + 1 в коде "8421").

На основании вышеизложенного можно отметить следующие недостатки применения кода "8421":

- необходимо отслеживать не только переносы из тетрад, но и значения модулей тетрад предварительной суммы;
- в общем случае невозможно произвести одновременно коррекцию во всех тетрадах, где может потребоваться коррекция.

Для преодоления отмеченных выше недостатков можно использовать другие коды, например "8421+3". При кодировании с избытком три каждая десятичная цифра представляется как $a'_i = a_i + 0011$, где a_i — код "8421" цифры.

Тогда при сложении

$$c_i^h = (a_i + 3) + (b_i + 3) + p_{i-1} = a_i + b_i + p_{i-1} + 6, \qquad (3.32)$$

где c^h — предварительная сумма, в тетраде всегда будет формироваться истинное значение десятичного переноса — для всех комбинаций десятичных слагаемых, для которых $a_i + b_i + p_{i-1} > 0$, значение $a'_i + b'_i + p_{i-1} > 15$. Однако если переноса из тетрады не было, то результат сформируется "с избытком 6", поэтому потребуется коррекция тетрады предварительной суммы — удаление из тетрады лишней тройки. Вычитание (-3) можно заменить сложени-

ем с дополнением до 3 — (+13). Обязательно возникающий при этом перенос не передается в следующую тетраду. Потеря переноса равносильно потере 16, т. е. $-16+13=-3$.

Если перенос из тетрады был, то его вес равен $2^4=16$, таким образом, из тетрады удаляется 16, а вес десятичного переноса — 10. Поэтому перенос из тетрады в коде "8421+3" уносит из тетрады лишнюю шестерку, которую и нужно добавить при коррекции. Но согласно (3.32) сложение тетрад "с избытком 3" приводит к получению суммы "с избытком 6", поэтому вместо добавления шестерки достаточно добавить тройку.

Итак, коррекции при сложении в коде "8421+3" подлежат все тетрады предварительной суммы, причем к тем тетрадам, из которых сформировался перенос, следует добавить константу 0011, а к тетрадам, из которых не было переноса, добавить константу 1101. Возникающие при коррекции межтетрадные переносы игнорируются!

Таким образом, коррекция при сложении в коде "8421+3", во-первых, определяется только значениями переносов из тетрад предварительной суммы и, во-вторых, может проводиться параллельно во всех тетрадах.

Пример 3.29

Результат — на рис. 3.38.

$A=3852=$	0110	1011	1000	0101
$B=5179=$	1000	0100	1010	1100
$C=9031$	1111	←0000	←0011	←0001
	1101	0011	0011	0011
$=$	1100	0011	0110	0100
	12 – 3=**9**	3 – 3=**0**	6 – 3=**3**	4 – 3=**1**

Рис. 3.38. Результат выполнения примера 3.29

Еще одним достоинством кода "8421+3" является простой способ получения дополнения до 9 — достаточно просто проинвертировать разряды кода.

Действительно, проинвертировав все разряды четырехразрядного двоичного числа a, мы получим его дополнение до $1111=15_{10}$, что в коде "с избытком 3" соответствует $15-(a+3)=(9-a)+3$.

Это свойство позволяет довольно просто реализовать операцию вычитания через сложение в обратном или дополнительном коде.

3.12. Машинная арифметика в остаточных классах

Органическим недостатком любой позиционной системы счисления является наличие *межразрядных связей*. Действительно, результат сложения в i-м разряде зависит не только от значений i-х разрядов слагаемых, но и от переноса из $i-1$ разряда и, в конечном итоге — от значений всех младших разрядов слагаемых: $i-1$, $i-2$, ..., 1, 0. Поэтому вычисление разрядов суммы может проходить только последовательно (с учетом формирования переноса из предыдущего (младшего) разряда. Это обстоятельство препятствует распараллеливанию процесса вычисления и, естественно, снижает быстродействие процессора.

В рамках позиционных систем счисления известно [2, 8, 11] несколько способов логического и схемотехнического ускорения арифметических операций — параллельный перенос, матричная и табличная арифметика и др., однако все они требуют весьма значительных аппаратных затрат.

Поиск новых путей построения арифметических устройств ЭВМ, позволяющий исключить зависимость между разрядами при выполнении арифметических операций, привел к применению в машинной арифметике аппарата *теории вычетов* — одного из разделов теории чисел. В рамках этого аппарата разработана [1] непозиционная система счисления — *система счисления в остаточных классах* (СОК).

3.12.1. Представление чисел в системе остаточных классов

Будем говорить, что "α есть остаток числа A по модулю p" (иногда говорят, что "A сравнимо с α по модулю p"), если имеет место следующее равенство:

$$\alpha = A - \left[\frac{A}{p}\right] \cdot p, \tag{3.33}$$

где $\left[\frac{A}{p}\right]$ — целая часть частного A/p, причем a — наименьший целый остаток от деления A на p.

Часто это соотношение записывают так:

$$\alpha \equiv A \,(\mathrm{mod}\, p). \tag{3.34}$$

Для представления чисел в СОК необходимо выбрать т. н. *систему оснований* — множество целых чисел p_1, p_2, ..., p_n. Тогда любое число A может быть представлено в СОК следующим образом:

$$A = (\alpha_1, \alpha_2, \ldots, \alpha_n), \tag{3.35}$$

где $\alpha \equiv A \pmod{p}$.

Обозначим произведение

$$\prod_{i=1}^{n} p_i = p_1 \cdot p_2 \cdot \ldots \cdot p_n = P. \tag{3.36}$$

Можно показать [1], что если все основания p_i — взаимно-простые числа, то между числами 0, 1, 2, ..., $(P-1)$ и числами, представленными в СОК согласно (3.35), имеет место *взаимно-однозначное соответствие*.

Пример 3.30

Пусть $p_1 = 3$, $p_2 = 5$, $p_3 = 7$ — взаимно-простые числа.

$P = 3 \cdot 5 \cdot 7 = 105$.

Представим в СОК несколько десятичных чисел:

17 = (2,2,3)	1 = (1,1,1)	100 = (1,0,2)
63 = (0,3,0)	0 = (0,0,0)	105 = (0,0,0)
55 = (1,0,6)	11 = (2,1,4)	106 = (1,1,1)

Заметим, что при выходе за пределы диапазона $[0, (P-1)]$ нарушается взаимно-однозначное соответствие между представлением чисел в позиционной системе счисления и СОК. Действительно.

$0 \equiv 105 \equiv 210 \equiv \ldots \equiv (\text{mod } 3), (\text{mod } 5), (\text{mod } 7)$;

$1 \equiv 106 \equiv 211 \equiv \ldots \equiv (\text{mod } 3), (\text{mod } 5), (\text{mod } 7)$

и т. д. Очевидно, для расширения диапазона представления чисел в СОК следует увеличить число и/или значения оснований.

3.12.2. Арифметические операции с положительными числами

Рассмотрим правила выполнения операций *сложения* и *умножения* в СОК в случае, если оба операнда и результат операции находятся в диапазоне $[0, P)$.

Пусть

$$A=(\alpha_1,\alpha_2,\ldots,\alpha_n),$$
$$B=(\beta_1,\beta_2,\ldots,\beta_n),$$
$$A+B=(\gamma_1,\gamma_2,\ldots,\gamma_n),$$
$$A\cdot B=(\delta_1,\delta_2,\ldots,\delta_n),$$

и при этом имеет место соотношение $A<P$, $B<P$, $A+B<P$, $A\cdot B<P$.

Покажем [1], что

$$\gamma_i=(\alpha_i+\beta_i) \pmod{p_i},$$
$$\delta_i=(\alpha_i\cdot\beta_i) \pmod{p_i},$$

при этом в качестве цифры результата берется наименьший остаток

$$\gamma_i=\alpha_i+\beta_i-\frac{\alpha_i+\beta_i}{p_i}\cdot p_i, \tag{3.37}$$

$$\delta_i=\alpha_i\cdot\beta_i-\frac{\alpha_i\cdot\beta_i}{p_i}\cdot p_i. \tag{3.38}$$

Действительно, на основании (3.33) можно написать

$$\gamma_i=\alpha_i+\beta_i-\frac{\alpha_i+\beta_i}{p_i}\cdot p_i, \text{ для } i=1,2,\ \ldots,n\ .$$

Из представления A и B следует, что

$$A=k_ip_i+\alpha_i,\ B=l_ip_i+\beta_i,\ i=1,2,\ \ldots,n\ , \tag{3.39}$$

где k_i и l_i — целые неотрицательные числа. Тогда

$$A+B=(k_i+l_i)p_i+\alpha_i+\beta_i,$$
$$\left[\frac{A+B}{p_i}\right]=k_i+l_i+\left[\frac{\alpha_i+\beta_i}{p_i}\right],\ i=1,2,\ \ldots,n\ .$$

Откуда

$$\gamma_i=\alpha_i+\beta_i-\left[\frac{\alpha_i+\beta_i}{p_i}\right]\cdot p_i,$$

что и доказывает (3.37).

В случае умножения

$$\delta_i=A\cdot B-\left[\frac{A\cdot B}{p_i}\right]\cdot p_i,\ i=1,2,\ \ldots,n\ .$$

Учитывая (3.39), получим

$$A \cdot B = k_i l_i p_i^2 + (\alpha_i l_i + \beta_i k_i) k_i + \alpha_i \beta_i,$$

$$\left[\frac{A \cdot B}{p_i}\right] = k_i l_i p_i + \alpha_i l_i + \beta_i k_i + \left[\frac{\alpha_i \cdot \beta_i}{p_i}\right], \; i = 1, 2, \ldots, n.$$

Следовательно

$$\delta_i = \alpha_i \cdot \beta_i - \left[\frac{\alpha_i \cdot \beta_i}{p_i}\right] \cdot p_i,$$

что и доказывает (3.38).

Рассмотрим несколько примеров, иллюстрирующих приведенные выше правила.

Пример 3.31

Выполним *сложение чисел, представленных в СОК*. Результат — *на* рис. 3.39.

17 = (2,2,3)		55 = (1,0,6)		55 = (1,0,6)
63 = (0,3,0)	+	17 = (2,2,3)		63 = (0,3,0)
80 ⇐ (2,0,3)		11 = (2,1,4)		118 ⇐ (1,3,6) = 13 = 118 − P
		83 ⇐ (2,3,6)		

Рис. 3.39. Результат выполнения примера 3.31

Обратите внимание, если результат выходит за пределы допустимого диапазона ($A + B \geq P$), то в СОК он неотличим от $A + B - P$ (путем весьма сложных ухищрений можно обнаружить переполнение суммы в СОК [1]).

Пример 3.32

Выполним *умножение чисел, представленных в СОК*. Результат — *на* рис. 3.40.

17 =(2, 2,3)	78 =(0,3,1)	18 =(0,3,4)
6 =(0,1,6)	1 =(1,1,1)	5 =(2,0,5)
102 ⇐ (0,2, 4)	78 ⇐ (0,3,1)	90⇐ (0,0,6)

Рис. 3.40. Результат выполнения примера 3.32

Операция *вычитания* в общем случае в СОК не определена, т. к. в СОК отсутствуют отрицательные числа. Однако в частных случаях, когда A, B, $(A-B)\in[0, P)$, можно записать

$$\lambda_i = \alpha_i - \beta_i - \left[\frac{\alpha_i - \beta_i}{p_i}\right]\cdot p_i, \qquad \lambda_i = (\alpha_i - \beta_i) \pmod{p_i}, \quad i = 1, 2, \ldots n. \tag{3.40}$$

Операция вычитания в тех случаях, когда ее результат положителен, выполняется вычитанием соответствующих цифр разрядов по модулю соответствующего основания, т. е. если цифра уменьшаемого меньше соответствующей цифры вычитаемого, то к цифре умеьшаемого добавляется соответствующее основание.

Пример 3.33

Рассмотрим вычитание $A-B$ для случаев $A>B$. Результат — на рис. 3.41.

17 =(2, 2,3)	78 =(0,3,1)	98 =(2, 3,0)
6 =(0,1,6)	41 =(2, 1,6)	55 =(1,0, 6)
11 ⇐ (2,1,4)	37 ⇐ (1,2, 2)	43 ⇐ (1,3,1)

Рис. 3.41. Результат выполнения примера 3.33

3.12.3. Арифметические операции с отрицательными числами

Если необходимо оперировать отрицательными числами, можно ввести т. н. *искусственные формы* представления чисел в СОК. Выражение (3.36) определяет диапазон представления чисел в СОК с основаниями p_1, p_2, ..., p_n. Пусть одно из оснований системы равно 2, например, для определенности $p_1 = 2$.

Обозначим через h величину

$$h = \frac{P}{2} = \frac{P}{p_1} = p_1 \cdot p_1 \cdot \ldots p_1 = (1, 0, 0, \ldots 0).$$

Будем оперировать числами, лежащими в диапазоне $0 < |N| < h$.

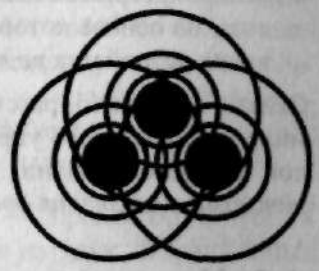
Примем в качестве нуля число h и будем представлять положительные числа $N = |N|$ в виде $N' = h + |N|$, а отрицательные числа $N = -|N|$ в виде

$N' = h - |N|$. Тогда при алгебраическом суммировании получим следующий вид представления положительных и отрицательных чисел:

$$N' = h + N.$$

Это означает, что в принятом представлении мы всегда будем иметь дело с положительными числами, однако числа в искусственной форме N' в интервале $[0, h)$ будут отображать отрицательные числа, а в интервале $[h, P)$ — положительные.

ГЛАВА 4



Организация устройств ЭВМ

4.1. Принцип микропрограммного управления

Для выполнения операций над информацией используются *операционные устройства* — арифметико-логические, управления, контроллеры ВУ и т. п. Функцией операционного устройства является выполнение заданного множества операций $F = \{f_1, f_2, ..., f_K\}$ над входными словами из множества D_I с целью вычисления выходных слов из множества D_O, представляющих результаты операций $D_O = f_k(D_I)$, $k = 1, ..., K$.

Функциональная и структурная организация операционных устройств, определяющая порядок функционирования и структуру устройств, базируется на *принципе микропрограммного управления*, который состоит в следующем [7]:

1. Любая операция f_k, реализуемая устройством, рассматривается как сложное действие, которое разделяется на последовательность элементарных действий над словами информации, называемых *микрооперациями*.
2. Для управления порядком следования микроопераций используются *логические условия*, которые, в зависимости от значений слов, преобразуемых микрооперациями, принимают значения "истина" или "ложь" (1 или 0).
3. Процесс выполнения операций в устройстве описывается в форме алгоритма, представляемого в терминах микроопераций и логических условий и называемого *микропрограммой*. Микропрограмма определяет порядок проверки значений логических условий и следования микроопераций, необходимый для получения требуемых результатов.

4. Микропрограмма используется как форма представления функции устройства, на основе которой определяются структура и порядок функционирования устройства во времени.

Сказанное можно рассматривать как содержательное описание принципа микропрограммного управления, из которого следует, что структура и порядок функционирования операционного устройства предопределяются *алгоритмами выполнения операций из* F.

4.2. Концепция операционного и управляющего автоматов

В функциональном и структурном отношении операционное устройство, входящее в состав ЭВМ, удобно представить разделенным на две части: операционный и управляющий автоматы (рис. 4.1).

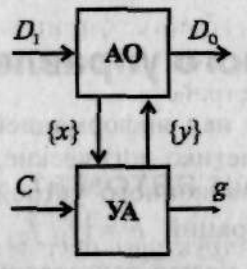


Рис. 4.1. *Устройство как композиция автоматов*

Операционный автомат (ОА) служит для хранения слов информации, выполнения набора микроопераций и вычисления значений логических условий, т. е. операционный автомат является структурой, организованной для выполнения действий над информацией. На вход ОА подаются входные данные D_1, которые в соответствии с алгоритмом операции преобразуются в выходные данные D_0. Кроме того, ОА вырабатывает множество $\{x\}$ осведомительных сигналов (логических условий) для управляющего автомата.

Управляющий автомат (УА) генерирует последовательность управляющих сигналов $\{y\}$, обеспечивающую выполнение в операционном автомате заданной последовательности элементарных действий, которая реализует алгоритм выполняемой операции. Управляющая последовательность генерируется в соответствии с заданным алгоритмом и с учетом значений логических условий x, формируемых ОА.

Часто операционное устройство может выполнять несколько различных операций (например, арифметико-логическое устройство может выполнять че-

тыре арифметических действия и несколько логических операций над входными словами). В этом случае на вход УА поступает команда C, определяющая тип выполняемой операции. Кроме того, поскольку различные операции над различными данными выполняются за разное время, УА формирует сигнал g, отмечающий окончание операции и готовность выходных данных.

Таким образом, любое операционное устройство — процессор (который обычно, в свою очередь, представляют состоящим из двух операционных устройств: АЛУ — арифметико-логического устройства и ЦУУ — центрального устройства управления), канал ввода/вывода, контроллер внешнего устройства — можно представить как композицию *операционного* и *управляющего автоматов*. Операционный автомат, реализуя действия над словами информации, является исполнительной частью устройства, работу которого организует управляющий автомат, генерирующий необходимые последовательности управляющих сигналов.

Такой подход позволяет разработать эффективные процедуры синтеза ОА и УА, формализовать эти процедуры и, в некоторых случаях, автоматизировать процесс синтеза цифровых устройств.

4.3. Операционный автомат

Исходным для разработки структуры операционного автомата (ОА) являются:

- описание входных и выходных слов ОА (множеств D_I и D_O);
- список множества операций из F, которые должны выполняться над словами.

Процесс разработки ОА, таким образом, следует начинать с определения *форматов входных* и *выходных слов* и разработки алгоритмов выполнения операций в терминах слов и стандартных действий над словами (сложение, копирование, инверсия, сдвиг и т. д.). Разработанные алгоритмы удобно представить в форме *граф-схемы алгоритма* (ГСА).

Далее необходимо разработать *структуру* ОА. Операционный автомат строится на базе операционных и логических элементов. Предложенные процедуры формального синтеза ОА [7] не получили широкого распространения; обычно используют т. н. "содержательный" метод синтеза.

Разработать структуру — значит *определить набор элементов*, входящих в нее, и *установить связи* между этими элементами. Структура реализуется, исходя из разработанных на предыдущем этапе алгоритмов таким образом,

чтобы обеспечить реализацию всех действий, предусмотренных в операторных вершинах ГСА.

Действия в структуре ОА выполняются под управлением микроопераций, поэтому при разработке ОА следует определить *полный список микроопераций*, наличие которых обеспечит выполнение в разработанной структуре всех предусмотренных в алгоритмах преобразований слов.

Наконец, формирование последовательности микроопераций в управляющем автомате осуществляется с учетом значений логических условий, которые формируются в ОА. Поэтому при разработке ОА следует сформировать *список логических условий*, определяемый содержимым условных вершин ГСА, и предусмотреть в структуре ОА (если это необходимо) специальные элементы для формирования этих логических условий.

Итак, процесс разработки ОА можно представить состоящим из следующих этапов:

1. Определение форматов входных и выходных данных (слов).
2. Разработка ГСА выполняемых операций.
3. Разработка структуры ОА — выбор элементов и организация связей.
4. Определения множества $\{y\}$ микроопераций, выполняемых в ОА.
5. Определения множества $\{x\}$ логических условий, формируемых в ОА.

4.3.1. Пример проектирования операционного автомата АЛУ

В качестве примера рассмотрим разработку операционного автомата арифметического устройства, реализующего операцию деления чисел с фиксированной запятой, представленных в прямом коде.

Определение форматов данных

Будем считать, что в арифметической операции деления участвуют операнды A — делимое и B — делитель. Результат операции C — частное. Кроме того, устройство должно формировать признаки результата — *двоичные переменные*:

- Z — признак нулевого результата;
- S — признак отрицательного результата;
- OV — признак переполнения.

Алгоритм операции алгебраического деления разрабатываются для 16-разрядных двоичных чисел с фиксированной запятой, представленных в *прямом*

коде. Знак числа кодируется в старшем (нулевом) разряде числа, запятая фиксирована после знакового разряда, таким образом, все числа могут быть только дробными (рис. 4.2).

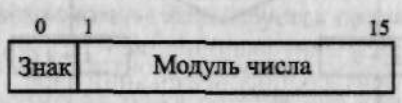


Рис. 4.2. Представление числа в прямом коде

Итак, в операциях участвуют следующие переменные:

- $A = a_0 a_1 a_2 ... a_{15}$ — первый операнд (делимое);
- $B = b_0 b_1 b_2 ... b_{15}$ — второй операнд (делитель);
- $C = c_0 c_1 c_2 ... c_{15}$ — результат операции (частное), в процессе выполнения алгоритма переменная C используется для хранения остатка;
- $D = d_0 d_1 d_2 ... d_{15}$ — переменная, в которой в процессе деления накапливаются цифры частного;
- a_0, b_0, c_0 — знаковые разряды.

Разработка алгоритма деления

В прямых кодах удобнее делить модули чисел. Знак результата не зависит от соотношения модулей делимого и делителя и определяется по выражению (4.1).

$$c_0 = a_0 \overline{b}_0 \vee \overline{a}_0 b_0. \qquad (4.1)$$

Деление чисел с фиксированной запятой в заданном формате невозможно, если модуль делимого не меньше модуля делителя. Поэтому сначала следует проверить соотношение операндов путем вычитания делителя из делимого. Если разность окажется положительной, то можно формировать признак переполнения $OV = 1$ и завершать операцию. В противном случае модуль частного оказывается меньше 1, т. е. переполнение отсутствует и деление возможно.

Алгоритмы деления *с восстановлением остатка* и *без восстановления остатка* подробно рассмотрены в *разд. 3.9*. Учитывая приведенный там сравнительный анализ алгоритмов, выберем метод деления без восстановления остатка.

ГСА деления *без восстановления остатка* представлена на рис. 4.3. Алгоритм предусматривает формирование знака результата согласно формуле (4.1) и сохранение его временно в переменной s. После этого производится деление модулей чисел (знаки операндов обнуляются).

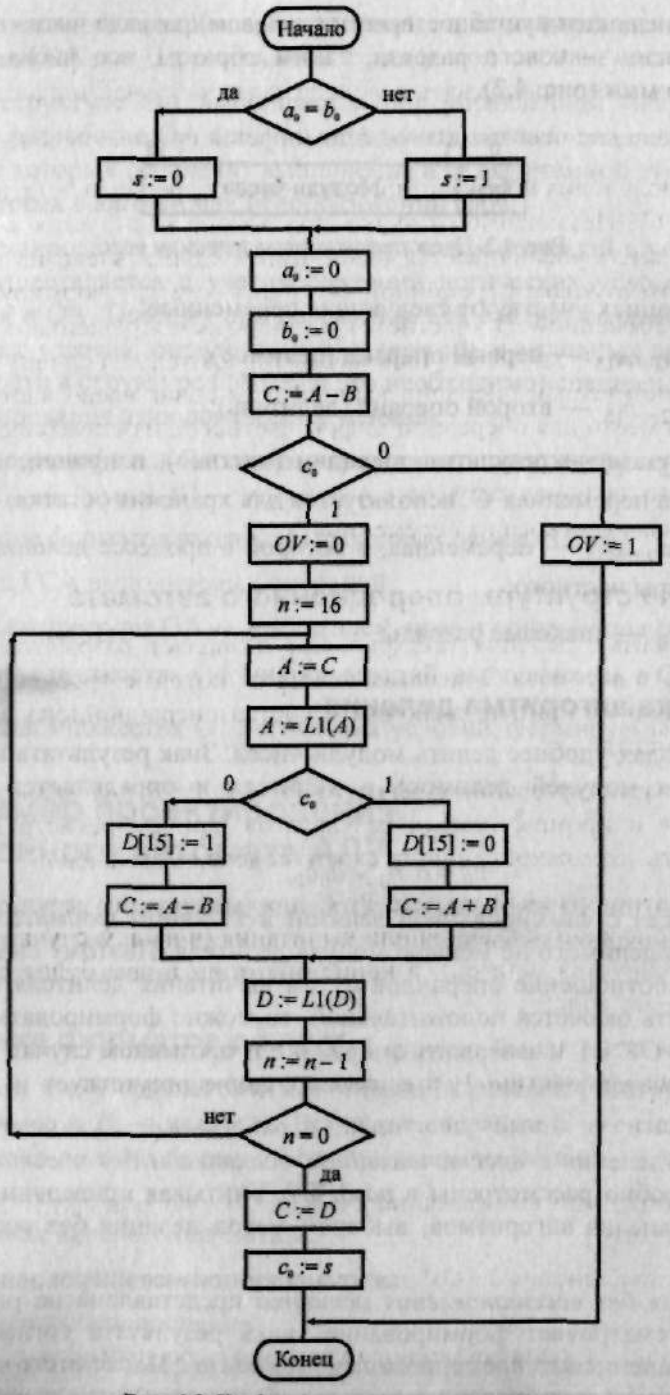


Рис. 4.3. Граф-схема алгоритма деления

Сначала производится пробное вычитание делителя из делимого. Поскольку знаки операндов — 0, то появление 1 в знаковом разряде разности означает, что $A < B$, и можно продолжать деление (целая часть частного равна 0).

При $c_0 = 0$ деление невозможно — формируется признак переполнения.

В процессе получения цифр частного значение очередного остатка принимает переменная C. Независимо от знака остатка она копируется в переменную A, которая затем увеличивается вдвое путем сдвига влево на один разряд. В зависимости от знака переменной C (знака остатка) формируется очередная цифра переменной D (частного) и принимается решение о действии на следующем шаге — добавлять или вычитать делитель из сдвинутого остатка. После арифметической операции выполняется сдвиг влево частного D (освобождается место для очередной цифры частного), изменяется счетчик цифр частного и проверяется условие выхода из цикла — получение шестнадцати цифр частного, включая самую первую цифру — "0 целых", на место которой копируется знак частного из переменной s.

Разработка структуры операционного автомата

Анализ алгоритма деления (см. рис. 4.3) позволяет разработать структуру операционного автомата. Учитывая действия, которые требуется выполнить для реализации алгоритма, включим в состав операционного автомата следующие элементы:

- два шестнадцатиразрядных регистра Рг A и Рг B для хранения входных операндов и промежуточных результатов, причем регистр Рг A должен обеспечить возможность сдвига своего содержимого влево;
- шестнадцатиразрядный регистр Рг C для размещения результата арифметической операции сложения или вычитания (в нашем случае в этом регистре формируется остаток): в конце операции в нем будет размещен результат — частное;
- шестнадцатиразрядный регистр Рг D с возможностью левого сдвига кода для размещения частного в процессе его формирования;
- шестнадцатиразрядный двоичный параллельный сумматор/вычитатель Сум/Выч;
- четырехразрядный вычитающий счетчик Сч n по модулю 16 для подсчета цифр частного;
- триггер переполнения Тг OV для хранения признака переполнения разрядной сетки;
- триггер знака Тг s для временного хранения знака частного;
- схема сравнения на "равно" знаковых разрядов исходных операндов;

- дешифратор DC "0" нулевой комбинации в разрядах $C[1:15]$, формирующий признак нулевого результата Z.

Связи между перечисленными выше элементами, а также управляющие ими *микрооперации* показаны на рис. 4.4, а в табл. 4.1 приведен полный список микроопераций и логических условий.

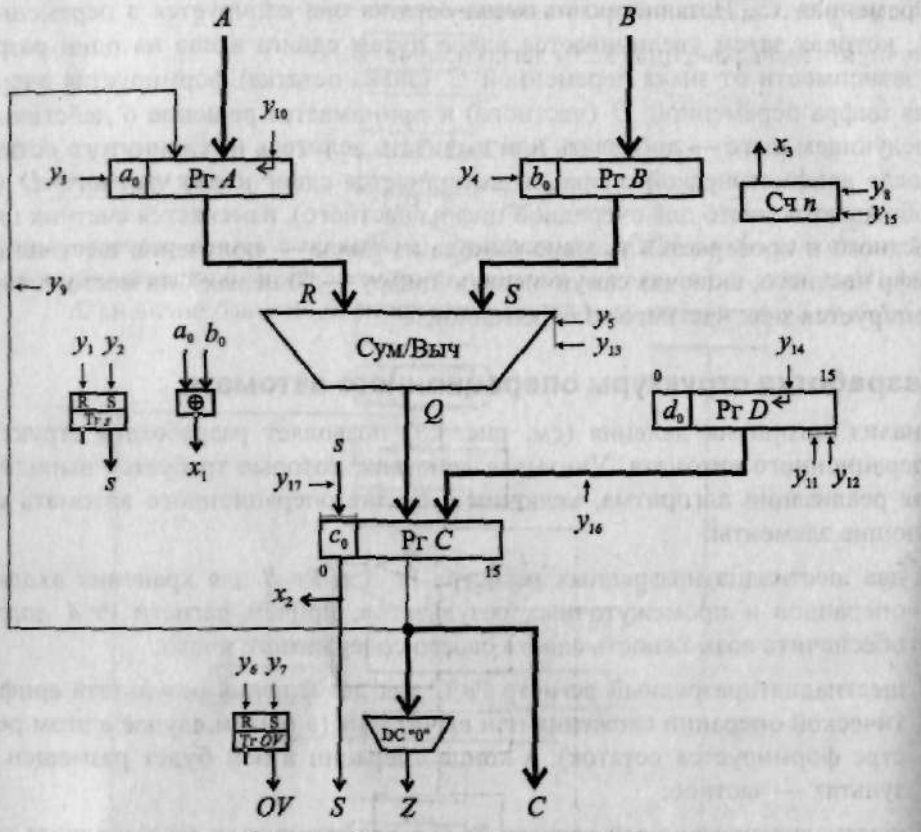


Рис. 4.4. *Операционный автомат АЛУ*

Таблица 4.1. Список микроопераций и логических условий

Микро-операция	Действие	Микро-операция	Действие	Логическое условие	Действие
y_1	$s := 0$	y_{10}	$A := LI(A)$	x_1	$a_0 := b_0$
y_2	$s := 1$	y_{11}	$D[15] := 1$	x_2	c_0
y_3	$a_0 := 0$	y_{12}	$D[15] := 0$	x_3	Сч $n := 0$

Таблица 4.1 *(окончание)*

Микро-операция	Действие	Микро-операция	Действие	Логическое условие	Действие
y_4	$b_0 := \overline{0}$	y_{13}	$C := A + B$		
y_5	$C := R + S$	y_{14}	$D := Л(D)$		
y_6	$OV := 0$	y_{15}	Сч $n :=$ Сч $- 1$		
y_7	$OV := 1$	y_{16}	$C := D$		
y_8	$n := 16$	y_{17}	$c_0 := s$		
y_9	$A := C$				

Внимательно посмотрим на рис. 4.4. Очевидно, любые действия, обозначенные в операторных вершинах алгоритма, приведенного на рис. 4.3, могут быть реализованы на разработанной нами структуре (см. рис. 4.4).

Теперь определим, какая последовательность *микроопераций* должна быть реализована в разработанной структуре, чтобы выполнилась операция деления, предусмотренная алгоритмом рис. 4.3. Простейшее решение — сохранить топологию графа алгоритма и заменить содержимое его операторных вершин на соответствующие микрооперации, а содержимое условных вершин — на соответствующие логические условия.

Полученный таким образом граф принято называть *микропрограммой* и рассматривать в качестве исходных данных при проектировании *управляющего (микропрограммного) автомата*. При этом содержимое операторной вершины графа соответствует *действиям*, выполняемым устройством в один такт дискретного времени.

При проектировании цифровых устройств обычно стремятся достичь максимальной скорости их работы. Один из путей достижения этой цели — параллельное (во времени) выполнение некоторых операций. Поэтому при преобразовании графа алгоритма в граф микропрограммы следует объединять в одной операторной вершине те микрооперации, которые могут быть в данной структуре выполнены одновременно с учетом реализуемого алгоритма. Совокупность микроопераций, выполняемых одновременно в один такт дискретного времени, называется *микрокомандой*.

Например, анализируя ГСА рис. 4.3, можно отметить, что операторы $a_0 := 0$; $b_0 := 0$ можно выполнить в структуре, изображенной на рис. 4.4, одновременно. То же можно сказать о паре операторов $D := Л(D)$; $n := n - 1$ и некоторых других. В то же время, операторы $A := C$, $A := Л(A)$ нельзя выполнять

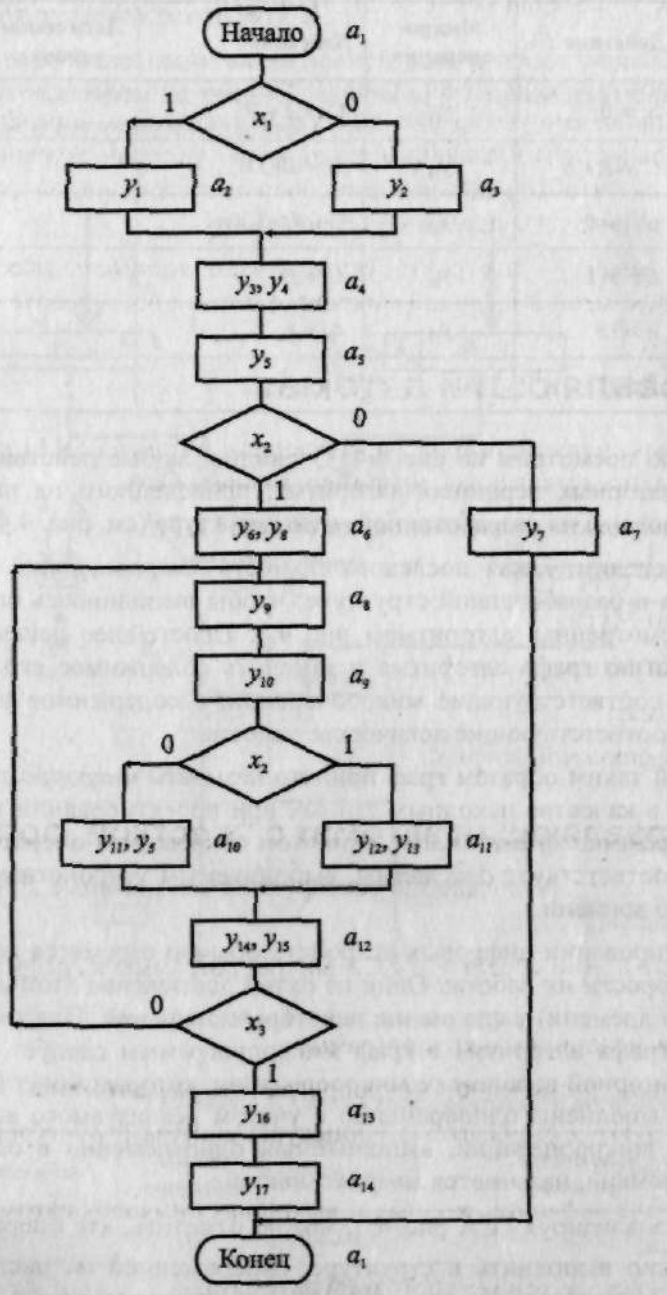


Рис. 4.5. Микропрограмма деления

одновременно. (Для ускорения этой процедуры можно передавать информацию из C в A со сдвигом: $C := L1(A)$, но это уже будет другая структура ОА.)

Проанализировав с этой точки зрения исходный алгоритм, получим микропрограмму, приведенную на рис. 4.5. Микропрограмма определяет, в какой последовательности и в зависимости от каких условий должны выдаваться *микрокоманды*, чтобы реализовалась операция деления на разработанной структуре (см. рис. 4.4) операционного автомата.

Следующая задача — построить *управляющий автомат*, обеспечивающий выдачу микрокоманд в заданной микропрограммой последовательности.

4.4. Управляющий автомат

Исходным для проектирования управляющего автомата (УА) является микропрограмма, представленная, например, в форме ГСА.

Различают два класса управляющих автоматов [2, 7]:

- с "жесткой" логикой:
  - автомат Мура;
  - автомат Мили;
  - С-автомат;
- с программируемой логикой.

4.4.1. Управляющий автомат с "жесткой" логикой

Автоматы с "жесткой" логикой проектируются как обычные конечные структурные автоматы [2, 7, 8].

Сначала необходимо перейти от ГСА микропрограммы к графу автомата, для чего следует:

1. Разметить исходную микропрограмму.
2. Построить по размеченной микропрограмме граф автомата.

Далее реализуются стандартные процедуры синтеза структурного автомата, заданного графом:

- кодирование алфавита входных и выходных символов автомата двоичными кодами;
- кодирование внутренних состояний автомата;
- выбор элемента памяти (типа триггера);

- построение автоматной таблицы переходов;
- синтез комбинационной схемы, реализующей функцию переходов КСх 1;
- синтез комбинационной схемы, реализующей функцию выходов КСх 2.

Процедура *разметки микропрограммы* ставит в соответствие символам состояний автомата (a_1, a_2, ..., a_M) некоторые объекты микропрограммы. Способы разметки микропрограмм различаются для автоматов различных типов.

Для автомата Мура разметка выполняется по следующим правилам [2]:

- символом a_1 отмечается начальная и конечная вершина ГСА;
- различные *операторные* вершины отмечаются разными символами состояний;
- все операторные вершины должны быть отмечены.

Для автомата Мили разметка выполняется по следующим правилам [2]:

- символом a_1 отмечается вход вершины, следующей за начальной, а также вход конечной вершины;
- *входы всех вершин, следующих за операторными*, должны быть отмечены символами состояний;
- если вход вершины отмечается, то лишь одним символом;
- входы различных вершин, за исключением конечной, отмечаются различными символами.

Пример проектирования УАЖЛ

Рассмотрим пример построения *управляющего автомата Мура* для устройства, реализующего операцию деления. Операционный автомат (см. рис. 4.4) и ГСА микропрограммы (см. рис. 4.5) этого устройства были построены ранее.

Шаг 1. Выполним *разметку микропрограммы*. Для этого сопоставим каждой операторной вершине ГСА в произвольном порядке (например, слева направо и сверху вниз) символ состояния автомата из множества $\{a_2, a_3, ..., a_{14}\}$. Начальную и конечную вершины сопоставим с начальным состоянием автомата a_1. Такая разметка показана на рис. 4.5.

Шаг 2. Построим *граф автомата*, заданного размеченной микропрограммой, которую получили на предыдущем шаге. Для этого вершины графа сопоставим с состояниями автомата a_1, a_2, ..., a_{14}. Соединим ориентированными ребрами те пары вершин графа, между которыми на ГСА микропро-

граммы существуют переходы, причем пометим ребра графа соответствующими условиями перехода. Если переход между двумя операторными вершинами микропрограммы осуществляется безусловно, то условие перехода на ребре графа — константа 1.

Построив таким образом граф, мы фактически задаем алфавиты внутренних состояний и входных символов и определяем функцию переходов. Для задания алфавита выходных символов и функции выходов (для автомата Мура функция выходов зависит только от его состояний) следует сопоставить каждой вершине автомата в качестве выходного символа содержимое соответствующей операторной вершины ГСА микропрограммы. Таким образом, получим граф микропрограммного автомата, который приведен на рис. 4.6.

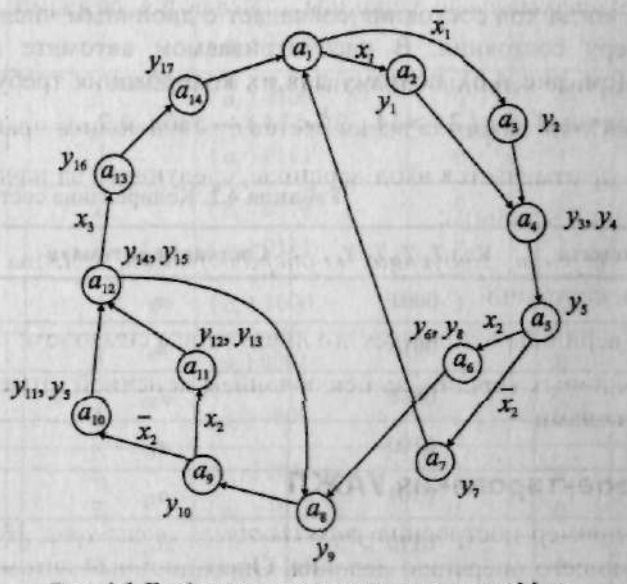


Рис. 4.6. Граф микропрограммного автомата Мура

Шаг 3. *Кодирование алфавитов входных* и *выходных символов* автомата двоичными кодами. Алфавит входных символов составляет множество *двоичных переменных* $X = \{x_1, x_2, x_3\}$, поэтому проблема кодирования входных символов двоичными переменными здесь не стоит. Что касается кодирования символов выходного алфавита, то отложим обсуждение этого вопроса до шага 8.

Шаг 4. В процессе *кодирования внутренних состояний автомата* могут решаться проблемы исключения гонок в автомате, проблемы минимизации комбинационной схемы, обеспечивающей функцию переходов автомата. Для решения этих задач разработаны достаточно сложные алгоритмы, которые описаны в литературе, например, [2, 5]. Здесь мы не будем касаться этой стороны процедуры синтеза автомата.

Следует отметить, что проблемы гонок могут быть полностью решены при использовании в автомате двухтактных элементов памяти, причем способ кодирования состояний в этом случае роли не играет. Правда, затраты оборудования при таком решении несколько возрастают, по сравнению с использованием метода противогоночного кодирования, но во-первых, эффективность метода заметно проявится лишь при достаточно большом числе состояний автомата (25—40), а во-вторых, большинство современных интегральных элементов памяти (триггеров) выпускаются именно двухтактными. Применение специальных методов кодирования "с учетом сложности комбинационных схем" [2] так же обеспечивает заметный эффект лишь для достаточно громоздких автоматов.

В нашем примере воспользуемся простейшим методом кодирования состояний автомата, когда код состояния совпадает с двоичным числом, соответствующим номеру состояния. В рассматриваемом автомате насчитывается 14 состояний (см. рис. 4.6), поэтому для их кодирования требуется четырехразрядный двоичный код ($2^4 > 14$; $2^3 < 14$) — табл. 4.2.

Таблица 4.2. Кодирование состояний автомата

Состояние автомата	Код $T_1\,T_2\,T_3\,T_4$	Состояние автомата	Код $T_1\,T_2\,T_3\,T_4$
a_1	0001	a_8	1000
a_2	0010	a_9	1001
a_3	0011	a_{10}	1010
a_4	0100	a_{11}	1011
a_5	0101	a_{12}	1100
a_6	0110	a_{13}	1101
a_7	0111	a_{14}	1110

Шаг 5. *Выбор элемента памяти* (типа триггера). При выборе элемента памяти следует учитывать простоту управления им. С этой точки зрения удобнее выбирать триггеры, управляемые по единственному информационному входу — к таким относятся D- и T-триггеры. В нашем примере в качестве элемента памяти автомата выберем синхронный двухтактный D-триггер. Очевидно, для реализации нашего автомата понадобятся четыре D-триггера.

Шаг 6. Построение *автоматной таблицы переходов* (табл. 4.3).

Эта таблица практически описывает функцию переходов автомата, строится по графу автомата и определяет, какие значения необходимо подать на

управляющие входы триггеров на каждом переходе автомата в новое состояние. Строка таблицы соответствует одному переходу автомата. Таким образом, автоматная таблица содержит столько строк, сколько ребер в графе автомата (включая и петли, если они имеются в графе).

Таблица 4.3. Автоматная таблица переходов

Исходное состояние	Условие перехода	Состояние перехода	Функции возбуждения			
			D_1	D_2	D_3	D_4
(a_1) 0001	x_1	(a_2) 0010	0	0	1	0
	$\overline{x}_1$	(a_3) 0011	0	0	1	1
(a_2) 0010	1	(a_4) 0100	0	1	0	0
(a_3) 0011	1	(a_4) 0100	0	1	0	0
(a_4) 0100	1	(a_5) 0101	0	1	0	1
(a_5) 0101	x_2	(a_6) 0110	0	1	1	0
	$\overline{x}_2$	(a_7) 0111	0	1	1	1
(a_6) 0110	1	(a_8) 1000	1	0	0	0
(a_7) 0111	1	(a_1) 0001	0	0	0	1
(a_8) 1000	1	(a_9) 1001	1	0	0	1
(a_9) 1001	x_2	(a_{11}) 1011	1	0	1	1
	$\overline{x}_2$	(a_{10}) 1010	1	0	1	0
(a_{10}) 1010	1	(a_{12}) 1100	1	1	0	0
(a_{11}) 1011	1	(a_{12}) 1100	1	1	0	0
(a_{12}) 1100	x_3	(a_{13}) 1101	1	1	0	1
	$\overline{x}_3$	(a_8) 1000	1	0	0	0
(a_{13}) 1101	1	(a_{14}) 1110	1	1	1	0
(a_{14}) 1110	1	(a_1) 0001	0	0	0	1

Шаг 7. Синтез комбинационной схемы, реализующей *функцию переходов* автомата. Эта комбинационная схема в нашем случае реализует четыре булевы функции D_1, D_2, D_3, D_4, зависящие от четырехразрядного двоичного

кода состояния автомата $T_1T_2T_3T_4$ и трехбитного вектора входных символов $x_1x_2x_3$. Комбинационная схема, описываемая системой четырех булевых функций от семи переменных, должна обеспечивать переходы автомата в соответствии с графом рис. 4.6.

Для построения этой схемы можно построить четыре карты Карно для D_1, D_2, D_3, D_4 по таблицам истинности, заданным соответствующими столбцами автоматной таблицы переходов, и минимизировать эти функции. Менее трудоемкий способ состоит в предварительной дешифрации состояний автомата (булевы функции четырех переменных) и записи функций возбуждения через значения a_i и x_k. Воспользуемся последним способом, тем более, что дешифрированные состояния автомата пригодятся нам и на следующем шаге при формировании функции выходов.

Итак, предусмотрим дешифратор, на входы которого поступает двоичный код состояния автомата $\tilde{T}_1\tilde{T}_2\tilde{T}_3\tilde{T}_4$, а на выходах формируется унитарный код $\bar{a}_1...\bar{a}_{i-1}a_i\bar{a}_{i+1}...\bar{a}_{14}$. Кстати, поскольку на входах дешифратора не могут появиться комбинации 0000 и 1111 (их мы не использовали при кодировании состояний), то схему дешифратора можно минимизировать. (Как? Попробуйте построить такой дешифратор самостоятельно.) Рассмотрим столбец D_1 автоматной таблицы переходов, отметив те наборы входных переменных функции возбуждения (из $\{a\}$ и $\{x\}$), на которых D_1 принимает единичные значения. Можно записать:

$$D_1 = a_6 \vee a_8 \vee a_9 \vee a_{10} \vee a_{11} \vee a_{12} \vee a_{13}. \qquad (4.2)$$

Обратите внимание, что функция D_1 не зависит от входных переменных $\{x\}$ (частный случай!). Действительно, переход, например, из состояния a_9 в зависимости от значения x_2 может произойти в a_{10} или в a_{11}, но в обоих этих состояниях значение старшего разряда кодов одинаково. То же для D_1 можно сказать и обо всех остальных переходах, зависящих от входных символов (из a_1, a_5, a_{12}).

Теперь запишем булевы выражения для остальных функций возбуждения.

$$D_2 = a_2 \vee a_3 \vee a_4 \vee a_5 \vee a_{10} \vee a_{11} \vee a_{12}x_3 \vee a_{13}. \qquad (4.3)$$

$$D_3 = a_1 \vee a_5 \vee a_9 \vee a_{13}. \qquad (4.4)$$

$$D_4 = a_1\bar{x}_1 \vee a_4 \vee a_5\bar{x}_2 \vee a_7 \vee a_8 \vee a_9x_2 \vee a_{12}x_3 \vee a_{14}. \qquad (4.5)$$

Шаг 8. Синтез комбинационной схемы, реализующей *функцию выходов*. Функция выходов автомата Мура зависит только от его внутреннего состояния и задана непосредственно на графе автомата (см. рис. 4.6). Выходами микропрограммного автомата являются *микрооперации*, поступающие в точки управления операционного автомата. Поэтому выходные символы микропрограммного автомата обычно не кодируют, а формируют на выходе значение вектора микроопераций. При большом числе микроопераций с целью уменьшения связности между ОА и УА на вход ОА передают не вектор микроопераций, а номер активной в данном такте микрооперации. Подробнее об этом — в следующем разделе.

Из графа автомата (см. рис. 4.6) видно, что микрооперация y_1 должна вырабатываться автоматом, когда он находится в состоянии a_2, микрооперация y_2 — в состоянии a_3 и т. д. Микрооперация y_5 должна вырабатываться автоматом, находящимся в состоянии a_5 и a_{10}. Запишем функцию выходов автомата в виде системы булевых функций[1]:

$$y_1 = a_2, y_2 = a_3, y_3 = a_4, y_4 = a_4, y_5 = a_5 \vee a_{10}, y_6 = a_6, y_7 = a_7,$$
$$y_8 = a_6, y_9 = a_8, y_{10} = a_9, y_{11} = a_{10}, y_{12} = a_{11}, y_{13} = a_{11}, y_{14} = a_{12}, \qquad (4.6)$$
$$y_{15} = a_{12}, y_{16} = a_{13}, y_{17} = a_{14}.$$

Шаг 9. Теперь изобразим *функциональную схему управляющего автомата*, используя выражения (4.2)—(4.6) с учетом выбранного типа элемента памяти.

Управляющий микропрограммный автомат с жесткой логикой (автомат Мура), изображенный на рис. 4.7, имеет три двоичных входа x_1, x_2, x_3, вход тактового сигнала CLK и семнадцать двоичных выходов — микрооперации $y_1, y_2, \ldots, y_{17}$.

Память автомата представлена четырьмя двухтактными синхронными D-триггерами, объединенными общей цепью синхронизации (CLK). Выходы триггеров поступают на вход дешифратора DC "4 → 16", на выходах которого формируется унитарный код текущего состояния автомата. Поскольку проектируемый автомат является автоматом Мура, выходы дешифратора фактически являются выходами управляющего автомата (значениями микроопераций). Исключение составляет микрооперация y_5, которая формируется как

[1] Напомним, что переменные $a_1, a_2, \ldots, a_{14}$ являются булевыми, принимающими единичное значение, когда автомат находится в соответствующем состоянии, и нулевое значение — во всех остальных случаях.

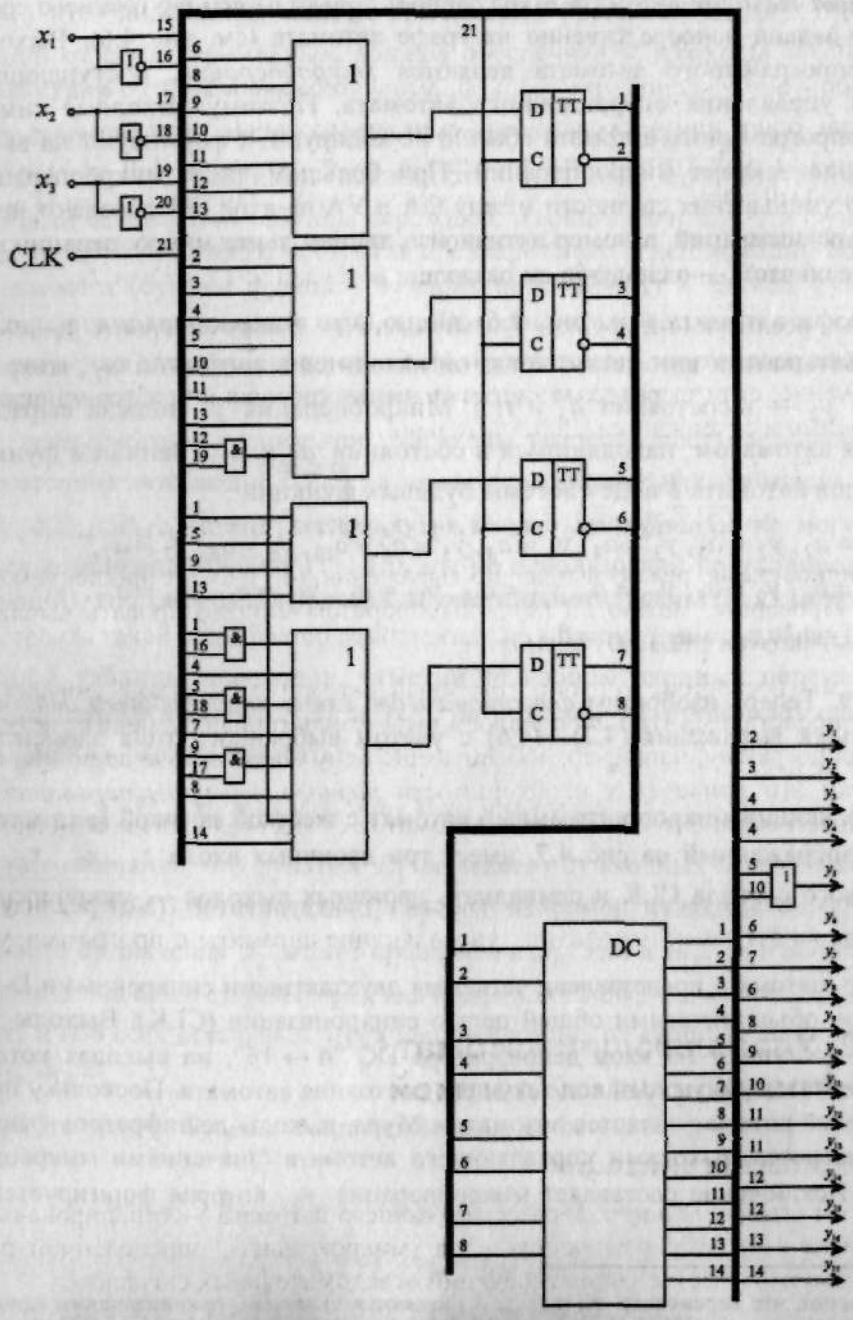


Рис. 4.7. Микропрограммный автомат Мура — функциональная схема

дизъюнкция двух различных состояний автомата, поскольку эта микрооперация присутствует в двух различных операторных вершинах исходной микропрограммы (см. рис. 4.5).

Функции возбуждения триггеров формируют значения D_i, $i \in \{1, 2, 3, 4\}$ на выходах одно- или двухуровневых схем, построенных согласно выражениям (4.2)—(4.5). Обратите внимание, что в выражениях (4.3) и (4.5) имеется одинаковый терм — $a_{12}x_3$. На функциональной схеме выход конъюнктора, реализующего этот терм, поступает на два входа различных дизъюнкторов, "обеспечивая" одновременно две функции возбуждения — D_2 и D_4.

Схемы, реализующие функции возбуждения, можно построить иначе. Для примера рассмотрим выражение (4.4). Состояния автомата, входящие в это выражение, если выразить их через значения состояний триггеров, окажутся *соседними* и склеиваются. Действительно:

$$D_3 = a_1 \vee a_5 \vee a_9 \vee a_{13} = \\ = \overline{T}_1\overline{T}_2\overline{T}_3T_4 \vee \overline{T}_1T_2\overline{T}_3T_4 \vee T_1\overline{T}_2\overline{T}_3T_4 \vee T_1T_2\overline{T}_3T_4 = \overline{T}_3T_4. \quad (4.7)$$

Очевидно, схема, реализованная по выражению (4.7), будет проще, чем схема (4.4). Проверьте, можно ли подобным образом минимизировать выражения остальных функций возбуждения.

Итак, мы построили микропрограммный автомат с "жесткой" логикой. Давайте представим, что нам понадобилось незначительно изменить исходную микропрограмму, например, добавить еще одну операторную вершину. Практически это приведет к необходимости *полного перепроектирования* всей схемы автомата. Это особенность автоматов с "жесткой" логикой является их серьезным недостатком.

Для того чтобы уменьшить зависимость структуры автомата от реализуемых им микропрограмм, используют управляющие автоматы с программируемой логикой.

4.4.2. Управляющий автомат с программируемой логикой

Принципы организации

Заметим, что функция любого управляющего автомата — генерирование последовательности управляющих слов (микрокоманд), определенной реализуемым алгоритмом с учетом значений осведомительных сигналов.

Если заранее разместить в запоминающем устройстве все необходимые для реализации заданного алгоритма (группы алгоритмов) микрокоманды, а по-

том выбирать их из памяти в порядке, предусмотренном алгоритмом (с учетом значения осведомительных сигналов), то получим управляющий автомат, структура которого слабо зависит от реализуемых алгоритмов, а поведение в основном определяется *содержимым запоминающего устройства*.

При изменениях реализуемого алгоритма в достаточно широких пределах структура такого автомата не меняется, достаточно лишь изменить содержимое ячеек запоминающего устройства. Управляющий микропрограммный автомат, построенный по таким принципам, называется управляющим автоматом с *программируемой логикой*.

Структурная схема такого автомата в самом общем виде приведена на рис. 4.8. Автомат включает в себя *запоминающее устройство микрокоманд* (обычно реализуемое как ПЗУ), *регистр микрокоманд* и *устройство формирования адреса микрокоманды*.

В каждом такте дискретного времени из памяти микрокоманд считывается одна микрокоманда, которая помещается в регистр микрокоманд. Микрокоманда содержит два поля (две группы полей), одно из которых определяет *набор микроопераций*, которые в данном такте поступают в операционный автомат, а другое содержит информацию для определения *адреса следующей микрокоманды*.

При проектировании управляющего автомата с программируемой логикой (УАПЛ) необходимо выбрать формат (форматы) микрокоманд (микрокоманды), способы кодирования микроопераций и адресации микрокоманд.

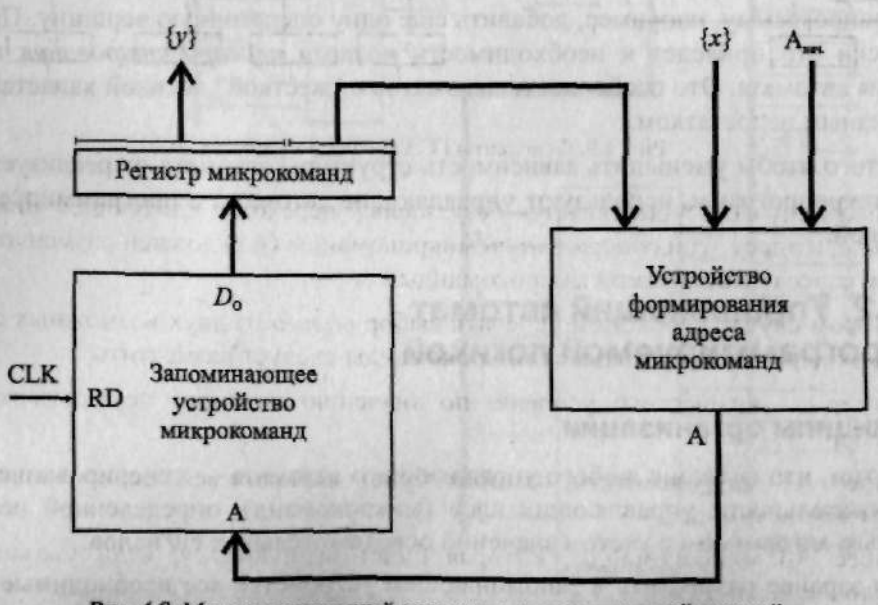


Рис. 4.8. Микропрограммный автомат с программируемой логикой

Адресация микрокоманд

Исходными данными для проектирования УАПЛ является, как и для УАЖЛ (*управляющий автомат с "жесткой" логикой*), микропрограмма, представленная, например, в форме ГСА. Каждая операторная вершина должна реализоваться в один такт машинного времени, причем после операторной вершины в ГСА может следовать:

- операторная вершина;
- условная вершина, оба выхода которой или один из них соединены с операторными вершинами, например, как показано на рис. 4.9, *а*;
- цепочка из двух или более условных вершин, выходы которых соединены с условными вершинами (рис. 4.9, *б*).

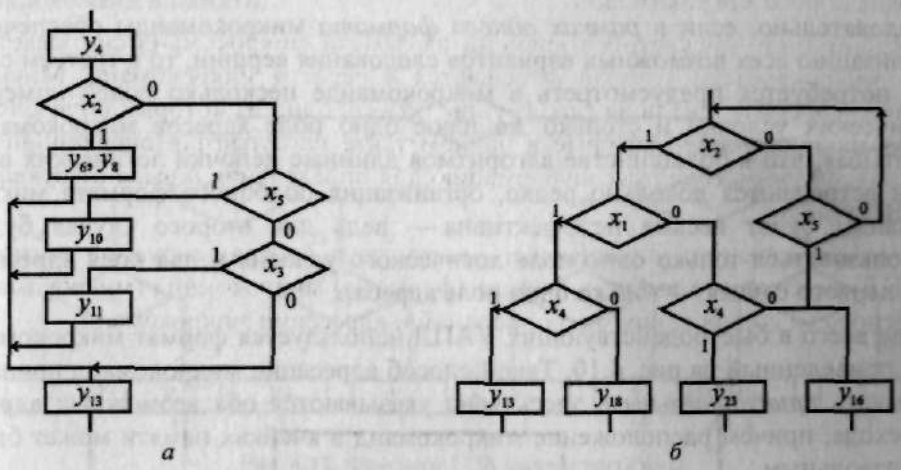


Рис. 4.9. Фрагменты ГСА микропрограмм

В первом случае осуществляется *безусловный* переход к следующей микрокоманде, и адрес этой *единственной* микрокоманды (A1) должен размещаться в поле адреса выполняемой микрокоманды.

Во втором случае необходимо сделать выбор *одного* из двух возможных следующих адресов. В поле адреса микрокоманды следует разместить:

- номер *x* логического условия, по значению которого осуществляется выбор;
- адрес A1 микрокоманды, которая будет выполняться, если указанное условие истинно;
- адрес A0 микрокоманды, которая будет выполняться, если указанное условие ложно.

В третьем случае количество проверяемых в микрокоманде условий и адресов переходов может быть произвольным, в т. ч. и достаточно большим. В этом случае длина микрокоманды может быть весьма велика.

При выборе формата микрокоманды нужно учитывать следующие обстоятельства:

- следует эффективно использовать разряды поля, обеспечив по возможности *его минимальную длину*;
- желательно ограничиться *единственным* форматом микрокоманды, в крайнем случае, выбирать минимально возможное разнообразие форматов.

Справедливость первого требования очевидна. Относительно второй предпосылки можно заметить, что при большом числе форматов соответственно усложняется схема декодирования микрокоманды.

Следовательно, если в *рамках одного формата* микрокоманды обеспечить реализацию всех возможных вариантов следования вершин, то в третьем случае потребуется предусмотреть в микрокоманде несколько полей номеров логических условий и столько же плюс одно поле адресов микрокоманд. Учитывая, что в большинстве алгоритмов длинные цепочки логических вершин встречаются довольно редко, организация подобного формата микрокоманды будет весьма неэффективна — ведь для второго случая будут использоваться только одно поле логического условия и два поля адреса, а для первого случая — только одно поле адреса.

Чаще всего в быстродействующих УАПЛ используется формат микрокоманды, приведенный на рис. 4.10. Такой способ адресации микрокоманд принято называть *принудительным*, здесь явно указываются оба возможных адреса перехода, причем расположение микрокоманд в ячейках памяти может быть произвольным.

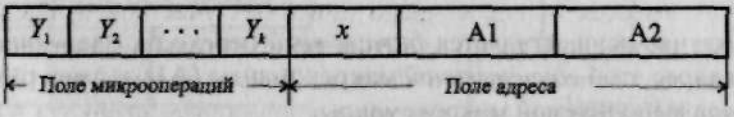


Рис. 4.10. Формат микрокоманды при принудительной адресации

Если в ГСА встречаются цепочки из k последовательных условных вершин, последняя из которых связана с операторной вершиной, то для их реализации в рамках формата микрокоманды, представленной на рис. 4.10, потребуется k микрокоманд (а следовательно, k тактов), в каждой из которых, кроме последней, поле микроопераций будет пустым. При выполнении таких микрокоманд в операционном автомате никаких действий не выполняется (он "простаивает"!), а управляющий автомат тратит $k-1$ тактов на определение следующего адреса.

Когда в ГСА встречается большое число таких цепочек, снижение производительности системы становится существенным. В этом случае используют другие подходы к формированию следующего адреса микрокоманды. Например, на рис. 4.11 представлен фрагмент ГСА, при реализации которого необходимо последовательно проверить три логических условия. Для быстрой реализации этого фрагмента достаточно выбрать некоторый базовый адрес (в данном случае он должен быть кратным 8), начиная с которого в ПЗУ микропрограмм расположить *последовательно* блок микрокоманд $y_{13}...y_{20}$. Для определения адреса очередной микрокоманды достаточно к базовому адресу прибавить (приписать справа) значение вектора логических условий $\tilde{x}_1\tilde{x}_2\tilde{x}_3$. Адрес следующей микрокоманды определяется, таким образом, за один такт, но разработчик лишается возможности произвольного размещения микрокоманд в памяти.

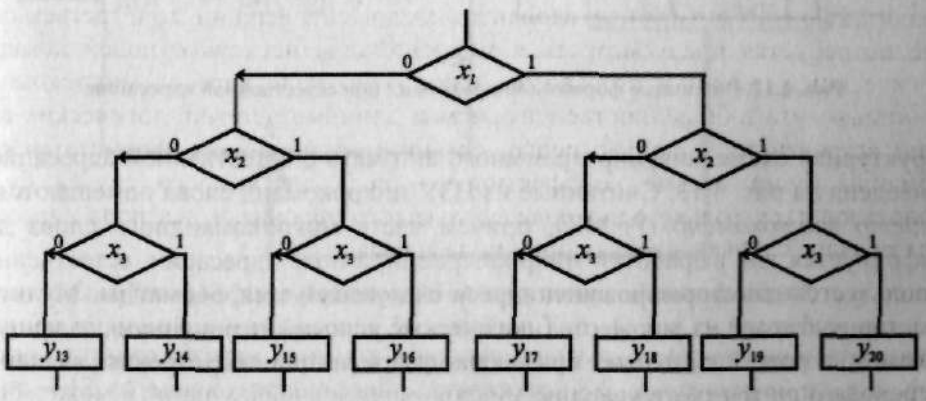


Рис. 4.11. Фрагмент ГСА микропрограммы

Если в ГСА имеется большое число линейных участков и, следовательно, безусловных переходов, то применение принудительной адресации ведет к неэффективному использованию памяти. Действительно, при безусловном переходе информативным является только поле адреса перехода А1, а поля x и А0 — не используются.

В целях уменьшения длины поля адреса микрокоманды можно использовать т. н. *естественную адресацию* (рис. 4.12), напоминающую механизм адресации команд в программе. В этом случае в состав устройства формирования адреса микрокоманды включают *регистр-счетчик адреса микрокоманд*, а в адресном поле микрокоманды — два поля: x и А1. Если заданное полем x условие истинно, то выполняется микрокоманда по адресу А1, иначе — микрокоманда по текущему значению счетчика адреса микрокоманд, предварительно увеличенному на единицу.

Таким образом, УАПЛ с естественной адресацией работает с той же скоростью, что и УАПЛ с принудительной адресацией, при этом длина микрокоманды уменьшается на длину одного адресного поля, зато в структуре УФАМК необходимо дополнительно предусмотреть регистр-счетчик.

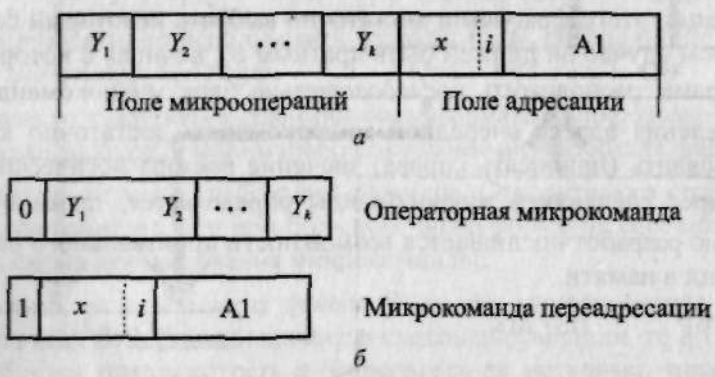


Рис. 4.12. Различные форматы микрокоманд при естественной адресации

Структурная схема микропрограммного автомата с естественной адресацией приведена на рис. 4.13. Считанные из ПЗУ микрокоманд слова помещаются в *регистр микрокоманд* (Рг МК), причем часть микрокомандного слова дешифрируется для выработки микроопераций, а поле адресации, естественно, используется для формирования адреса следующей микрокоманды. Мультиплексор выбирает из множества логических условий переменную, заданную полем x, а поле i позволяет при необходимости проинвертировать значение выбранного логического условия:

$$x_k \oplus i = \begin{cases} x_k & \text{при } i = 0; \\ \bar{x}_k & \text{при } i = 1. \end{cases}$$

Если значение выбранного логического условия истинно, то осуществляется переход по микропрограмме: при единичном значении на выходе сумматора по модулю два в *регистр-счетчика адреса микрокоманды* (Рг Сч А МК) загружается значение поля А1 микрокоманды, содержащее адрес перехода. Если условие не выполняется (ложно), то загрузки нового адреса в Рг Сч А МК *не производится, зато его прежнее значение увеличивается на единицу*. В случае безусловного перехода (после операторной вновь следует операторная вершина) так же следует просто увеличить на единицу содержимое Рг Сч А МК. Для этого достаточно среди множества логических условий иметь *константу 0* (тождественно ложное логическое условие) и именно ее номер указывать в поле x в случае безусловного перехода.

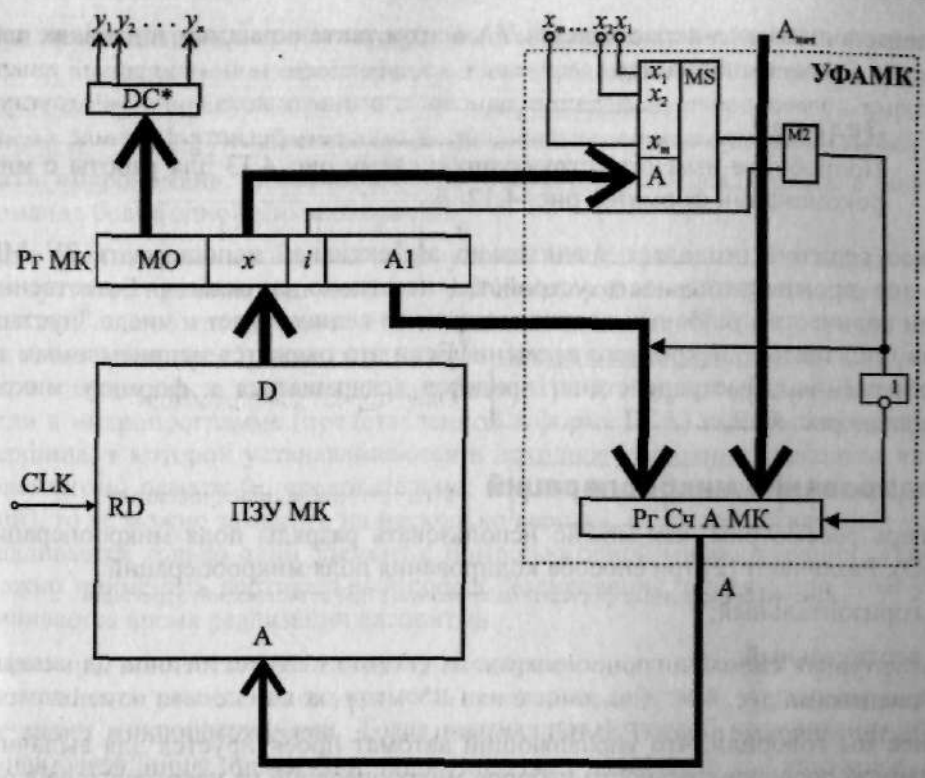


Рис. 4.13. Микропрограммный автомат с естественной адресацией

Итак, для реализации формата микрокоманды с *естественной адресацией* (рис. 4.12, *а*) можно использовать структуру УА, показанную на рис. 4.13. Однако не всегда такой формат обеспечит оптимальные характеристики управляющего автомата.

Действительно, на практике часто встречаются алгоритмы, содержащие в основном линейные участки и лишь изредка — условные вершины. Тогда для большинства микрокоманд (для тех, за которыми в ГСА следует операторная вершина) поля переадресации формата рис. 4.12, *а* использоваться не будут.

В этом случае имеет смысл введение двух различных форматов микрокоманд, показанных на рис. 4.12, *б*. Один формат содержит только информацию о микрооперациях и после выполнения такой микрокоманды всегда добавляется единица к Рг Сч А МК — таким образом реализуется линейный участок.

Команды другого формата применяются только в случае реализации условной вершины. Как обычно, проверяется значение заданного логического условия (или его инверсии), и если оно истинно — в Рг Сч А МК загружается значение поля A1 микрокоманды, иначе к Рг Сч А МК добавляется единица.

При этом никаких микроопераций УА в этом такте не выдает и никаких действий в ОА не выполняется.

Задание

Попробуйте изменить структурную схему рис. 4.13 для работы с микрокомандами форматов рис. 4.12, *б*.

Такое решение позволяет значительно эффективней использовать ЗУ МК, однако производительность устройства несколько снижается. Естественно, если количество условных вершин велико, то велико будет и число "пустых" (для ОА) тактов дискретного времени. Если это окажется неприемлемым по соображениям быстродействия, придется возвращаться к формату микрокоманды рис. 4.12, *а*.

Кодирование микроопераций

Теперь рассмотрим, как можно использовать разряды поля микроопераций (МО). Различают [2] три способа кодирования поля микроопераций:

- горизонтальный;
- вертикальный;
- смешанный.

Ранее мы говорили, что управляющий автомат проектируется для выдачи в заданной последовательности наборов микроопераций из некоторого наперед *определенного множества микроопераций* $Y = \{y_1, y_2, ..., y_n\}$.

При *горизонтальном способе* кодирования каждой микрооперации $y_i \in \{y_1, ..., y_n\}$ ставится в соответствие разряд поля микроопераций микрокомандного слова. В этом случае количество разрядов поля микроопераций N равно числу n различных микроопераций, вырабатываемых УА.

Достоинствами горизонтального способа кодирования являются:

- возможность формирования произвольных микрокоманд из заданного набора микроопераций;
- простота реализации схем формирования микроопераций, фактически — их отсутствие, т. к. выход каждого разряда поля микроопераций регистра микрокоманд является выходной линией УА — соответствующей микрооперацией.

Недостаток — чаще всего неэффективно используется память микрокоманд. Действительно, если число микроопераций УА составляет 80, а количество выполняемых в микрокоманде микроопераций — не более 6 (типичные характеристики для УА АЛУ), то в восьмидесятиразрядном поле микроопераций каждого микрокомандного слова будет не более 6 единиц.

При вертикальном способе кодирования в поле микроопераций помещается номер выполняемой микрооперации. При этом количество разрядов N, которое следует предусмотреть в поле микроопераций, определяется выражением: $N = k \geq \log_2 n$. *Достоинство* способа в экономном использовании памяти микрокоманд. *Недостаток* — в невозможности реализовать в микрокоманде более одной микрооперации.

Если реализуемые алгоритмы и структура ОА таковы, что в каждом такте дискретного времени выполняется не более одной микрооперации, то *вертикальный способ кодирования* — оптимальное решение. В иных случаях можно попытаться преобразовать исходную микропрограмму к такому виду, чтобы в каждом такте выполнялось не более одной микрооперации. Например, если в микропрограмме (представленной в форме ГСА) имеется операторная вершина, в которой устанавливаются в исходное состояние несколько ячеек (элементов) памяти (и, следовательно, она включает несколько микроопераций), то ее можно заменить на несколько вершин, в каждой из которых устанавливается только один элемент с помощью одной микрооперации. Теперь можно применить вертикальный способ кодирования, правда, при этом увеличивается время реализации алгоритма.

Однако во многих случаях структура операционного автомата не допускает возможности разнесения во времени некоторых действий, управляемых различными микрооперациями. Тогда вертикальный способ кодирования поля микроопераций не применим.

Вертикальный и горизонтальный способы кодирования — две крайности. Истина обычно лежит "посередине". Рассмотрим *смешанный способ кодирования*, идея которого состоит в следующем. Если во всех микропрограммах, реализуемых УА, нет микрокоманды с большим, чем s, числом микроопераций, то в поле микроопераций можно предусмотреть s подполей разрядностью k, в каждом из которых помещать номер нужной микрооперации. Такой способ позволяет в любой микрокоманде реализовать произвольную s-ку микроопераций, т. е. сохранить гибкость горизонтального кодирования, при возможном значительном сокращении разрядности поля микроопераций: $N = s \cdot k = s \log_2 n$. Так, для приведенного выше примера ($n = 80$, $s = 6$) определим $k = 7 \geq \log_2 80$, $N = 7 \cdot 6 = 42$ (тоже, конечно, немало), что позволит почти вдвое сократить разрядность поля микроопераций по сравнению с горизонтальным способом кодирования.

Эффективность применения смешанного кодирования существенно зависит от значения s, которое может лежать в диапазоне $1 \leq s \leq n$. При $s = 1$ имеем случай вертикального кодирования, при $s = n$ — горизонтального.

Канонический способ смешанного кодирования, идея которого представлена выше, предполагает, что каждое из s подполей микроопераций содержит k

разрядов, следовательно, в любом подполе можно закодировать любую микрооперацию $y \in Y$. Возможно, например, построение микрокоманды, содержащей s одинаковых микроопераций $y_i y_i ... y_i$, что является явно бессмысленным.

С целью сокращения разрядности полей микроопераций множество микроопераций Y разбивается на подмножества Y_1, Y_2, ..., Y_p, такие, что

$$\bigcup_{i=1}^{p} Y_i = Y; \quad \forall (Y_i \cap Y_j = \varnothing) \text{ при } i \neq j. \tag{4.8}$$

Каждое подполе поля микроопераций кодирует микрооперации только одного *подмножества* $Y_i \subset Y$. *Поскольку* $\forall |Y_i| < |Y|$, *разрядность* k_i *каждого из* подполей может быть меньше k. Очевидно, при "удачном" (пока скажем так) *распределении микроопераций по подмножествам можно будет реализовать* любую операторную вершину ГСА микропрограммы с помощью одной микрокоманды (т. е. достигнуть быстродействия, характерного для горизонтального способа кодирования), при этом значительно уменьшить разрядность поля микроопераций даже по сравнению с каноническим способом смешанного кодирования.

"Удачное" разбиение исходного множества микроопераций связано с понятием *совместимости (несовместимости)* микроопераций [7]. Некоторые из используемых в микропрограмме микроопераций могут выполняться *параллельно во времени, в то время как другие — только последовательно*. Свойство совокупности микроопераций, гарантирующее возможность их одновременного выполнения, называется *совместимостью*. Микрооперации, не обладающие указанным свойством, называются *несовместимыми*.

Рассматриваются два аспекта совместимости. Совместимость, обусловленная содержанием операторов, реализуемых под действием микроопераций, называется *функциональной*. Примером двух функционально несовместимых микроопераций могут служить следующая пара: (y_8: Сч n := 0) и (y_{15}: Сч n := Сч $n-1$) и вообще любые микрооперации, присваивающие различные значения одной и той же переменной.

Если невозможность одновременного выполнения микроопераций связана с ограничениями возможностей структуры операционного автомата, то такая несовместимость называется *структурной*. Например, операторы (Рг C := Рг A) и (Рг D := Рг B) функционально совместимы, но если в конкретной структуре ОА связь между этими регистрами осуществляется через общую магистраль (шину), то они структурно несовместимы.

Вернемся к способам разбиения исходного множества микроопераций на подмножества. Очевидно, в каждое подмножество следует включать только

взаимно несовместимые микрооперации. При проектировании УА возникает вопрос: какой тип совместимости микроопераций учитывать при разбиении исходного множества Y на подмножества?

Если несовместимыми считать только те микрооперации, которые принципиально нельзя реализовать на заданной (спроектированной) структуре ОА, то таких пар окажется немного, большинство микроопераций будут попарно совместимыми, следовательно, их необходимо включать в разные подмножества. При этом число подмножеств p может превысить значение s и приближаться к n.

Если рассматривать в качестве совместимых только те микрооперации, которые размещаются в одной операторной вершине реализуемых алгоритмов, а все остальные считать несовместимыми, даже если их можно выполнить одновременно в структуре ОА (но не требуется при реализации данных алгоритмов), то $p \to s$, эффективность кодирования будет значительно выше. Правда, если потребуется модифицировать реализуемый алгоритм или добавить еще группу алгоритмов для реализации, и в одной операторной вершине окажутся микрооперации, включенные ранее в одно подмножество, придется заново перепроектировать УА.

Разработано несколько формальных методов [7] разбиения множества микроопераций на подмножества. В простейшем случае можно воспользоваться методом "прямого включения". Рассмотрим пример проектирования УАПЛ по заданной микропрограмме.

Пример проектирования УАПЛ

Мы уже говорили, что исходным для проектирования УА является микропрограмма, представленная, например, в форме ГСА. На рис. 4.14 изображена некоторая микропрограмма, которую мы будем считать исходной для проектирования нашего автомата.

Заметим, что на этапе проектирования управляющего автомата семантика ГСА не рассматривается: сейчас нас уже не интересует "правильность" микропрограммы относительно реализуемого алгоритма. Просто имеется *синтаксически правильно* построенная ГСА и требуется разработать устройство, реализующее это поведение.

В качестве управляющего устройства будем проектировать *управляющий микропрограммный автомат с программируемой логикой*.

Общая структура такого устройства представлена на рис. 4.8. Исходя из описанных выше вариантов организации адресации и способов кодирования поля микроопераций, выберем *естественную адресацию* и *смешанный способ ко-*

дирования микроопераций. Ограничимся единственным форматом микрокоманды.

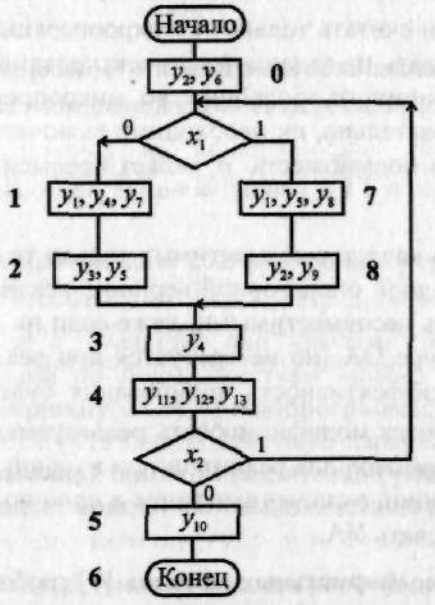


Рис. 4.14. Исходная микропрограмма для проектирования автомата

Определение формата микрокоманды

На разрядность полей микрокоманды влияют следующие параметры:

- количество различных микроопераций, формируемых УА, в конечном итоге определяет (с учетом выбранного способа кодирования) длину поля микроопераций;
- количество различных логических условий определяет длину поля x;
- количество вершин ГСА связано с общим числом микрокоманд, а следовательно, с объемом памяти микропрограмм и разрядностью поля адреса микрокоманды.

Множество микроопераций Y, используемых в заданной ГСА — $Y = \{y_1, y_2, ..., y_{13}\}$, мощность множества $|Y| = 13$. При горизонтальном кодировании поле микроопераций будет занимать 13 разрядов. Вертикальный способ кодирования микроопераций к заданной ГСА неприменим, поскольку ГСА содержит вершины с двумя и тремя микрооперациями. Попробуем реализовать разбиение множества Y на подмножества несовместимых микроопераций. Воспользуемся методом прямого включения,

учитывая, что отношение совместимости задано на самой ГСА. Строго говоря, следовало бы построить матрицу совместимости микроопераций, но в рассматриваемом примере небольшой размер алгоритма позволяет определять отношение совместимости непосредственно по ГСА.

На сколько подмножеств следует разбивать исходное множество? По меньшей мере, на $s = 3$ в нашем случае. Образуем три подмножества — Y_1, Y_2, Y_3 и разместим в них микрооперации операторной вершины, имеющей s микроопераций. Если в ГСА таких вершин несколько — выберем любую из них.

$$Y_1 = \{y_1\}, \quad Y_2 = \{y_4\}, \quad Y_3 = \{y_7\}.$$

Теперь разместим по множествам микрооперации следующей вершины, содержащей (в нашем случае) три микрооперации:

$$Y_1 = \{y_1\}, \quad Y_2 = \{y_4, y_5\}, \quad Y_3 = \{y_7, y_8\}.$$

Заметим, что первая микрооперация второй рассматриваемой вершины совпадает с первой микрооперацией первой вершины. Она уже присутствует в множестве Y_1 ($y_1 \in Y_1$), поэтому не включается вторично. Наконец, разместим микрооперации третьей "тройной" вершины:

$$Y_1 = \{y_1, y_{11}\}, \quad Y_2 = \{y_4, y_5, y_{12}\}, \quad Y_3 = \{y_7, y_8, y_{13}\}.$$

Теперь нераспределенными остались микрооперации (некоторые) "двойных" и "одинарных" вершин. Вершина (y_2, y_6) — обе микрооперации несовместимы с уже распределенными, поэтому могут располагаться произвольно, лишь бы они находились в разных подмножествах:

$$Y_1 = \{y_1, y_{11}, y_2\}, \quad Y_2 = \{y_4, y_5, y_{12}, y_6\}, \quad Y_3 = \{y_7, y_8, y_{13}\}.$$

Вершина (y_2, y_9) — y_9 нельзя помещать в Y_1, поскольку совместимая с ней $y_2 \in Y_1$. Подмножества лучше заполнять равномерно, поэтому разместим y_9 в Y_3:

$$Y_1 = \{y_1, y_{11}, y_2\}, \quad Y_2 = \{y_4, y_5, y_{12}, y_6\}, \quad Y_3 = \{y_7, y_8, y_{13}, y_9\}.$$

Остались две нераспределенные микрооперации — y_3 и y_{10}, первая из которых совместима с y_5, поэтому ее нельзя помещать в Y_2, а вторая несовместима ни с какими другими и может размещаться произвольно. Поместим их в множество, имеющее пока наименьшую мощность — Y_1:

$$Y_1 = \{y_1, y_{11}, y_2, y_3, y_{10}\}, \quad Y_2 = \{y_4, y_5, y_{12}, y_6\}, \quad Y_3 = \{y_7, y_8, y_{13}, y_9\}.$$

Все 13 микроопераций распределились по трем подмножествам, при этом выполняются условия (4.8) (т. е. имеет место разбиение исходного множества Y), однако УА обычно должен вырабатывать еще одну микрооперацию, свидетельствующую об окончании выполнения алгоритма и предназначенную для использования не в ОА, а в управляющем автомате верхнего уровня иерархии. Назовем эту микрооперацию y_k и включим в произвольное множество (например, в Y_2), поскольку она, естественно, несовместима ни с одной микрооперацией. Итак, имеем следующее распределение:

$$Y_1 = \{y_1, y_{11}, y_2, y_3, y_{10}\}, \quad Y_2 = \{y_4, y_5, y_{12}, y_6, y_k\}, \quad Y_3 = \{y_7, y_8, y_{13}, y_9\}.$$

Для кодирования элементов каждого из трех подмножеств потребуется по три двоичных разряда. Может показаться, что для Y_3 хватит и двух, ведь $|Y_3| = 4 = 2^2$, однако следует учесть, что в каждом подмножестве необходимо предусмотреть один код для случая отсутствия микрооперации из этого подмножества в микрокоманде. Оптимальным разбиением исходного множества будет такое, когда $\forall |Y_i| = 2r - 1$, где r — натуральное число.

В подмножестве Y_3 всего одна "лишняя" микрооперация, а среди кодов Y_1 и Y_2 есть свободные. Попробуем перенести одну из микроопераций из Y_3 в другое подмножество, сохраняя, естественно, требование к попарной несовместимости всех микроопераций одного подмножества. Очевидно, первые три элемента подмножества Y_3 нельзя перенести в другое, т. к. они являются микрооперациями из "тройных" вершин. Зато микрооперация y_9 совместима только с $y_2 \in Y_1$, поэтому y_9 можно перенести в Y_2. Окончательно получим, предварительно упорядочив элементы подмножеств в порядке возрастания индексов:

$$Y_1 = \{y_1, y_2, y_3, y_{10}, y_{11}\}, \quad Y_2 = \{y_4, y_5, y_6, y_9, y_{12}, y_k\}, \quad Y_3 = \{y_7, y_8, y_{13}\}.$$

Теперь мы можем определить размеры полей микрокоманды. Поле микроопераций будет состоять из трех подполей — Y_1, Y_2, Y_3 (назовем их по именам соответствующих подмножеств), размером в 3, 3 и 2 двоичных разрядов соответственно.

Поле номера условия x должно содержать номер одного из двух логических условий — x_1, x_2 (один разряд?), однако для повышения гибкости процесса микропрограммирования удобно иметь возможность выбирать еще и *тождественно истинное* и *тождественно ложное* условия. Итак, поле x занимает два разряда.

Наконец, поле адреса определяется объемом памяти микропрограмм. Если в нашем примере мы будем считать, что разрабатываем УА только для реали-

зации микропрограммы рис. 4.14, а она содержит 8 вершин, не считая начальной, конечной и условных, количество микрокоманд (каждая микрокоманда — ячейка памяти, имеющая свой адрес), выдаваемых УА, будет никак не менее 8, а реально — (1,2, ..., 1,3) × 8, то для поля адреса в микрокоманде следует отвести 4 разряда ($2^4 = 16 > 1{,}3 \times 8 \approx 11$).

В поле адреса будет располагаться адрес памяти — двоичный номер ячейки, а в полях Y_i и x — коды микроопераций и логических условий. Окончательно формат микрокоманды будет иметь вид, приведенный на рис. 4.15.

Y_1	Y_2	Y_3	x	A1
3	3	2	2	4

Рис. 4.15. Формат микрокоманды

Кодировка микроопераций и логических условий

Здесь может осуществляться произвольно, например так, как показано в табл. 4.4.

Выбрав кодировку, можно начинать писать микропрограмму в машинных *микрокодах*. Фактически мы формируем содержимое ПЗУ микропрограмм (табл. 4.5).

Анализируя ГСА микропрограммы (см. рис. 4.14), увидим, что в первом такте работы автомата должны быть выданы микрооперации y_2 и y_6. Учитывая, что в начальном состоянии автомата Рг Сч А МК удобно установить в 0, микрокоманда, расположенная по нулевому адресу, должна сформировать микрооперации y_2 и y_6. Из табл. 4.4 следует, что $y_2 \in Y_1$ и имеет код 010, а $y_6 \in Y_2$ (код 011). Микрооперации, включенные в множество Y_3, в этой микрокоманде отсутствуют. Тогда поле микроопераций микрокоманды должно содержать следующий код: 010 011 00.

После первой операторной вершины ГСА следует условная вершина, содержащая логическую переменную x_1. Следовательно, в микрокоманде должна анализироваться переменная x_1. При $x_1 = 0$ следующей должна выполняться микрокоманда (y_1, y_4, y_7) по адресу 1, иначе — микрокоманда (y_1, y_5, y_8) по неизвестному пока адресу. Заполним строку таблицы для нулевой ячейки памяти следующим кодом: 010 011 00 01 ????. К этой строке нам еще придется вернуться для заполнения поля адреса перехода.

По адресу 1 должна располагаться микрокоманда, формирующая микрооперации (y_1, y_4, y_7) и безусловно передающая управление следующей микро-

команде (y_3, y_5). Заполняем строку таблицы для первой ячейки памяти: 001 001 01 00 *xxxx*. В поле *x* этой микрокоманды код 00 указывает на тождественно ложное условие (константу 0) — к Рг Сч А МК будет добавлена 1, а содержимое поля адреса перехода не используется.

Таблица 4.4. Таблицы кодирования микроопераций и логических условий

Код	Y_1	Y_2	Y_3	Код	x
000	∅	∅	∅	00	Константа 0
001	y_1	y_4	y_7	01	x_1
010	y_2	y_5	y_8	10	x_2
011	y_3	y_6	y_{13}	11	Константа 1
100	y_{10}	y_9			
101	y_{11}	y_{12}			
110	—	y_k			
111	—	—			

Действуя аналогичным образом, заполняем строки табл. 4.5, соответствующие адресам ПЗУ 3, 4, 5, 6 (на рис. 4.14 рядом с операторными вершинами обозначены адреса соответствующих микрокоманд). В микрокоманде по адресу 4 выполняется условный переход по переменной x_2, причем адрес перехода при $x_2 = 1$ пока также неизвестен. По адресу 6 размещается микрокоманда, соответствующая конечной вершине ГСА, завершающая работу микропрограммы микрооперацией y_k. Таким образом, завершено микропрограммирование участка ГСА от начальной до конечной вершины, соответствующего нулевым значениям логических переменных.

Теперь можно вернуться к логической вершине, размещенной после первой операторной. Микрокоманду, следующую за ее единичным выходом (y_1, y_5, y_8), можно разместить по следующему свободному (7) адресу. Поэтому в поле адреса перехода ячейки 0 теперь можно поместить код 0111. Сама микрокоманда по адресу 7 формирует три микрооперации и переходит к микрокоманде по адресу 8: 001 010 10 00 *xxxx*.

Микрокоманда по адресу 8 не связана с логической вершиной, но она должна передавать управление уже существующей (по адресу 3) микрокоманде. По-

этому ее поле $x=11$ адресует тождественно истинное условие, а в поле переадресации указан адрес 3.

Таблица 4.5. Содержимое ПЗУ микропрограмм

Адрес	Y_1	Y_2	Y_3	x	A1
0	010	011	00	01	0111 (7)
1	001	001	01	00	xxxx
2	011	010	00	00	xxxx
3	000	001	00	00	xxxx
4	101	101	11	10	1001 (9)
5	100	000	00	00	xxxx
6	000	110	00	00	xxxx
7	001	010	10	00	xxxx
8	010	100	00	11	0011 (3)
9	000	000	00	01	0111 (7)
10	000	000	00	11	0001 (1)

Наконец, остался неопределенным адрес перехода в микрокоманде по адресу 4. Сейчас уже всем операторным вершинам ГСА (включая конечную) соответствуют микрокоманды в ячейках ПЗУ. На какую из них следует передать управление после проверки условия x_2, если оно окажется истинным? Из ГСА видно, что тогда следует проверить условие x_1 — случай двух подряд расположенных условных вершин. Передать управление на адрес 0, где проверяется это условие? Но тогда выполнятся и микрооперации y_1, y_4, y_7, а это микропрограммой не предусмотрено. Очевидно, в микропрограмму следует включить дополнительно микрокоманды (в нашем случае — по адресам 9 и 10), которые не формируют никаких микроопераций, а обеспечивают только передачу управления. Первая осуществляет условный переход по переменной x_1 на адрес 7, вторая (которая будет выполняться только при $x_1=0$) — безусловно на адрес 1. Теперь код микропрограммы полностью сформирован.

Осталось изобразить структурную схему разработанного управляющего автомата (рис. 4.16).

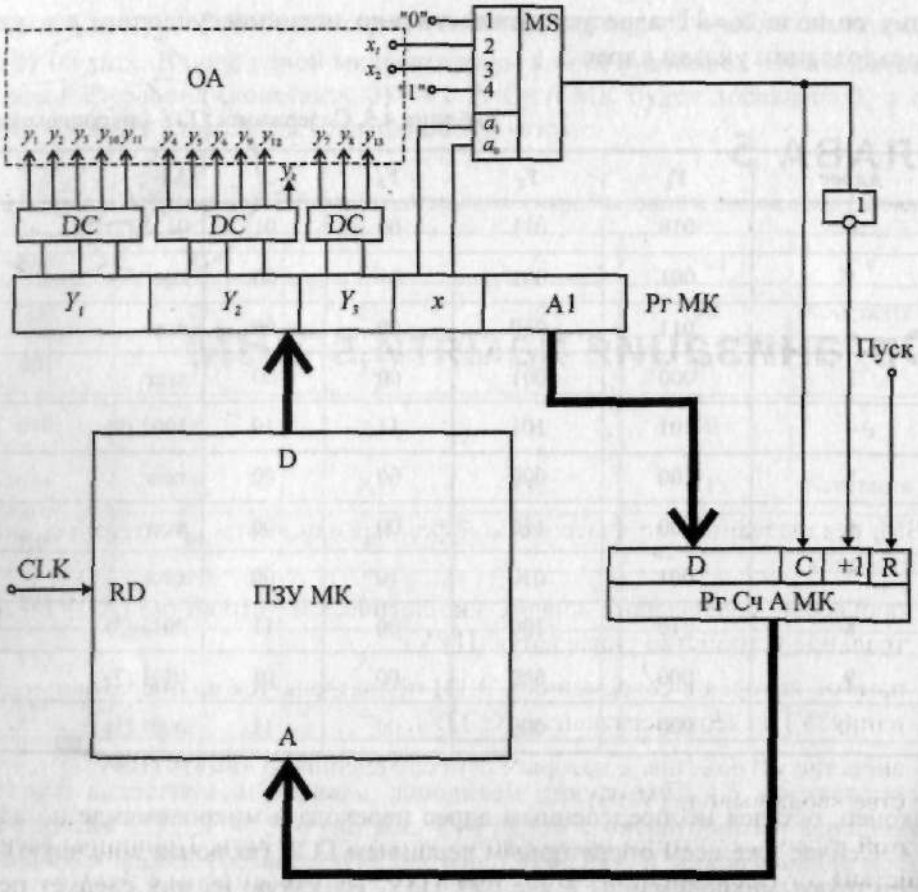
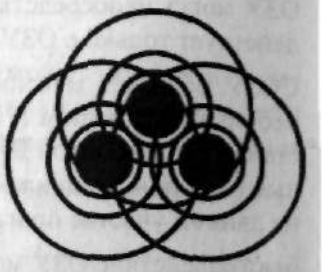


Рис. 4.16. *Структурная схема управляющего автомата*

ГЛАВА 5



Организация памяти в ЭВМ

ЭВМ, реализованная по классической фон-неймановской архитектуре, включает в себя:

- процессор, содержащий арифметико-логическое устройство (АЛУ) и центральное устройство управления (ЦУУ);
- память, которая в современных ЭВМ подразделяется на оперативную (ОП или ОЗУ) и сверхоперативную (СОЗУ);
- внешние устройства, к которым относят внешнюю память (ВЗУ) и устройства ввода/вывода (УВВ).

В этой главе рассмотрим организацию устройств памяти. Принципы взаимодействия других устройств ЭВМ с процессором рассмотрены в *разд. 6.3*.

5.1. Концепция многоуровневой памяти

Известно, что память ЭВМ предназначена для хранения программ и данных, причем эффективность работы ЭВМ во многом определяется характеристиками ее памяти. Во все времена к памяти предъявлялись три основных требования: большой *объем*, высокое *быстродействие* и низкая (умеренная) *стоимость*.

Все перечисленные выше требования к памяти являются взаимно-противоречивыми, поэтому пока невозможно реализовать один тип ЗУ, отвечающий всем названным требованиям. В современных ЭВМ организуют комплекс разнотипных ЗУ, взаимодействующих между собой и обеспечивающих приемлемые характеристики памяти ЭВМ для каждого конкретного применения.

В основе большинства ЭВМ лежит трехуровневая организация памяти: сверхоперативная (СОЗУ) — оперативная (ОЗУ) — внешняя (ВЗУ). СОЗУ и

ОЗУ могут непосредственно взаимодействовать с процессором, ВЗУ взаимодействует только с ОЗУ.

СОЗУ обладает максимальным быстродействием (равным процессорному), небольшим объемом (10^1—10^5 байтов) и располагается, как правило, на кристалле процессорной БИС. Для обращения к СОЗУ не требуются магистральные (машинные) циклы. В СОЗУ размещаются наиболее часто используемые *на данном участке программы данные, а иногда — и фрагменты программы.*

Быстродействие ОЗУ может быть ниже процессорного (не более чем на порядок), а объем составляет 10^6—10^9 байтов. В ОЗУ располагаются подлежащие выполнению программы и обрабатываемые данные. Связь между процессором и ОЗУ осуществляется *по системному или специализированному интерфейсу* и требует для своего осуществления машинных циклов.

Информация, находящаяся в ВЗУ, не может быть непосредственно использована процессором. Для использования программ и данных, расположенных в ВЗУ, *их необходимо предварительно переписать в ОЗУ. Процесс обмена информацией* между ВЗУ и ОЗУ осуществляется средствами специального канала или (реже) — непосредственно под управлением процессора. Объем ВЗУ практически неограничен, а быстродействие на 3—6 порядков ниже процессорного.

Схематически взаимодействие между процессором и уровнями памяти представлено на рис. 5.1.

Рис. 5.1. Взаимодействие ЗУ различных уровней в составе ЭВМ

Следует помнить, что *положение ЗУ в иерархии памяти ЭВМ определяется* не элементной базой запоминающих ячеек (известны случаи реализации ВЗУ на БИС — "электронный диск" и, наоборот, организация оперативной памяти на электромеханических ЗУ — магнитных барабанах), а возможностью доступа процессора к данным, расположенным в этом ЗУ.

При организации памяти современных ЭВМ (МПС) особое внимание уделяется сверхоперативной памяти и принципам обмена информацией между ОЗУ и ВЗУ.

5.2. Сверхоперативная память

Применение СОЗУ в иерархической памяти ЭВМ может обеспечить повышение производительности ЭВМ за счет снижения среднего времени обращения к памяти T при условии, что время цикла СОЗУ T_C будет (значительно) меньше времени цикла ОЗУ T_0. Очевидно:

$$T = p_C \cdot T_C + (1 - p_C) \cdot T_0, \qquad (5.1)$$

где p_C — вероятности обращения к СОЗУ. Обозначим так же: p_O — вероятности обращения к ОЗУ.

Из (5.1) следует, что повышение производительности ЭВМ может осуществляться двумя путями:

- уменьшением отношения $\frac{T_C}{T_0}$;
- увеличением вероятности p_C обращения в СОЗУ.

Первый путь связан, прежде всего, с технологическими особенностями производства БИС и здесь не рассматривается.

Если считать, что информация размещается в СОЗУ и ОЗУ случайным образом, то вероятности p_C и p_O пропорциональны объемам соответствующих ЗУ. В этом случае $p_C << p_O$ и наличие в ЭВМ СОЗУ практически не влияет на ее производительность.

То же можно было бы сказать и о ситуации, когда отношение $\frac{T_C}{T_0} \approx 1$, но не следует забывать, что наличие в ЭВМ СОЗУ с прямой адресацией (РОН) позволяет включать в систему команд короткие команды, использовать косвенно-регистровую адресацию и, в конечном итоге, увеличивать производительность ЭВМ даже при $T_C = T_0$.

Итак, для эффективного применения СОЗУ следует таким образом распределять информацию по уровням памяти ЭВМ, чтобы в СОЗУ всегда располагались наиболее часто используемые в данный момент коды.

Принято различать СОЗУ по способу доступа к хранимой в нем информации. Известны два основных класса СОЗУ по этому признаку:

- с прямым доступом;
- с ассоциативным доступом.

5.2.1. СОЗУ с прямым доступом

СОЗУ с прямым доступом (РОН — регистры общего назначения) получило широкое распространение в большинстве современных ЭВМ. Фактически РОН — это небольшая регистровая намять, доступ к которой осуществляется специальными командами. Стратегия размещения данных в РОН целиком определяется программистом (компилятором). Обычно в РОН размещают многократно используемые адреса (базы, индексы), счетчики циклов, данные активного фрагмента задачи, что повышает вероятность обращения в ячейки РОН по сравнению с ячейками ОЗУ.

5.2.2. СОЗУ с ассоциативным доступом

Применение *СОЗУ с ассоциативным доступом* позволяет автоматизировать процесс размещения данных в СОЗУ, обеспечивая "подмену" активных в данный момент ячеек ОЗУ ячейками СОЗУ. Эффективность такого подхода существенно зависит от выбранной стратегии замены информации в СОЗУ, причем использование ассоциативного СОЗУ имеет смысл только при условии $T_C << T_0$.

Принцип ассоциативного доступа состоит в следующем. Накопитель ассоциативного запоминающего устройства (АЗУ) разбит на два поля — информационное и признаков. Структура информационного поля накопителя соответствует структуре обычного ОЗУ, а запоминающий элемент поля признаков, помимо функции записи, чтения и хранения бита, обеспечивает сравнение хранимой информации с поступающей и выдачу признака равенства.

Признаки равенства всех элементов одной ячейки поля признаков объединяются по "И" и устанавливают в 1 индикатор совпадения ИС, если информация, хранимая в поле признака ячейки, совпадает с информацией, подаваемой в качестве признака на вход *P* накопителя.

Во второй фазе обращения (при чтении) на выход данных *D* последовательно поступает содержимое информационных полей тех ячеек, индикаторы совпадения которых установлены в 1 (если таковые найдутся).

Способ использования АЗУ в качестве сверхоперативного иллюстрирует рис. 5.2. В информационном поле ячеек АСОЗУ — копия информации некоторых ячеек ОЗУ, а в поле признаков — адреса этих ячеек ОЗУ. Когда процессор генерирует обращение к ОЗУ, он одновременно (или прежде) инициирует процедуру опроса АСОЗУ, выдавая в качестве признака адрес ОЗУ.

Если имеет место совпадение признака ячейки с запрашиваемым адресом (не более одного раза, алгоритм загрузки АСОЗУ не предусматривает возможности появления одинаковых признаков), то процессор обращается (по чтению

или по записи) в информационное поле этой ячейки АСОЗУ, при этом блокируется обращение к ОЗУ. Если требуемый адрес не найден в АСОЗУ, инициируется (или продолжается) обращение к ОЗУ, причем в АСОЗУ создается копия ячейки ОЗУ, к которой обратился процессор. Повторное обращение процессора по этому адресу будет реализовано в АСОЗУ (на порядок быстрее, чем в ОЗУ).

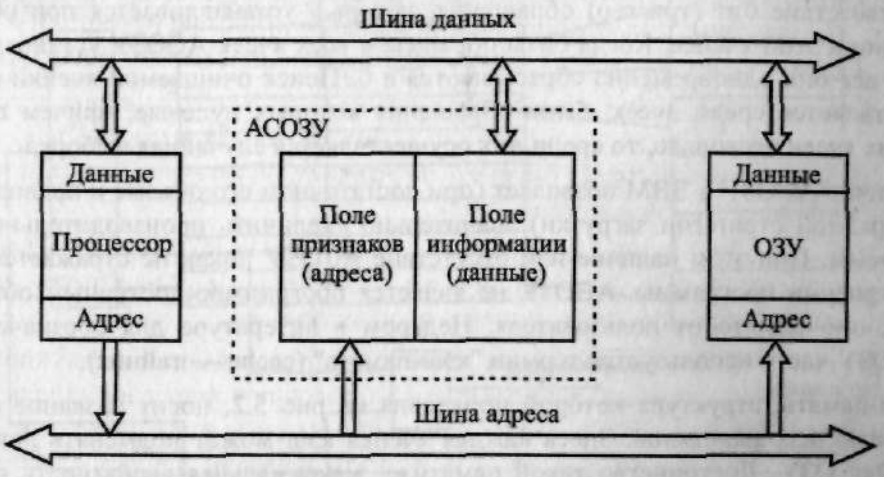


Рис. 5.2. СОЗУ с ассоциативным доступом

Таким образом, в АСОЗУ создаются копии тех ячеек ОЗУ, к которым в данный момент обращается процессор в надежде, что "в ближайшее время" произойдет новое обращение по этому адресу. (Существуют и другие стратегии загрузки АСОЗУ, например, если процессор обращается в ОЗУ по определенному адресу, то в АСОЗУ перемещается содержимое целого блока соседних ячеек.)

При необходимости записи в АСОЗУ новой информации требуется отыскать свободную ячейку, а при ее отсутствии (что чаще всего и бывает) — отыскать ячейку, содержимое которой можно удалить из АСОЗУ. При этом следует помнить, что если во время пребывания ячейки в АСОЗУ в нее производилась запись, то требуется не просто очистить содержимое ячейки, а записать его в ОЗУ по адресу, хранящемуся в поле признаков, т. к. процессор, отыскав адрес в АСОЗУ, производит запись только туда, оставляя в ОЗУ старое значение (т. н. *"АСОЗУ с обратной записью"*). Возможен и другой режим работы СОЗУ — со сквозной записью, при котором всякая запись осуществляется и СОЗУ, и в ОЗУ.

При поиске очищаемой ячейки чаще всего используют *метод случайного выбора*. Иногда отмечают ячейки, в которые не проводилась запись, и поиск "кандидата на удаление" проводят из них.

Более сложная процедура замещения предполагает учет длительности пребывания ячеек в АСОЗУ, или частоты обращения по этому адресу, или времени с момента последнего обращения. Однако все эти методы требуют дополнительных аппаратных и временных затрат.

Одним из наиболее дешевых способов, позволяющих учитывать поток обращений к ячейкам, является следующий. Каждой ячейке АСОЗУ ставится в соответствие бит (триггер) обращения, который устанавливается при обращении к этой ячейке. Когда биты обращения всех ячеек АСОЗУ установятся в 1, все они одновременно сбрасываются в 0. Поиск очищаемой ячейки осуществляется среди ячеек, биты обращения которых нулевые, причем если таких ячеек несколько, то среди них осуществляется случайная выборка.

Наличие АСОЗУ в ЭВМ позволяет (при достаточном его объеме и правильно выбранной стратегии загрузки) значительно увеличить производительность системы. При этом наличие или отсутствие АСОЗУ никак не отражается на построении программы. АСОЗУ не является программно-доступным объектом, оно скрыто от пользователя. Недаром в литературе для обозначения АСОЗУ часто используется термин "кэш-память" (cache — тайник).

Кэш-память, структура которой приведена на рис. 5.2, носит название *полностью ассоциативной*. Здесь каждая ячейка кэш может подменять любую ячейку ОЗУ. Достоинство такой памяти — максимальная вероятность кэш-попадания (при прочих равных условиях), по сравнению с другими способами организации кэш. К недостаткам можно отнести сложность ее структуры (а следовательно, и высокую стоимость). Действительно, в каждом разряде поля признаков необходимо реализовать, наряду с возможностями записи и хранения, функцию сравнения хранимого бита с соответствующим битом признака, а потом конъюнкцию результатов сравнения разрядов в каждой ячейке.

Кэш-память с *прямым отображением* требует минимальных затрат оборудования (по сравнению с другими вариантами организации кэш), но имеет минимальную вероятность кэш-попаданий. Суть организации (рис. 5.3) состоит в следующем. Физическая оперативная намять разбивается на блоки (множества) одинакового размера, количество которых (блоков) соответствует числу ячеек кэш, причем каждой строке ставится в соответствие определенное множество ячеек памяти, не пересекающееся с другими. Все ячейки множества претендуют на одну строку кэш.

Такая организация кэш исключает собственно ассоциативный поиск, а следовательно, значительно упрощается схема ячейки поля признаков. Действительно, здесь копия требуемой ячейки оперативной памяти может располагаться в *единственной* строке кэш. Часть физического адреса (на рис. 5.3 — старшая) определяет номер множества и, следовательно, строку кэш. Содер-

жимое этой строки выбирается по обычному адресному принципу, и поле тега сравнивается с младшей частью физического адреса. Таким образом, для всей кэш-памяти (любого размера) достаточно единственной схемы сравнения.

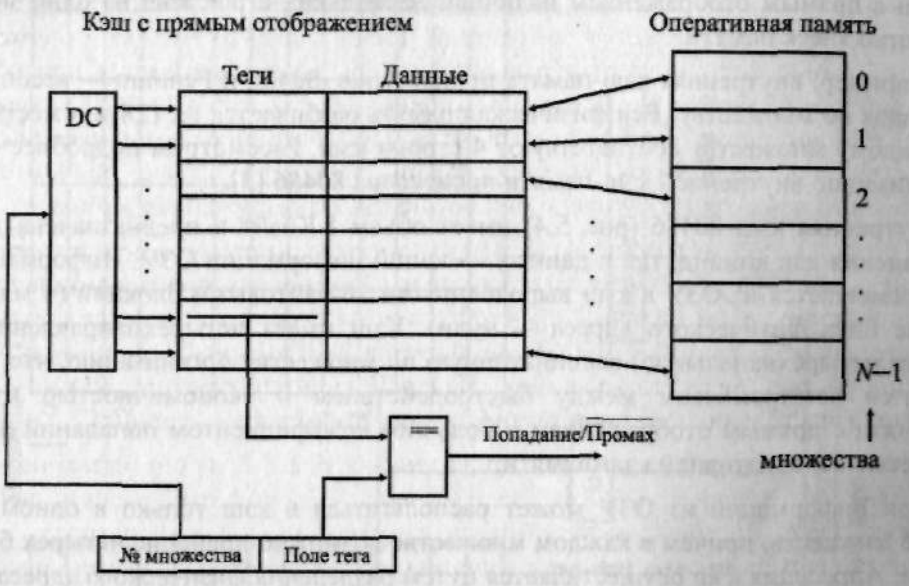


Рис. 5.3. Кэш с прямым отображением

Однако предложенная выше структура имеет существенный недостаток. Если проводить разбиение памяти на множества, как показано на рис. 5.3, то в большинстве случаев кэш будет использоваться крайне неэффективно.

Во-первых, хотя адресное пространство физической памяти 32-разрядных микропроцессоров составляет 2^{32} байтов, в современных ПЭВМ обычно используют намять объемом 2^{25}—2^{29} байтов. Следовательно, строки кэш, отображаемые на старшие (физически отсутствующие) множества памяти, никогда не будут использованы.

Во-вторых, если в множества включать следующие подряд ячейки ОЗУ, то копии никаких двух последовательных ячеек ОЗУ нельзя одновременно иметь в кэш (кроме случая последней и первой ячеек двух соседних множеств), что противоречит одной из основополагающих стратегий загрузки кэш — целесообразности копирования в кэш группы последовательных ячеек ОЗУ.

Для исключения отмеченных недостатков разбиение ячеек памяти на множества осуществляется таким образом, чтобы соседние ячейки относились к

разным множествам, что достигается размещением поля номера множества не в старших, а в младших разрядах физического адреса.

Для дальнейшего увеличения вероятности кэш-попаданий можно реализовать вариант кэш-памяти, *ассоциативной по множеству*, которая отличается от кэш с прямым отображением наличием нескольких строк кэш на одно множество ячеек памяти.

Например, внутренняя кэш-память процессоров i80486 и Pentium — ассоциативная по множеству. Вся физическая память разбивается на 128 множеств, а каждому множеству соответствуют 4 строки кэш. Рассмотрим подробнее организацию внутренней кэш-памяти процессора 80486 [3].

Внутренняя кэш 80486 (рис. 5.4) имеет объем 8 Кбайт и предназначена для хранения как команд, так и данных — копий информации ОЗУ. Информация *перемещается из ОЗУ в кэш выровненными 16-байтовыми блоками* (4 младшие бита физического адреса — нули). Кэш имеет четырехнаправленную (или четырехканальную) ассоциативную по множеству организацию, что является компромиссом между быстродействием и экономичностью кэш-памяти с прямым отображением и большим коэффициентом попаданий полностью ассоциативной кэш-памяти.

Блок информации из ОЗУ может располагаться в кэш только в одном из 128 множеств, причем в каждом множестве возможно хранение четырех блоков. Адресация кэш осуществляется путем разделения физического адреса на три поля:

- 7 битов поля индекса (A4—A10) определяют номер множества, в котором *проводится поиск*;
- старшие 21 бит адреса являются полем тега (признака), по которому осуществляется ассоциативный поиск (внутри множества из четырех блоков);
- четыре младшие бита адреса определяют позицию байта в блоке.

Когда при чтении возникает промах, в кэш копируется из ОЗУ 16-байтовый блок (строка), содержащий запрошенную информацию.

4-битовое поле достоверности показывает, являются ли в данный момент кэшированные данные достоверными (для каждого блока (строки) множества — свой бит). При очистке кэш-памяти или сбросе процессора все биты достоверности сбрасываются в 0. Когда производится заполнение строки кэш, место для заполнения выбирается просто нахождением любой недостоверной строки (из четырех строк "своего" множества).

Если недостоверных строк нет, то реализуется алгоритм замещения строк *"псевдоLRU"* ("наиболее давно используемый"). Для каждого множества в блоке отведено три бита LRU, которые обновляются при каждом кэш-

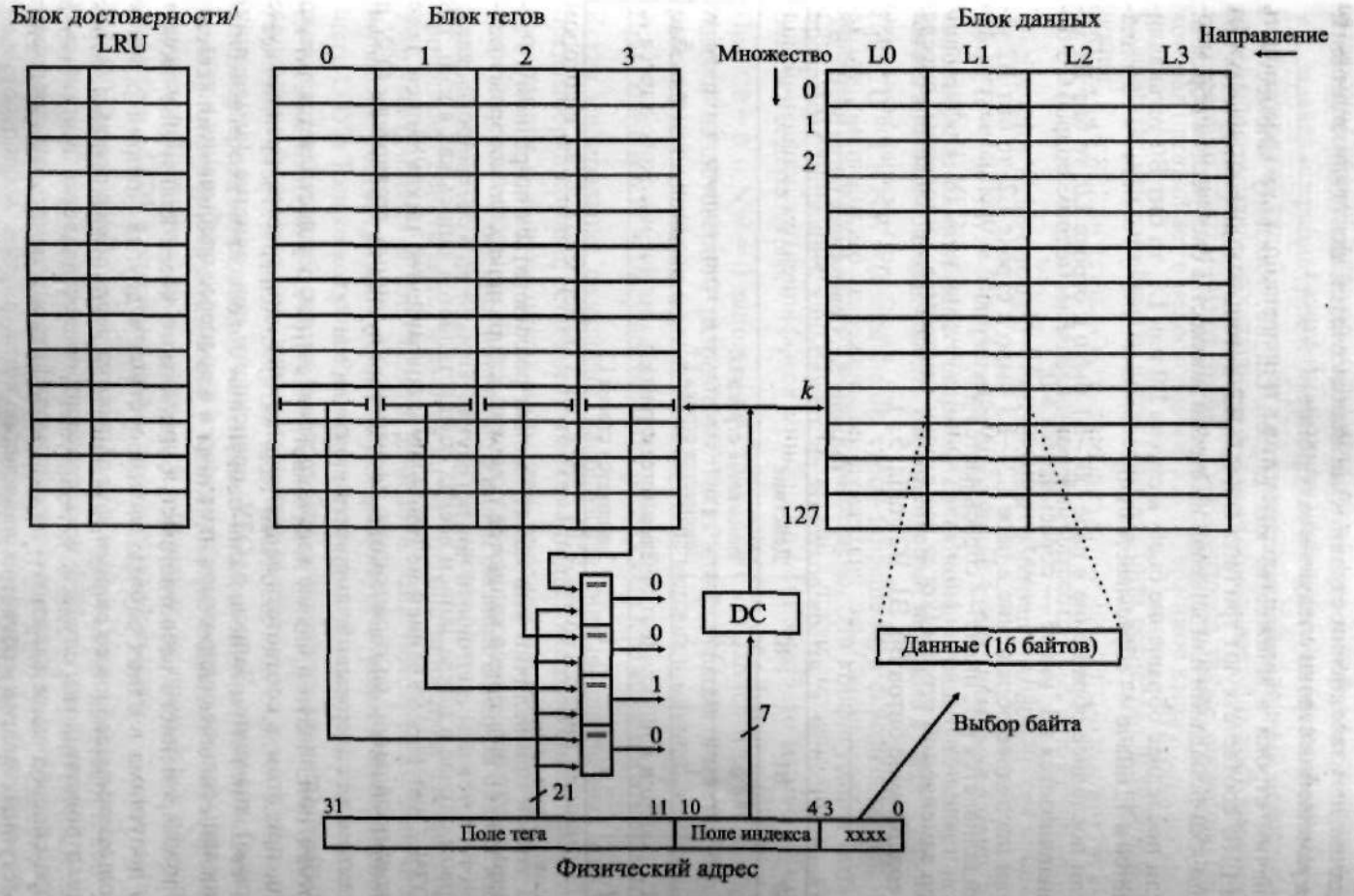


Рис. 5.4. Внутренняя кэш-память 80486

попадании или заполнении строки. Они используются для реализации алгоритма замещения строки следующим образом.

Обозначим строки в множестве как L0, LI, L2, L3. Каждому множеству в блоке LRU соответствуют три бита B0, B1, B2, которые модифицируются при каждом кэш-попадании или заполнении строки множества следующим образом:

- если последнее обращение было к строке L0 или L1, то бит B0 устанавливается в 1, иначе — сбрасывается в 0;
- если последнее обращение в паре L0—L1 было к строке L0, то бит B1 устанавливается в 1, иначе — сбрасывается в 0;
- если последнее обращение в паре L2—L3 было к строке L2, то бит B2 устанавливается в 1, иначе — сбрасывается в 0.

Выбор заменяемой строки (когда все строки множества достоверны) определяет содержимое битов B0, B1, B2 (табл. 5.1).

Таблица 5.1. Содержимое битов B0, B1, B2

B0	B1	B2	Действие
0	0	x	Заменяется строка L0
0	1	x	Заменяется строка L1
1	x	0	Заменяется строка L2
1	x	1	Заменяется строка L3

Цикл записи при наличии кэш-памяти может реализоваться по-разному. Различают кэш со *сквозной записью* и кэш с *обратной записью*.

В первом случае в цикле записи всегда осуществляется запись как в кэш, так и в ОЗУ. Этот способ записи не приводит к сокращению цикла записи даже при кэш-попадании, но гарантирует идентичность данных по адресам ОЗУ и кэш.

При обратной записи в случае кэш-попадания запись осуществляется только в кэш, при этом в соответствующей ячейке ОЗУ сохраняется прежнее (уже неверное) значение. Запись в ОЗУ происходит при очистке (замещении) строки кэш, если ее содержимое изменялось в процессе пребывания в кэш.

Ситуация временного несоответствия содержимого ячеек кэш и ОЗУ может быть допустима в одних случаях и недопустима в других (например, когда несколько процессоров со своими кэш общаются через общее поле ОЗУ). Поэтому в большинстве случаев пользователю предоставляется возможность выбора способа записи в кэш — за счет модификации некоторых программно-доступных флагов в регистре управления.

В 80486 строки кэш-памяти можно по отдельности объявить недостоверными, задавая операцию недостоверности кэш-памяти на шине процессора. При инициализации такой операции кэш сравнивает объявленный недостоверным адрес с тегом строк, находящихся в кэш, и сбрасывает бит достоверности при обнаружении соответствия тегов. Предусмотрена также операция очистки, которая превращает в недостоверное все содержимое кэш.

Конфигурацией кэш-памяти управляют два бита регистра CR0 состояния машины:

- CD (Cache Disable) — запрещение кэш-памяти;
- NW (Not Write-through) — несквозная (обратная) запись.

При CD = 1 и NW = 1 запрещено заполнение строк, сквозная запись и объявление кэш-памяти недостоверной. Такая конфигурация позволяет использовать внутреннюю кэш-память как *быстродействующее ЗУПВ*.

При CD = 1 и NW = 0 заполнение строк запрещено, а сквозная запись и объявление кэш-памяти недостоверной разрешено. Эта конфигурация позволяет программе запрещать кэш-память на короткое время, а затем разрешать без очистки содержимого.

При C0 = 0 и NW = 0 заполнение строк, сквозная запись и объявление кэш-памяти недостоверной разрешены. Такая конфигурация является обычной рабочей для кэш-памяти.

При C0 = 0 и NW = 1 осуществляется работа кэш в режиме обратной записи.

Когда кэширование разрешено, кэшируются считывания данных из ОЗУ и предвыборка команд, если внешняя схема подает входной сигнал разрешения кэш-памяти в данном цикле шины или текущий элемент таблицы страниц разрешает кэширование. В тех циклах, где кэширование запрещено при промахе, заполнение строки кэш-памяти не производится. Однако кэш-память продолжает действовать, несмотря на то, что она запрещена для заполнения. Уже находящиеся в кэш-памяти данные используются, если, конечно, они являются достоверными. (Фактически реализуется режим быстродействующего ОЗУ.) Только когда все данные в кэш-памяти отмечены как недостоверные, что происходит при ее очистке, все внутренние запросы считывания приводят к формированию внешних циклов шины.

Когда разрешена сквозная запись, все записи, в том числе и при кэш-попадании, инициируют запись в память. Когда сквозная запись запрещена, внутренний запрос записи, вызвавший попадание, не приводит к производству записи в ОЗУ, а операции недостоверности запрещены.

Когда запрещены кэширование и сквозная запись, кэш-память можно использовать как быстродействующее статическое ОЗУ. В такой конфигурации на шину процессора передаются только записи, вызвавшие промах, а операции недостоверности игнорируются. Если предполагается использовать этот ре-

жим (CD = 1 и NW = 1), следует предварительно загрузить достоверные строки, используя операции чтения из памяти или регистров.

5.3. Виртуальная память

Выше были рассмотрены способы организации сверхоперативной памяти и ее взаимодействия с оперативной. Не менее, а порой и более важной проблемой является организация взаимодействия в паре ОЗУ — ВЗУ.

Известно, что в современных ЭВМ (кроме простейших) реализовано *динамическое распределение памяти* между несколькими задачами, существующими в ЭВМ в процессе решения. Даже для однозадачных конфигураций проблема динамического распределения памяти не теряет актуальности, т. к. в памяти, помимо задачи пользователя, всегда присутствует операционная система или ее фрагмент.

Наличие динамического распределения памяти предполагает, что программа компилируется в т. н. "логических" адресах, а в процессе работы происходит автоматическое преобразование логических адресов в физические.

Наибольшее распространение в ЭВМ получил метод динамического распределения памяти, называемый *страничной организацией виртуальной памяти*.

При использовании этого метода вся память ЭВМ (ОЗУ и ВЗУ) рассматривается как единая *виртуальная память*. Адрес в этой памяти называется *виртуальным* или *логическим*. Вся виртуальная память делится на фрагменты одинакового размера, называемые *виртуальными страницами*. Размер страницы обычно составляет 0,5—4 Кбайт. Виртуальный адрес представляется состоящим из двух частей — номера страницы и номера слова на странице (смещения).

Физическая память ЭВМ (ОЗУ и ВЗУ) так же делится на страницы, причем размер физической страницы выбирается равным размеру виртуальной. Таким образом, одна физическая страница может хранить одну виртуальную, причем порядок следования виртуальных страниц в программе совсем не обязательно сохранять на физических страницах. Достаточно лишь установить однозначное соответствие между номерами виртуальных и физических страниц.

Соответствие между номерами виртуальных и физических страниц устанавливается с помощью специальной *страничной таблицы* (СТ), которую поддерживает операционная система. Размер физической страницы равен размеру виртуальной, поэтому преобразования смещений на странице не производятся.

Поскольку размер СТ достаточно велик, она хранится целиком в ОЗУ и модифицируется операционной системой всякий раз, когда в распределении памяти происходят изменения.

Для увеличения скорости обращения к памяти активная часть СТ обычно хранится в специальной быстродействующей памяти, организованной, как правило, по ассоциативному принципу. При этом в поле признаков АЗУ СТ хранятся виртуальные адреса страниц (иногда вместе с номером программы — в мультипрограммных системах), а в информационной части — соответствующие им номера физических страниц.

Если в результате преобразования виртуального адреса в физический оказывается, что требуемая физическая страница располагается в ВЗУ, то выполнение программы становится невозможным, пока не произойдет "подкачка" требуемой страницы в ОЗУ. Такая ситуация называется *страничным сбоем* и должна формировать внутреннее прерывание, по которому запускается подпрограмма чтения страницы из ВЗУ в ОЗУ.

При этом возникает серьезная проблема поиска той страницы, которую можно удалить из ОЗУ, чтобы на освободившееся место записать требуемую страницу. Серьезность проблемы обусловлена тем, что неудачный выбор удаляемой страницы (в ближайшее время она вновь понадобится) связан со значительной потерей времени на передачу страниц между ОЗУ и ВЗУ.

5.3.1. Алгоритмы замещения

Правило, по которому при возникновении страничного сбоя выбирается страница для удаления из ОЗУ, называется *алгоритмом замещения*.

Для данной программы, порождающей некоторый поток обращений к памяти, существует, по крайней мере, одна такая последовательность замещений страниц, которая дает для этой программы минимальное количество страничных сбоев.

Теоретически доказано, что минимальное число страничных сбоев будет получено, если в алгоритме замещения использовать информацию о потоке обращений к страницам в будущем *(алгоритм Минковского — Шора)* или, по крайней мере, о вероятности обращений к страницам в будущем.

Алгоритмы замещения, использующие "информацию о будущем", называются *физически нереализуемыми*, их обычно применяют для оценки качества эвристических алгоритмов замещения.

Эвристические алгоритмы замещения используют информацию о потоке обращений к страницам в прошлом (историю процесса) для экстраполяции характеристик потока обращений в будущем. Как правило, используют три типа информации о прошлом: время пребывания страницы в ОЗУ (или, что то же — очередность поступления страниц), число обращений к страницам за определенный промежуток времени или отрезки времени с момента последнего обращения к страницам.

Эффективность эвристического алгоритма можно характеризовать отношением:

$$k = \frac{N_0}{N_e},$$

где N_0 — число страничных сбоев при решении данной задачи с применением физически нереализуемого алгоритма; N_e — то же с применением исследуемого эвристического алгоритма.

Эвристический алгоритм можно считать выбранным удачно (для данного класса задач), если коэффициент k близок к 1. Значение N_0 может быть получено путем моделирования решения задачи (повторное) с предварительно зафиксированным потоком обращений к страницам.

При выборе подходящего алгоритма замещения следует учитывать не только его эффективность k, но и аппаратные затраты и затраты времени на его реализацию.

Например, для реализации т. н. *НДИ-алгоритма* (наиболее давно используемая) каждой странице, находящейся в ОЗУ, ставится в соответствие таймер, который сбрасывается при обращении к странице. При страничном сбое необходимо осуществить поиск максимального элемента массива таймеров страниц. Для некоторых задач выигрыш времени за счет увеличения k при применении НДИ-алгоритма, по сравнению с алгоритмом случайного замещения, может быть сравним с потерей времени на поиск максимальных значений таймеров.

Некоторые алгоритмы замещения учитывают одновременно несколько параметров прошлого потока обращений.

Алгоритм *"Карабкающаяся страница"* (КС-алгоритм) поддерживает последовательность номеров страниц, находящихся в ОЗУ. При любом обращении к странице ее номер в последовательности перемещается на одну позицию в направлении начала, меняясь местами с предыдущим в последовательности номером (исключение — обращение к странице, номер которой стоит в начале последовательности). При возникновении страничного сбоя из ОЗУ удаляется страница, номер которой расположен в конце последовательности, а номер вновь поступившей страницы помещается в конец последовательности. КС-алгоритм учитывает как время пребывания страницы в ОЗУ, так и интенсивность обращения к странице, причем не требует значительных аппаратных затрат, а при страничном сбое — времени на поиск.

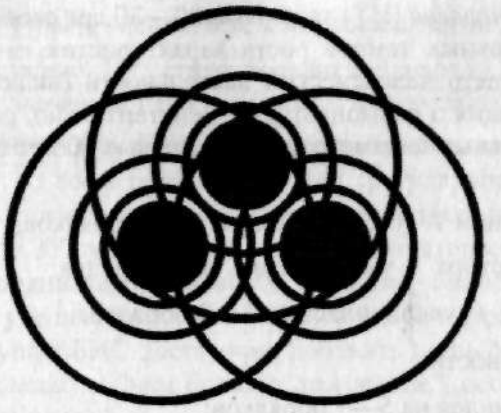
Алгоритм *"Рабочий комплект"* (РК-алгоритм) более сложен в реализации, но позволяет адаптировать свои параметры под конкретный класс задач. Все страницы ОЗУ, к которым было обращение в течение отрезка времени T,

образуют т. н. *рабочий комплект* и не подлежат удалению из ОЗУ. Остальные страницы (не вошедшие в рабочий комплект) образуют две очереди кандидатов на замещение, причем в первую очередь попадают страницы, на которые не было записи во время пребывания их в ОЗУ. При страничном сбое удаляется страница из первой очереди (FIFO — первый пришел из рабочего комплекта — первый ушел из ОЗУ), а если первая очередь пуста, то — из второй. Из очереди страница может опять попасть в рабочий комплект, если к ней будет обращение. Для реализации РК-алгоритма каждой странице ставится в соответствие таймер на T, причем каждое обращение к странице сбрасывает таймер (и переводит страницу в рабочий комплект, если она там отсутствовала), а переполнение таймера выводит страницу из рабочего комплекта. Подбором величины T можно оптимизировать РК-алгоритм под конкретный класс задач.

5.3.2. Сегментная организация памяти

До сих пор предполагалось, что виртуальная память, которой располагает программист, представляет собой непрерывный массив с единой нумерацией слов. Однако при написании программы удобно располагать несколькими независимыми сегментами (кода, данных, подпрограмм, стека и др.), причем размеры сегментов, как правило, заранее не известны. В каждом сегменте слова нумеруются с нуля независимо от других сегментов. В этом случае виртуальный адрес представляется состоящим из трех частей: <номер сегмента> <номер страницы> <номер слова>. В машине к виртуальному адресу может добавиться слева еще <номер задачи>. Таким образом, возникает определенная иерархия полей виртуального адреса, которой соответствует иерархия таблиц, с помощью которых виртуальный адрес переводится в физический. В конкретных системах может отсутствовать тот или иной элемент иерархии.

Виртуальная память была первоначально реализована на "больших" ЭВМ, однако по мере развития микропроцессоров в них так же использовались идеи страничной и сегментной организации памяти.



ЧАСТЬ II

Архитектура микропроцессорных систем

Интегральная технология (ИТ) за первые 20—30 лет своего развития достигла таких относительных темпов роста характеристик качества, которых не знала ни одна область человеческой деятельности (включая и такие бурно растущие, как авиация и космонавтика). Действительно, рассмотрим динамику изменений основных параметров ИТ за первые 20 лет ее развития (1960—1980 гг.):

- степень интеграции N увеличилась на 5—6 порядков;
- площадь транзистора S уменьшилась на 3 порядка;
- рабочая частота f увеличилась на 1—3 порядка;
- факторы добротности:
  - $f \times N$ увеличился на 5—7 порядков;
  - $P \times t$ уменьшился на 4 порядка, где t — задержка на элементе, P — мощность, рассеиваемая элементом;
- надежность (при сопоставлении элементо-часов) увеличилась на 4—8 порядков;
- производительность технологии (в транзисторах) увеличилась на 4—6 порядков;
- цена на транзистор в составе ИС уменьшилась на 2—4 порядка.

Американцы подсчитали, что если бы авиапромышленность в те же годы имела аналогичные темпы роста соответствующих показателей качества (стоимость — скорость — расход топлива = стоимость — быстродействие — рассеиваемая мощность), то "Боинг 767" стоил бы $500, облетал земной шар за 20 мин и расходовал на этот полет 10 л горючего.

Успехи ИТ в области элементной базы позволяли "поглощать" кристаллом все более высокие уровни ЭВМ: сначала — логические элементы, потом — операционные элементы (регистры, счетчики, дешифраторы и т. д.), далее — операционные устройства. Степень функциональной сложности, достигнутой в ИС, определяется особенностью технологии, разрешающей способностью инструмента, а также структурными особенностями схемы: регулярностью, связностью.

Под *регулярностью схемы* здесь будем понимать степень повторяемости элементов и связей по одной или двум координатам (при размещении структуры на плоскости). *Связность* — число внешних выводов схемы.

Кроме того, следует иметь в виду, что выпуск ИС был экономически оправдан лишь для функционально универсальных схем, обеспечивающих их достаточно большой тираж.

С этой точки зрения интересно взглянуть на соотношение ИС логики и памяти в процессе эволюции ИС — СИС — БИС — СБИС. Первые ИС (степень

интеграции $N \sim 10^1$) были исключительно логическими элементами. При достижении N примерно 10^2 стали появляться, наряду с операционными элементами, первые элементы памяти объемом в 16 — 64 — 128 битов.

По мере дальнейшего роста степени интеграции память стала быстро опережать "логику", т. к. по всем трем параметрам (регулярность, связность, тираж) имела перед логическими схемами преимущество. Действительно, структура накопителя ЗУ существенно регулярна (повторяемость элементов и связей по двум координатам), связность ее растет пропорционально логарифму объема (при увеличении объема памяти вдвое и сохранении без изменения способа доступа в БИС достаточно добавить лишь один вывод). Наконец, память "нужна всем" и "чем больше, тем лучше", особенно если "*больше, но за ту же (почти) цену*".

Что касается ИС логики, то на уровне $N \sim 10^3$ на кристалле можно уже размещать устройство ЦВМ (например, АЛУ, ЦУУ), но схемы логики (особенно управление) существенно нерегулярны, их связность (сильно зависящая от конкретной схемы) растет примерно пропорционально N, причем такие схемы, как правило, не являлись универсальными и не могли выпускаться большими тиражами (исключения в то время — БИС часов и калькуляторов).

Разработка первого микропроцессора (МП) — попытка создать универсальную логическую БИС, которая настраивается на выполнение конкретной функции после изготовления средствами программирования. На подобную БИС — МП первоначально предполагалось возложить лишь достаточно произвольные управляющие функции, однако позже МП стал использоваться как элементная база ЦВМ четвертого и последующих поколений. Появление МП вызвало необходимость разработки целого спектра универсальных логических БИС, обслуживающих МП: контроллеры прерываний и прямого доступа в память (ПДП), шинные формирователи, порты ввода/вывода и др.

Первый МП был разработан фирмой Intel и выпущен в 1971 г. на основе *p-МОП-технологии* (i4004). В 1972 и 1973 годах этой же фирмой были выпущены модели i4040, i8008. Эти микропроцессоры относились к т. н. *первому поколению*, обладали весьма ограниченными функциональными возможностями и очень быстро были вытеснены вторым поколением, которое было реализовано на основе n-МОП-технологии, что позволило, прежде всего, поднять тактовую частоту примерно на порядок относительно микропроцессоров первого поколения. Кроме того, прогресс интегральной технологии позволил повысить степень интеграции транзисторов на кристалле, а следовательно, увеличить сложность схемы.

Микропроцессоры *второго поколения*, самым распространенным из которого был выпущенный в 1974 г. i8080 (отечественный аналог — К580ВМ80), от-

личались достаточно развитой системой команд, наличием подсистем прерывания, прямого доступа в память, снабжался достаточным числом вспомогательных БИС, обеспечивающих управление памятью, параллельный и последовательный обмен с внешними устройствами, реализацию векторных прерываний, ПДП и др.

Многие идеи, заложенные в архитектуру систем на базе 8-разрядного микропроцессора i8080, неизменными используются и в современных мощных микропроцессорах.

Постоянное стремление к увеличению быстродействия ЭВМ привело разработчиков микропроцессоров "на поле" биполярной интегральной технологии, прежде всего — ТТЛ, где были выпущены микропроцессоры, отнесенные к *третьему поколению*, причем архитектура этих микропроцессоров существенно отличалась от их предшественников.

Известно, что для любого электронного прибора справедливо соотношение:

$$\Delta P \cdot \Delta t = \text{const},$$

где ΔP — энергия переключения, Δt — время переключения.

ТТЛ-транзисторы в составе ИС обладали (в то время) на порядок большим (по сравнению с n-МОП) быстродействием и соответственно на порядок большим потреблением мощности. Технологические трудности в то время не позволяли широко использовать активные способы отвода тепла от кристалла, поэтому единственный способ сохранения работоспособности кристалла в этих условиях — снижение степени интеграции.

Первый из выпущенных микропроцессоров третьего поколения — i3000 был двухразрядным! Очевидно, сохранение в этом случае традиционной архитектуры, характерной для микропроцессоров второго поколения, не привело бы к увеличению производительности системы, несмотря на то, что тактовая частота кристалла увеличивалась значительно (на порядок).

Решение этой проблемы повлекло значительные структурные изменения в микропроцессорах третьего поколения по сравнению со вторым:

- микропроцессоры выпускались в виде секций со средствами межразрядных связей, позволяющими объединять в одну систему произвольное число секций для достижения заданной разрядности. В состав секций включалось АЛУ, РОН и некоторые элементы устройства управления;
- устройство управления выносилось на отдельный кристалл (группу кристаллов), общий для всех процессорных секций;
- за счет резерва внешних выводов (малая разрядность) предусматривались отдельные шины адреса, ввода и вывода данных, причем данные от разных источников вводились по различным шинам;

- кристаллы управления представляли собой управляющий автомат с программируемой логикой, что позволяет достаточно легко реализовать практически любую систему команд на фиксированной структуре операционного устройства.

Таким образом, разработчики систем на базе микропроцессоров третьего поколения получали две "дополнительные степени свободы" — возможность выбрать произвольную разрядность процессора (кратную разрядности секции) и самостоятельно реализовать практически произвольную систему команд, оптимизированную для решения задач конкретного класса.

Поскольку микропроцессор в такой архитектуре размещался на нескольких кристаллах БИС: арифметико-логические секции, схемы управления вместе с БИС памяти микрокоманд, вспомогательные БИС (например, схемы ускоренного распространения переноса для АЛС) и др., то подобные микропроцессоры стали называть *многокристальными*, в отличие от *однокристальных* микропроцессоров второго поколения.

Очевидно и то, что разработка систем на многокристальных микропроцессорах требовала значительно больших усилий, времени и квалификации разработчиков, по сравнению с разработкой системы на "готовых" микропроцессорах второго поколения с фиксированной структурой и системой команд.

В конце 70-х и начале 80-х годов прошлого века значительное число отечественных и зарубежных фирм разрабатывали и выпускали серии БИС многокристальных микропроцессоров, причем разрядность секций постепенно увеличивалась до 4, 8 и даже 16 битов.

К тому времени технология уже не являлась решающим фактором классификации МП, ибо появились разновидности технологий одного типа, обеспечивающие очень широкий спектр характеристик МП, широкое распространение получили комбинированные технологии (например, $И^2Л$ + ТТЛШ). Поэтому многокристальные МП выпускались как по биполярной, так и по МДП-технологиям.

Одной из наиболее удачных разработок этого направления можно считать комплект БИС серии Am2900 фирмы AMD и близкую ему по архитектуре отечественную серию К1804 [13].

Параллельно интенсивно развивалась архитектура однокристальных микропроцессоров, наиболее характерным представителем которой можно считать семейство *x*86 фирмы Intel. Развитие этого направления отличал безудержный рост производительности процессоров, обусловленный увеличением разрядности процессоров, тактовой частоты, реализацией параллелизма на всех уровнях работы процессора и применением других архитектурных решений, характерных ранее для "больших" ЭВМ.

Быстро возрастающие возможности микропроцессоров позволяли "захватывать" в область цифровой обработки информации все новые сферы человече-

ской деятельности (достаточно вспомнить появление и распространение персональных ЭВМ).

Однако в сфере применения микропроцессоров всегда существовали задачи, для решения которых не требовалась высокая производительность процессора (например, управление несложным инерционным технологическим оборудованием, бытовыми приборами). В этих случаях на первый план выступали такие параметры, как надежность, простота реализации (стоимость). Для решения таких задач использование мощных однокристальных микропроцессоров становилось существенно избыточным.

Возрастающие возможности технологии в этом случае использовались не для увеличения производительности процессора, а для размещения на кристалле, наряду с относительно простым процессором, тех устройств, которые в традиционной архитектуре располагались на плате рядом с микропроцессором в виде отдельных БИС (СИС): тактовый генератор, ПЗУ, ОЗУ, порты параллельного и последовательного обмена, контроллер прерываний, таймеры и др.

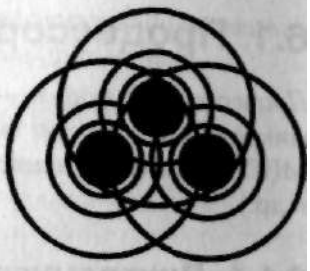
Таким образом, были получены полностью "самодостаточные" *однокристальные микроЭВМ* (ОМЭВМ). Это направление стало интенсивно развиваться, вначале на базе 8-разрядной архитектуры. Наиболее популярными из них можно считать ОМЭВМ семейств MCS-51 фирмы Intel, MC68HC11 фирмы Motorola, PIC16 и PIC18 фирмы Microchip.

По мере развития на кристаллах ОМЭВМ стали, помимо перечисленных выше устройств, размещать аналого-цифровые и цифроаналоговые преобразователи, блоки энергонезависимой памяти (EEPROM), сложные таймерные системы, схемы управления специализированными ВУ (например, семисегментной индикацией) и др.

Дальнейшее развитие технологии привело к появлению 16- и даже 32-разрядных однокристальных микроЭВМ (наиболее известные — от фирмы Motorola), включающих, наряду с мощным центральным процессором, специализированные процессоры — таймерный и ввода/вывода, работающие независимо от центрального, широкий набор блоков памяти и внешних устройств. Модульность архитектуры кристалла ОМЭВМ позволяет в рамках одного семейства варьировать в широких пределах набор параметров кристалла: состав и объем блоков памяти, набор внешних устройств и даже тип помещаемого на кристалл центрального процессора.

Таким образом, пользователю предоставляется возможность выбора в очень широких пределах архитектуры и параметров ОМЭВМ. При этом он получает "готовую" ЭВМ, не требующую схемотехнических и архитектурных доработок. В итоге современные ОМЭВМ практически полностью заняли ту нишу, в которой долгое время существовали многокристальные микропроцессоры.

ГЛАВА 6



Базовая архитектура микропроцессорной системы

Пожалуй, наиболее популярными в мире (и в нашей стране) являлись и являются однокристальные микропроцессоры семейства *x*86 фирмы Intel. Семейство берет свое начало от первого 8-разрядного микропроцессора i8080 (отечественный аналог — К580ВМ80) и включает 16- и 32-разрядные микропроцессоры i8086, i80286, i80386, i80486, Pentium, ..., Pentium 4.

Схемотехнические решения систем на i8080 можно было бы считать базовыми, но его система команд значительно отличается от языка старших моделей микропроцессоров семейства. Поэтому "родоначальником" семейства принято считать первый 16-разрядный микропроцессор — i8086 (отечественный аналог — К1810ВМ86), на котором, кстати, были реализованы персональные ЭВМ IBM PC XT.

Анализ архитектуры микропроцессорных систем (МПС) целесообразно начинать с рассмотрения простейшей (базовой) модели, отражающей основные принципы организации процессора, его системы команд, функционирование основных подсистем. Большинство принципиальных решений, реализованных в МПС на базе младших моделей семейства, сохранились и в старших моделях.

Рассмотрим кратко организацию МПС на базе микропроцессора i8086. При этом выделим для рассмотрения следующие подсистемы:

- процессорный модуль;
- память;
- ввод/вывод;
- прерывания;
- прямой доступ в память со стороны ВУ.

6.1. Процессорный модуль

Процессорный модуль — основная часть любой МПС. Помимо собственно микропроцессора, он включает ряд вспомогательных схем, без которых МПС не может функционировать (тактовый генератор, интерфейсные схемы и др.).

6.1.1. Внутренняя структура микропроцессора

Структурная схема микропроцессора i8086 представлена на рис. 6.1. Микропроцессор включает в себя три основных устройства:

- УОД — устройство обработки данных;
- УСМ — *устройство связи с магистралью*;
- УУС — устройство управления и синхронизации.

УОД предназначено для выполнения команд и включает в себя 16-разрядное АЛУ, системные регистры и другие вспомогательные схемы; блок регистров (РОН, базовые и индексные) и блок микропрограммного управления.

УСМ обеспечивает формирование 20-разрядного физического адреса памяти и 16-разрядного адреса ВУ, выбор команд из памяти, обмен данными с ЗУ, ВУ, другими процессорами по магистрали. УСМ включает в себя сумматор адреса, блок регистров очереди команд и блок сегментных регистров.

УУС обеспечивает синхронизацию работы устройств МП, выработку управляющих сигналов и сигналов состояния для обмена с другими устройствами, анализ и соответствующую реакцию на сигналы других устройств МПС.

Микропроцессор i8086 может работать в одном из двух режимов — *минимальном* и *максимальном*. Минимальный режим предназначен для реализации однопроцессорной конфигурации МПС с организацией, подобной МПС на базе i8080, но с увеличенным адресным пространством, более высоким быстродействием и значительно расширенной системой команд. Максимальный режим предполагает наличие в системе нескольких микропроцессор, работающих на общую системную шину. МПС на базе i8086 с использованием максимального режима не получили широкого распространения. Более того, в последующих моделях своих микропроцессоров (80286, 80386, 80486) фирма Intel отказалась от поддержки мультипроцессорной архитектуры. Поэтому мы здесь не будем рассматривать особенности организации максимального режима.

На внешних выводах МП i8086 широко используется принцип *мультиплексирования сигналов* — передача разных сигналов по общим линиям с разде-

лением во времени. Кроме того, одни и те же выводы могут использоваться для передачи разных сигналов в зависимости от режима (min — max).

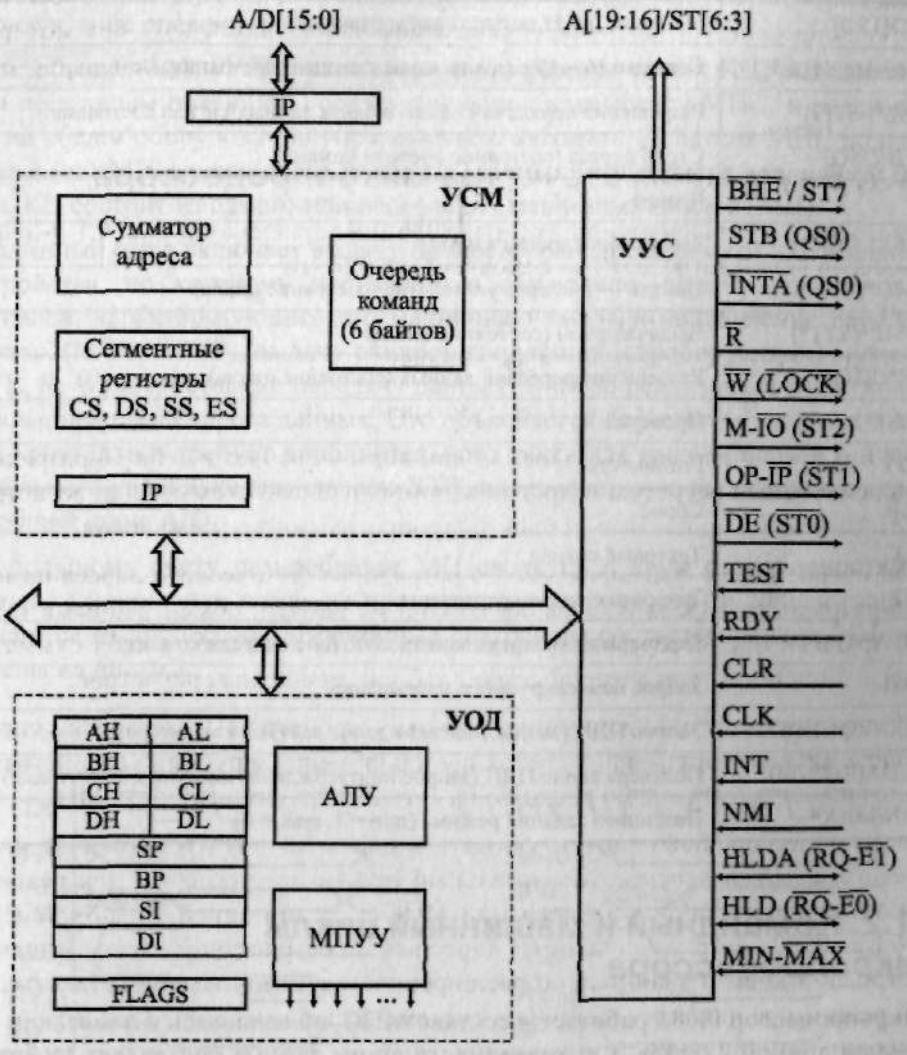


Рис. 6.1. *Структура микропроцессора i8086*

В табл. 6.1 приведено описание внешних выводов МП i8086. При описании выводов косой чертой (/) разделены сигналы, появляющиеся на выводе в разные моменты машинного цикла. В круглых скобках указаны сигналы, характерные только для максимального режима. Символ * после имени сигнала — знак его инверсии.

Таблица 6.1. Внешние выводы МП i8086

Внешний вывод	Описание
A/D[15:0]	Младшие 0—15 разряды адреса/данные
A[19:16]/ST[6:3]	Старшие 16—19 разряды адреса/сигналы состояния
BHE* /ST[7]	Разрешение передачи старшего байта данных/сигнал состояния
STB(QSO)	Строб адреса (состояние очереди команд)
R*	Чтение
W* /(LOCK*)	Запись (блокировка канала)
M-IO*(ST2*)	Память — внешнее устройство (состояние цикла)
OP-IP*(ST1*)	Выдача/прием (состояние цикла)
DE*(STO*)	Разрешение передачи данных (состояние цикла)
TEST*	Проверка
RDY	Готовность
CLR	Сброс
CLC	Тактовый сигнал
INT	Запрос внешнего прерывания
INTA*(QS1)	Подтверждение прерывания (состояние очереди команд)
NMI	Запрос немаскируемого прерывания
HLD(RQ*/EO)	Запрос ПДП (запрос/подтверждение доступа к магистрали)
NLDA(RQ* /EI)	Подтверждение ПДП (запрос/подтверждение доступа к магистрали)
MIN/MAX*	Потенциал задания режима (min = 1, max = 0)

6.1.2. Командный и машинный циклы микропроцессора

Микропроцессор i8086 работает в составе МПС, обмениваясь с памятью и ВУ словами длиной 2 байта, т. к. разрядность шины данных составляет 16 битов. В основе работы микропроцессора лежит *командный цикл* — действия по выбору из памяти и выполнению одной команды.

Любой командный цикл (КЦ) начинается с извлечения из памяти первого слова команды по адресу, хранящемуся в счетчике команд (PC). Команды i8086 могут иметь длину от 1 до 6 байтов, причем в первом слове содержится информация о длине команды. Таким образом, для извлечения из памяти одной команды может потребоваться одно или несколько обращений к ОЗУ.

В зависимости от типа и формата команды, способов адресации и числа операндов командный цикл может включать в себя различное число обращений к памяти и ВУ, поскольку кроме чтения самой команды в КЦ может потребоваться чтение операндов и размещение результата.

Хотя обращения к ЗУ/ВУ располагаются в разных частях КЦ, выполняются они по единым правилам, соответствующим интерфейсу МПС, и реализованы на общем оборудовании управляющего автомата. Действия МПС по передаче в (из) МП одного слова команды (данных) называются *машинным циклом*. КЦ состоит из одного или нескольких машинных циклов (МЦ).

Машинный цикл включает выдачу процессором адреса памяти или внешнего устройства, по которому производится обращение, выдачу управляющих сигналов, характеризующих тип машинного цикла и направление передачи данных (M-IO, OP-IP), выдачу синхронизирующих (стробирующих) сигналов (STB, R, W) и собственно передачу данных. В i8086 реализована мультиплексированная шина адреса/данных. Это объясняется дефицитом внешних выводов кристалла и требует дополнительного такта для выдачи адреса и дополнительного управляющего сигнала STB, идентифицирующего наличие адреса на общей шине A/D.

По большому счету разнообразие МЦ сводится к двум разновидностям — *чтению* (данные или команды принимаются в процессор) и *записи* (данные выдаются из процессора). Временные диаграммы соответствующих МЦ приведены на рис. 6.2.

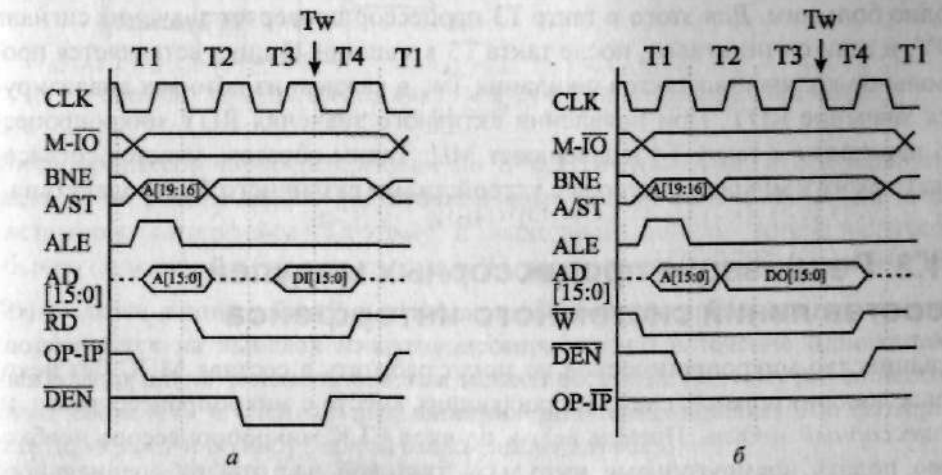


Рис. 6.2. Машинные циклы микропроцессора i8086: *а* — цикл ЧТЕНИЕ; *б* — цикл ЗАПИСЬ

Цикл начинается с формирования в такте T1 сигнала M-IO, определяющего тип устройства — память или ВУ, с которым осуществляется обмен данны-

ми. Длительность сигнала M-IO равна длительности машинного цикла, и он используется для селекции адреса устройств. В T1 и в начале T2 МП выдает адреса A[19:16] и A[15:0] и сигнал BHE, который вместе с A0 определяет выбор передачи либо всего слова, либо одного из его байтов. По спаду строба ALE адрес фиксируется во внешних регистрах-защелках. В такте T2 происходит переключение шин: на выводы A[19:16]/ST[6:3] поступают сигналы состояния; а выводы A/D[15:0] используются для приема/передачи данных.

Описанные выше машинные циклы являются *синхронными*; их длительность определяется только процессором. Однако такой обмен возможен лишь с устройствами, быстродействие которых не уступает процессорному. В противном случае микропроцессор должен реализовать *асинхронный* способ обмена, включающий анализ сигнала от устройства о готовности к обмену или о завершении процедуры обмена.

Роль такого сигнала в i8086 (и всех процессорах старших моделей семейства x86) играет вход RDY (от англ. *ready* — готовность), который всегда должен быть активным при синхронном обмене (с "быстрыми" устройствами). При обмене с "медленными" устройствами значение RDY должно оставаться неактивным (в разных процессорах активным для RDY может быть уровень логической 1 или логического 0) до тех пор, пока устройство, с которым связывается процессор, не завершит процедуру обмена, сообразуясь со своим быстродействием.

Время ожидания процессором готовности устройства может быть сколь угодно большим. Для этого в такте T3 процессор проверяет значение сигнала RDY, и если он неактивен, после такта T3 в машинный цикл вставляется произвольное количество тактов ожидания Tw, в каждом из которых анализируется значение RDY. При появлении активного значения RDY микропроцессор переходит к такту T4 и завершает МЦ. Таким образом, удается согласовывать работу микропроцессора с устройствами различного быстродействия.

6.1.3. Реализация процессорных модулей и состав линий системного интерфейса

Большинство микропроцессоров не могут работать в составе МПС без некоторых дополнительных схем, составляющих вместе с микропроцессором т. н. *процессорный модуль*. Прежде всего, на вход CLK микропроцессора необходимо подать прямоугольные импульсы тактовой частоты от специального внешнего тактового генератора.

Для микропроцессора i8086 частота тактовых импульсов может лежать в диапазоне 2—6 МГц.

На рис. 6.3 приведен один из вариантов упрощенной функциональной схемы процессорного модуля на базе i8086. На схеме не показаны некоторые элементы и связи (например, схема начального сброса и др.).

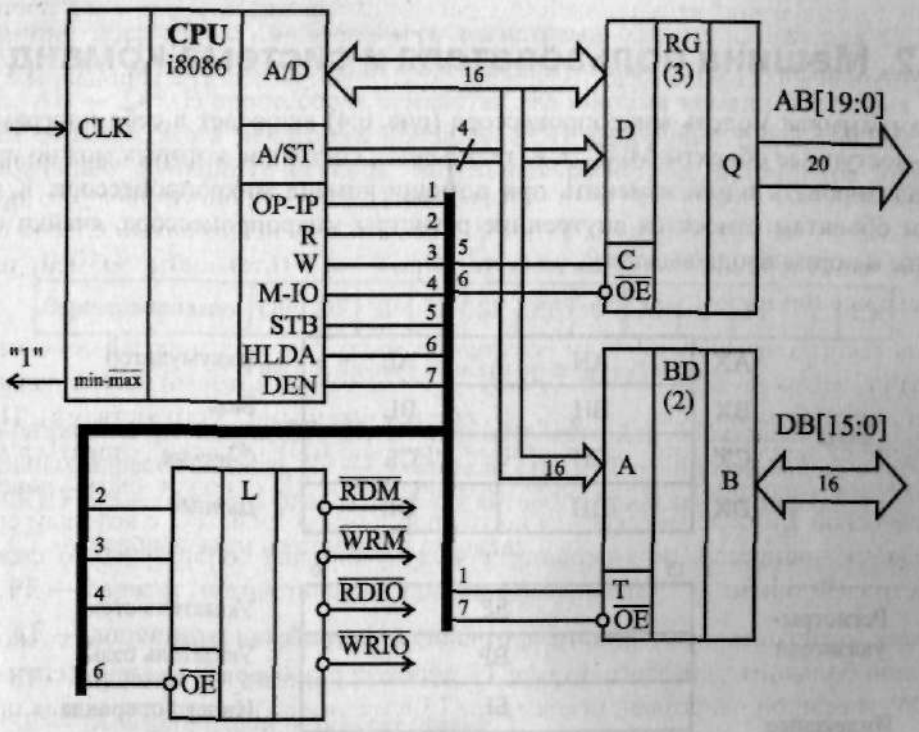


Рис. 6.3. Структура процессорного модуля на базе микропроцессора i8086

Микропроцессор i8086 реализован по n-МДП-технологии, и его выходные каскады не обеспечивают достаточной нагрузочной способности для линий системного интерфейса. Поэтому к выходным линиям микропроцессора обычно подключают буферные схемы BD, реализованные по технологии ТТЛ.

Кроме того, шины адреса и данных в i8086 мультиплексированы. Адрес удерживается на выводах микропроцессора только в течение одного такта машинного цикла, а использоваться должен весь МЦ. Поэтому адрес необходимо запомнить в специальных внешних регистрах-защелках RG (которые, кстати, играют и роль буферной схемы шины адреса).

Наконец, часто требуется преобразовать управляющие сигналы, выдаваемые микропроцессором, в стандартные сигналы системного интерфейса. Так, i8086 формирует выходные сигналы, идентифицирующие тип машинного цикла, и сигналы стробирования: M-IO, OP-IP, R, W. Системная шина ис-

пользует сигналы записи и чтения памяти — RDM, WRM и записи и чтения внешнего устройства — RDIO, WRIO. Преобразования процессорных сигналов в шинные осуществляет простая логическая схема L.

6.2. Машина пользователя и система команд

Программная модель микропроцессора (рис. 6.4) включает в себя программно-доступные объекты МПС, т. е. те объекты, состояние которых можно проанализировать и/или изменить при помощи команд микропроцессора. К таким объектам относятся внутренние регистры микропроцессора, ячейки памяти и порты ввода/вывода.

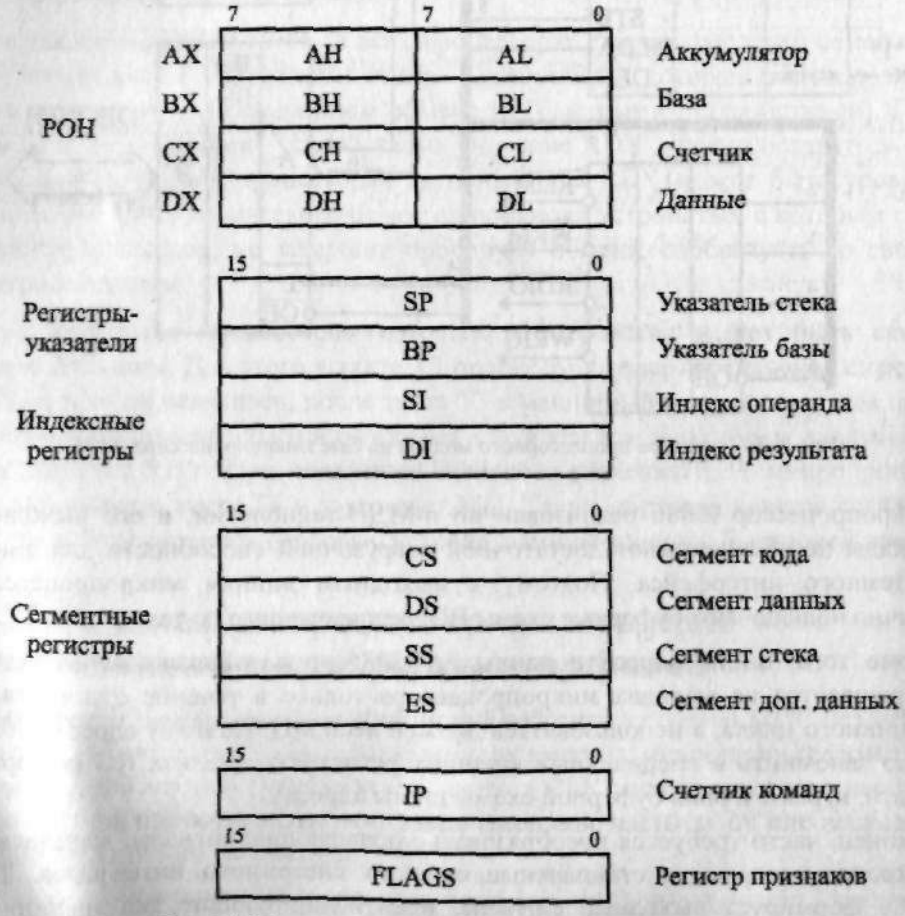


Рис. 6.4. Микропроцессор i8086 — машина пользователя

Рассмотрим *машину пользователя* i8086. Кроме показанных на рис. 6.4 регистров процессора, в машину пользователя i8086 включатся адресное пространство памяти объемом 1 Мбайт и два пространства портов ввода и вывода по 64 Кбайт каждое.

Помимо операций с 16-разрядными регистрами общего назначения (РОН) AX — DX, допускается обращение к каждому байту этих регистров: AL — DL, AH — DH. В процессорах семейства *x*86 система команд построена таким образом, что в некоторых командах РОН выполняют определенные по умолчанию функции счетчиков, индексных регистров, источников адреса и др.

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Зарезервировано				OF	DF	IF	TF	SF	ZF	0	AF	0	PF	1	CF

Рис. 6.5. Формат регистра признаков i8086

16-разрядные регистры BP, SI, DI используются для образования исполнительных адресов памяти, SP — указатель стека, IP — программный счетчик (СчК), Flags — регистр флагов, формат которого приведен на рис. 6.5, где:

- CF — перенос/заем из старшего разряда;
- PF — паритет (четность числа единиц в результате);
- AF — дополнительный перенос (из 3-го разряда);
- ZF — нулевой результат;
- SF — отрицательный результат (знак);
- OF — признак арифметического переполнения;
- DF — направление, определяет направление модификации адресов массивов в командах цепочек (увеличение или уменьшение адреса);
- IF — маскирует внешнее прерывание по входу INT (при IF = 1 прерывание разрешено);
- TF — управляет пошаговым режимом работы микропроцессора.

При TF = 1 после выполнения каждой команды автоматически формируется прерывание с вектором 1.

6.2.1. Распределение адресного пространства

Адресное пространство МП определяется в i8086 разрядностью шины адреса/данных и адреса и составляет 2^{20} байтов = 1 Мбайт. В этом адресном пространстве микропроцессору одновременно доступны лишь четыре сегмента,

два из которых (DS и ES) предназначены для размещения данных, CS — сегмент кода (для размещения программы) и SS — сегмент стека.

Размеры сегментов определяются разрядностью логических адресов команд, данных и стека. Логические адреса команд и стека (верхушки) хранятся в 16-разрядных регистрах IP и SS соответственно, а логический адрес данных вычисляется в команде одним из многочисленных, предусмотренных системой команд, способов и также составляет 16 битов.

Таким образом, размер каждого сегмента в i8086 составляет 2^{16} байтов = = 64 Кбайт. Положение сегмента в адресном пространстве (его начальный адрес) определяется содержимым одноименного сегментного регистра. Формирование физического адреса иллюстрируется на рис. 6.6, из которого видно, что граница сегмента в адресном пространстве может быть установлена не произвольно, а таким образом, чтобы начальный адрес сегмента был кратен 16.

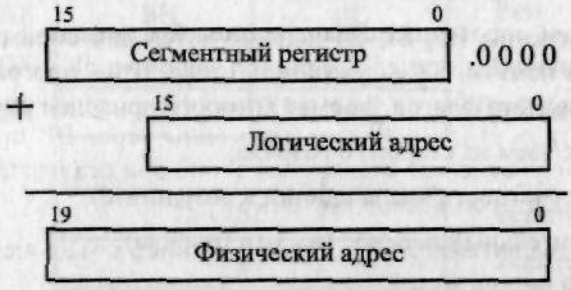


Рис. 6.6. Формирование физического адреса в i8086

По умолчанию сегментные регистры выбираются для образования физического адреса следующим образом: при считывании команды по адресу IP используется CS, при обращении к данным — DS или ES, при обращении к стеку — SS. С помощью специальных приставок к команде (префиксов) можно назначить для использования произвольный сегментный регистр (кроме пары CS:IP, *которая не подлежит модификации*). *Границы сегментов могут быть* выбраны таким образом, что сегменты будут изолированы друг от друга, пересекаться или даже полностью совпадать. Например, если загрузить CS = SS = DS = ES = 0, то все сегменты будут совпадать друг с другом и начинаться с нулевого адреса — вариант организации адресного пространства i8080.

6.2.2. Система команд i8086

Система команд i8086 и, вообще, всего семейства *x*86 подробно описана в многочисленных справочниках и руководствах, например [11, 12, 13], поэто-

му далее мы кратко остановимся только на особенностях системы команд i8086, не вдаваясь в излишние подробности.

i8086 отличается разнообразием форматов команд и способов адресации. Длина команды может составлять от 1 до 6 байтов, причем в первых двух байтах (иногда — в первом) определяется код операций, количество и длина операндов и способ их адресации. В остальных байтах команды могут размещаться непосредственный операнд, прямой адрес или смещение.

Большинство команд i8086 являются *двухадресными*, причем один адрес определяет регистр процессора, а другой — ячейку памяти или регистр.

Операнд в памяти может адресоваться *прямо* или *косвенно* посредством содержимого базовых (BP, BX) или индексных (SI, DI) регистров, а также их суммы. Предусмотрены многочисленные варианты относительной адресации, при которых логический адрес образуется как сумма двух или трех слагаемых — одного или двух регистров процессора и 8- или 16-разрядного смещения, размещаемого в команде.

Режимы адресации спроектированы с учетом эффективной реализации языков высокого уровня. Например, к простой переменной можно обратиться в режиме прямой адресации, а к элементу массива — в режиме косвенной адресации посредством BX, SI. Режим адресации через BP предназначен для доступа к данным из сегмента стека, что удобно при реализации рекурсивных процедур и компиляторов языков высокого уровня.

Система команд насчитывает 113 базовых команд, объединенных в следующие группы:

- команды передачи данных:
  - между регистрами и памятью (включая стек), обмен содержимым источника и приемника;
  - ввод, вывод, табличное преобразование;
  - загрузка исполнительного адреса в РОН, загрузка 4-байтового адресного объекта в регистры-указатели (начальный адрес сегмента и смещение в сегменте);
  - передача содержимого регистра F флагов в память, в стек и из стека;
- арифметические команды:
  - сложение, вычитание, умножение и деление двоичных чисел со знаком и без знака (произведение и делимое представляются числами двойной длины);
  - десятичная коррекция сложения и вычитания упакованных двоично-десятичных чисел;

- десятичная коррекция сложения, вычитания, умножения и деления распакованных двоично-десятичных чисел;

- логические команды и сдвиги:
  - инверсия, конъюнкция, дизъюнкция, неравнозначность;
  - TEST — поразрядная конъюнкция операндов с установкой флагов, но без занесения результатов;
  - сдвиги на 1 или заданное число разрядов (константа сдвига располагается в CL);
- команды передачи управления: переходы, вызовы, возвраты имеют две разновидности — *внутрисегментные* ("близкие") и *межсегментные* ("дальние"). При близких передачах загружается только IP, при дальних — IP и CS. Передачи управления могут быть прямыми (целевой адрес — в команде) или косвенными (целевой адрес вычисляется с использованием стандартных режимов адресации). В 16 командах условных переходов проверяются отношения знаковых и беззнаковых чисел. Имеются 4 команды управления циклами, которые рассчитаны на размещение числа повторений цикла в регистре CX;
- команды обработки цепочек данных манипулируют последовательностями байтов или слов в памяти. Время обработки цепочек этими командами гораздо меньше, чем соответствующей программной реализацией.

6.3. Функционирование основных подсистем МПС

Теперь можно рассмотреть функционирование основных подсистем базовой МПС с интерфейсом типа *"общая шина"*. Этот термин используется в двух смыслах: во-первых, как обозначение принципа организации связи процессора с другими устройствами в составе ЭВМ, во-вторых, как обозначение (в русском переводе) конкретного интерфейса Unibus мини-ЭВМ семейства PDP-11 фирмы DEC.

Unibus явился, пожалуй, первым интерфейсом, в котором были полностью реализованы принципы "общей шины":

- все линии интерфейса делятся на три группы: *адрес*, *данные*, *управление*;
- все устройства, в т. ч. процессор, подключаются к линиям интерфейса одинаковым образом;
- идентификация объектов на шине (ячеек памяти, регистров внешних устройств) осуществляется с помощью уникального для каждого объекта *адреса*;

- в каждый момент времени по шине могут взаимодействовать только два устройства, одно из которых является *активным*, а другое — *пассивным*. Активное устройство формирует адрес обмена, управляющие сигналы и может выдавать (в цикле *записи*) или принимать (в цикле *чтения*) данные, которые принимает или выдает пассивное устройство;
- обмен между устройствами может осуществляться в синхронном или асинхронном режиме. При *синхронном обмене* все временные характеристики обмена определяются только активным устройством, которое не анализирует ни готовность пассивного к обмену, ни факт завершения обмена. Синхронный обмен допустим лишь с быстродействующими пассивными устройствами (их быстродействие должно быть не ниже быстродействия активного устройства), которые всегда готовы к обмену (например, регистр двоичной индикации). При *асинхронном обмене* пассивное устройство формирует сигнал готовности к обмену и/или сигнал завершения обмена, которые анализирует активное устройство.

Интерфейсы, реализующие принципы "общей шины", широко распространились в мини- и микроЭВМ, МПС различного назначения. Многие из них, правда, нарушали некоторые принципы "канонической общей шины", например, за счет появления отдельных адресных пространств регистров процессора и портов ввода и вывода. Однако основные принципы, изложенные выше, сохраняются в многочисленных разновидностях таких интерфейсов.

К достоинствам интерфейсов типа "общая шина" можно отнести его относительную простоту, гибкость системы и возможность ее модификации в широких пределах.

К недостаткам — невозможность распараллеливания процессов обмена (одновременно осуществляется связь только пары устройств). Кроме того, наличие на общей шине устройств с существенно различным быстродействием затрудняет достижение оптимальных характеристик системы.

В *разд. 6.1.3* мы подробно рассмотрели организацию процессорного модуля на базе процессора i8086. Состав внешних выводов процессорного модуля (см. рис. 6.3) позволяет подключить его к интерфейсу, реализованному по принципу общей шины:

- линии адреса AB[19:0];
- линии данных DB[15:0];
- линии управления RDM, WRM, RDIO, WRIO.

Другие линии управления, входящие в состав интерфейса, будут добавлены при рассмотрении соответствующих подсистем.

6.3.1. Оперативная память

Объем адресного пространства МПС с интерфейсом "общая шина" определяется главным образом разрядностью шины адреса и, кроме того, номенклатурой управляющих сигналов интерфейса. Управляющие сигналы могут определять тип объекта, к которому производится обращение (ОЗУ, ВУ, стек, специализированные ПЗУ и др.). В случае, если МП не выдает сигналов, идентифицирующих тип пассивного устройства (или они не используются в МПС) — для селекции берутся только адресные линии. Число адресуемых объектов составляет в этом случае 2^k, где k — разрядность шины адреса. Будем называть такое адресное пространство *единым*. Иногда говорят, что ВУ в едином адресном пространстве "отображены на память", т. е. адреса ВУ занимают адреса ячеек памяти.

При использовании информации о типе устройства, к которому идет обращение, одни и те же адреса можно назначать для устройств разных типов, осуществляя селекцию с помощью управляющих сигналов.

Так, большинство МП выдают в той или иной форме информацию о типе обращения. В результате в большинстве интерфейсов присутствуют отдельные *управляющие линии для обращения к памяти и вводу/выводу, реже — к стеку или специализированному ПЗУ*. В результате суммарный объем адресного пространства МПС может превышать величину 2^k.

Например, *системная шина МПС на базе микропроцессора i8086 включает 20-разрядную шину адреса и управляющие сигналы, идентифицирующие обращение к памяти* (RDM, WRM) и вводу/выводу (RDIO, WRIO). Поэтому в системе доступны 1 Мбайт ячеек памяти (адреса 00000 — FFFFF) + 64 Кбайт адресов ввода + 64 Кбайт адресов вывода (0000 — FFFF). Последняя величина определяется тем, что в командах ввода/вывода процессоров семейства *x*86 адрес внешнего устройства имеет разрядность 16 битов.

Диспетчер памяти

При необходимости расширить объем памяти за пределы адресного пространства можно воспользоваться т. н. *диспетчером памяти*. В простейшем случае он представляет собой программно-доступный регистр, который должен располагаться в пространстве ввода/вывода. В него заносится номер активного в данный момент банка памяти, причем объем банка может равняться объему адресного пространства МП.

Очевидно, в каждый момент времени процессору доступен только один банк. При необходимости перехода в другой банк памяти МП должен предварительно выполнить программную процедуру (часто всего одну команду) перезагрузки содержимого регистра номера банка.

К развитию этой идеи можно отнести механизм сегментации памяти в 16- и 32-разрядных МП фирмы Intel.

6.3.2. Ввод/вывод

Подсистема ввода/вывода (ПВВ) обеспечивает связь МП с внешними устройствами, к которым будем относить:

- устройства ввода/вывода (УВВ): клавиатура, дисплей, принтер, датчики и исполнительные механизмы, АЦП, ЦАП, таймеры и т. п.;
- внешние запоминающие устройства (ВЗУ): накопители на магнитных дисках, "электронные диски", CD и др.

В рамках рассмотрения ПВВ будем полагать термины "УВВ" и "ВУ" синонимами, т. к. обращение к ним со стороны процессора осуществляется по одним законам.

ПВВ в общем случае должна обеспечивать выполнение следующих функций:

- согласование форматов данных, поскольку процессор всегда выдает/принимает данные в параллельной форме, а некоторые ВУ — в последовательной. С этой точки зрения различают устройства *параллельного* и *последовательного обмена*. В рамках параллельного обмена не производится преобразование форматов передаваемых слов, в то время как при последовательном обмене осуществляется преобразование параллельного кода в последовательный и наоборот. Все варианты, при которых длина слова ВУ (больше 1 бита) не совпадает с длиной слова МП, сводятся к разновидностям параллельного обмена;
- организация режима обмена — формирование и прием управляющих сигналов, идентифицирующих наличие информации на различных шинах, ее тип, состояние ВУ (Готово, Занято, Авария), регламентирующих временные параметры обмена. По способу связи процессора и ВУ (активного и пассивного) различают *синхронный* и *асинхронный обмены*, различия между которыми мы обсудили в начале настоящей главы.

Параллельный обмен

Простейшая подсистема параллельного обмена должна обеспечить лишь дешифрацию адреса ВУ и электрическое подключение данных ВУ к системной шине данных DB по соответствующим управляющим сигналам. На рис. 6.7 показаны устройства параллельного ввода и вывода информации в составе МПС на базе буферных регистров К580ИР82.

Очевидно, при обращении процессора (он в подобных циклах играет роль активного устройства) к *устройству ввода*, адрес соответствующего регистра

помещается процессором на шину адреса и формируется управляющий сигнал RDIO. Дешифратор адреса, включающий и линию RDIO, при совпадении адреса и управляющего сигнала активизирует выходные линии регистра и его содержимое поступает по шине данных в процессор.

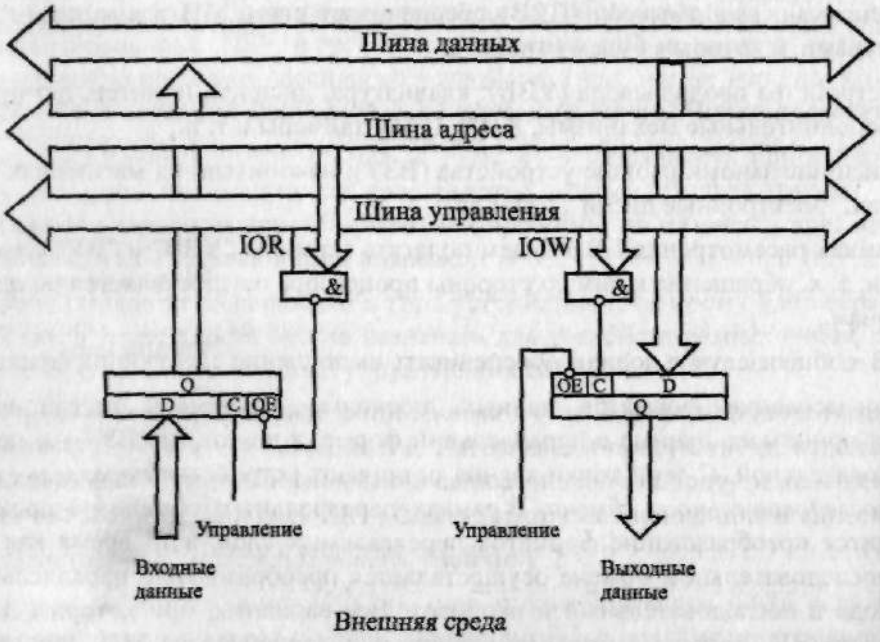


Рис. 6.7. Параллельный обмен на базе буферных регистров

Аналогично идет обращение к *устройству вывода*. Совпадение адреса устройства на шине адреса с активным уровнем сигнала WRIO обеспечивает "защелкивание" состояния шины данных в регистре вывода.

Характерно, что при таком способе обмена процессор не анализирует готовность ВУ к обмену, а длительность существования адреса, данных и управляющего сигнала целиком определяется тактовой системой процессора и принятым алгоритмом командного цикла. Напомним, что такой способ обмена называется *синхронным*. Синхронный обмен реализуется наиболее просто, но он возможен только с устройствами, всегда готовыми к обмену, либо процессор должен перед выполнением команды ввода/вывода программными средствами убедиться в готовности ВУ к обмену (обычно в этом случае предварительно анализируется состояние флага готовности, формируемого ВУ). Кроме того, быстродействие ВУ, взаимодействующее с процессором в синхронном режиме, должно гарантировать прием/выдачу данных за фиксированное время, выделенное процессором на цикл обмена.

Во многих микропроцессорных комплектах выпускают специальные интерфейсные БИС, существенно расширяющие (по сравнению с использованием регистров) возможности разработчиков при организации параллельного обмена в МПС. Такие БИС обычно имеют несколько каналов передачи информации, позволяют программировать направление передачи (ввод или вывод) по каждому каналу и выбирать способ обмена — синхронный или асинхронный.

Типичным примером такой БИС может служить программируемый контроллер параллельного обмена (далее "контроллер") 8255A (отечественный аналог — К580ВВ55).

Контроллер параллельного обмена К580ВВ55 [13] представляет собой трехканальный байтовый интерфейс и позволяет организовать обмен байтами с периферийным оборудованием в различных режимах. Он включает в себя три 8-разрядные канала ввода/вывода A, B и C, буфер шины данных, 8-разрядный регистр управления Y и блок управления.

Подключение контроллера к системной шине показано на рис. 6.8. Каналы адресуются двумя линиями адреса A1, A0. В МПС контроллер размещают, как правило, в пространстве адресов ввода/вывода. Поэтому в качестве стробов чтения и записи используются сигналы RDIO, WRIO, для селекции контроллера по CS дешифрируются старшие разряды адреса, а для выбора адресуемого объекта внутри контроллера — два младших.

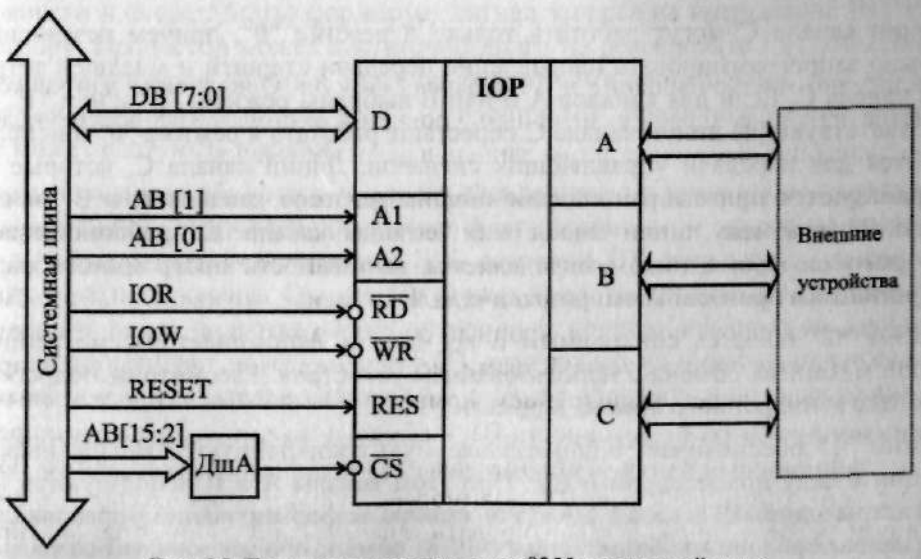


Рис. 6.8. Подключение контроллера 8255 к системной шине

Каналы контроллера программируются для работы в одном из трех режимов:

- режим "0" — синхронный однонаправленный ввод/вывод;
- режим "1" — асинхронный однонаправленный ввод/вывод;
- режим "2" — асинхронный двунаправленный ввод/вывод.

Режим работы контроллера устанавливается кодом управляющего слова, которое предварительно записывается в регистр управления Y.

В *режиме "0"* контроллер может работать как четыре порта ввода/вывода: A[7:0], B[7:0], C[7:4], C[3:0], причем каждый порт может быть независимо запрограммирован на ввод или на вывод. При этом к порту, определенному как выходной, нельзя обращаться по чтению, а на входной порт нельзя выводить информацию.

В асинхронном однонаправленном *режиме "1"* могут работать только каналы A и B, причем соответствующие линии канала C придаются каналам A и B для передачи управляющих сигналов. Как и в режиме "0", каналы A и B программируются на ввод или вывод (независимо).

В *режиме "2"* может работать только канал A, к которому в этом случае можно обращаться как по записи, так и по чтению (двунаправленный асинхронный обмен). При этом канал B может быть запрограммирован как на работу в режиме "1", так и в режиме "0".

Выбор режимов каналов и направления передачи данных в них осуществляется загрузкой во внутренний управляющий регистр Y соответствующего кода.

Линии канала C могут работать только в режиме "0", причем независимо можно запрограммировать направление передачи старшей и младшей тетрады канала C. Если для каналов A и/или B выбраны режимы "2" и/или "1", то соответствующие линии канала C перестают работать в режиме "0" и используются для передачи управляющих сигналов. Линии канала C, которые не используются при выбранной комбинации режимов каналов A и B, можно использовать как линии ввода или вывода канала C, работающего в "0"-режиме. Кроме того, всегда имеется возможность программного сброса/установки произвольного разряда канала C.

Режим "0" является синхронным и во многом напоминает рассмотренный выше механизм обмена с использованием регистров. Рассмотрим подробнее процесс асинхронного обмена в режиме "1".

Режим "1" обеспечивает однонаправленную асинхронную передачу информации между процессором и ВУ. При этом каналы A и B используются как регистры данных, а канал C — для приема и формирования управляющих сигналов, сопровождающих асинхронный обмен, причем каждый разряд канала C имеет строго определенное функциональное назначение [13].

Например, если канал запрограммирован на ввод в режиме "1", то процессор может вводить данные этого канала только "будучи уверенным" в их готовности. Об этой готовности ему должен сообщить контроллер путем установки специального признака — флага в определенном разряде регистра С и, может быть, формированием запроса на прерывание с соответствующим вектором (о прерываниях подробнее *см. в разд. 6.3.3*). С другой стороны, внешнее устройство, подключенное к каналу, не должно выдавать новую порцию информации, пока прежняя не будет прочитана процессором.

Для обеспечения синхронизации *ввода* в режиме "1" каналу придаются три линии канала С для передачи управляющих сигналов:

- STB (строб записи) — сигнал, формируемый ВУ для записи очередного байта данных в регистр канала:
- IBF (подтверждение приема) — сигнал, формируемый контроллером для ВУ в тот момент, когда процессор прочитал содержимое регистра канала. Пока сигнал IBF неактивен, ВУ запрещается вырабатывать новый строб записи;
- INT (запрос прерывания) — вырабатывается контроллером для процессора после того, как очередной байт данных запишется в регистр канала. Это же событие устанавливает флаг готовности канала в разряде регистра С.

Обмен начинается с подачи ВУ сигнала STB, по которому данные помещаются в регистр канала. Контроллер, во-первых, сбрасывает сигнал IBF, запрещая ВУ выработку нового строба, и, во-вторых, устанавливает флаг готовности и (может быть) формирует сигнал запроса на прерывание INT процессору. Процессор может достаточно долго не реагировать на сообщение о готовности канала, занятый более приоритетными процедурами. Все это время установлены готовность и INT и сброшен IBF, новая порция информации не может поступить в канал.

Когда процессор обратится по адресу канала и введет хранящуюся в регистре информацию, контроллер сбрасывает флаг готовности и запрос на прерывание INT и устанавливает сигнал IBF, разрешая ВУ записывать следующий байт в регистр канала. Однако ВУ может быть достаточно инерционным и довольно долго подготавливает следующую порцию информации, но пока ВУ не сформирует новый сигнал STB, контроллер не выработает сигнал готовности и, следовательно, процессор не будет обращаться по адресу канала.

Подобный режим обмена позволяет исключить как потерю информации в контроллере, так и повторный ввод в процессор прежней информации.

Аналогично реализуется и асинхронный режим вывода. Каналу, запрограммированному на вывод в режиме "1", придаются три линии управления канала С:

- OBF (выходной буфер заполнен) — сигнал формируется контроллером для ВУ после того, как процессор записал в регистр канала новую порцию информации;
- ACK (подтверждение записи) — сигнал от ВУ контроллеру, подтверждающий прием очередного байта;
- INT (запрос прерывания) — запрос прерывания от контроллера процессору для выдачи процессором в канал следующего байта информации.

Процедуры ввода и вывода в режиме "2" осуществляются аналогично соответствующим процедурам в режиме "1".

Последовательный обмен

При организации последовательного обмена ключевыми могут считаться две проблемы:

- синхронизация битов передатчика и приемника;
- фиксация начала сеанса передачи.

Различают два способа передачи последовательного кода: *синхронный* и *асинхронный*.

При синхронном методе передатчик генерирует две последовательности — информационную TxD и синхроимпульсы CLK, которые передаются на приемник по разным линиям. Синхроимпульсы обеспечивают синхронизацию передаваемых битов, а начало передачи отмечается по-разному. При организации *внешней синхронизации* (рис. 6.9) сигнал начала передачи BD генерируется передатчиком и передается на приемник по специальной линии.

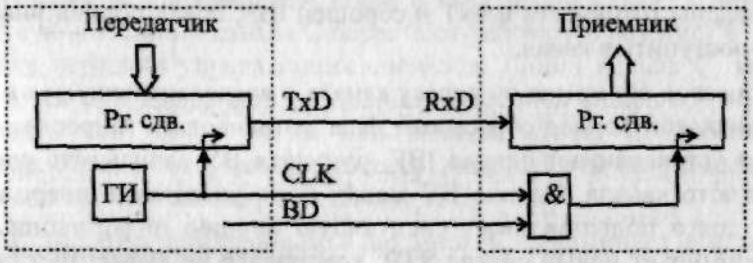


Рис. 6.9. Последовательный синхронный обмен

Системы с *внутренней синхронизацией* генерируют на линию данных специальные коды длиной 1—2 байта — символы синхронизации. Для каждого приемника предварительно определяются конкретные синхросимволы, таким образом можно осуществлять адресацию конкретного абонента из нескольких, работающих на одной линии. Каждый приемник постоянно принимает

биты с линии RxD, формирует символы и сравнивает с собственными синхросимволами. При совпадении принятых символов с заданными для этого приемника синхросимволами последующие биты поступают в канал данных приемника. В случае реализации внутренней синхронизации между приемником и передатчиком "прокладывают" только две *линии — данных и синхро*-импульсов.

Наконец, при *асинхронном* способе обмена можно ограничиться одной линией — данных. Для надежной синхронизации обмена в асинхронном режиме:

- передатчик и приемник настраивают на работу с одинаковой частотой;
- передатчик формирует стартовый и стоповый биты, отмечающие начало и конец посылки;
- передача ведется короткими посылками (5—9 битов), а частоты передачи выбираются сравнительно низкими.

Принцип последовательного асинхронного обмена по единственной линии показан на рис. 6.10. Пока передачи нет, на линии передатчик удерживает высокий уровень (H). Передача начинается с выдачи в линию стартового бита низкого уровня (длительности всех битов τ одинаковы и определяются частотой передатчика $f_T = 1/\tau$).

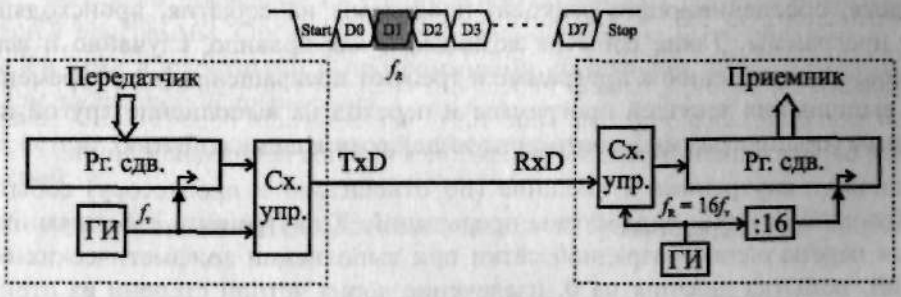


Рис. 6.10. Последовательный асинхронный обмен

Частота приемника f_R устанавливается равной $16 \times f_T$. Когда приемник обнаруживает на линии перепад H → L, он включает счетчик тактов до 16, причем еще дважды за период τ проверяет состояние линии. Если низкий уровень (L) подтверждается, приемник считает, что принял старт-бит, и включает счетчик принимаемых битов. Если во второй и третьей проверке на линии определяется H-уровень, то перепад считается помехой и старт-бит не фиксируется.

Каждый последующий (информационный) бит принимается таким образом, что за период τ трижды проверяется состояние линии (например, в 3, 8 и 11 тактах приемника) и значение принимаемого бита определяется по мажо-

ритарному принципу. Принятый бит помещается слева в сдвиговый регистр приемника. После принятия последнего информационного бита (количество битов в посылке определяется протоколом обмена и составляет обычно от 5 до 9) обязательно должен последовать стоповый бит Н-уровня. Во время поступления стоп-бита содержимое сдвигового регистра приемника передается в память, а в регистр передатчика может загружаться новая порция информации для передачи. Отсутствие стопового бита воспринимается приемником как ошибка передачи посылки.

После стопового бита можно формировать стартовый бит новой посылки или "держать паузу" произвольной длительности, при которой на линии присутствует Н-уровень.

Наличие стартового бита позволяет в начале каждой посылки синхронизировать фазы приемника и передатчика, компенсировав неизбежный уход фаз передатчика и приемника. Короткие посылки и относительно низкая частота передачи позволяют надеяться, что неизбежное рассогласование частот передатчика и приемника не приведет к ошибкам при передаче посылки.

6.3.3. Прерывания

Подсистема прерываний — совокупность аппаратных и программных средств, обеспечивающих реакцию программы на события, происходящие вне программы. Такие события возникают, как правило, случайно и асинхронно по отношению к программе и требуют прекращения (чаще временного) выполнения текущей программы и переход на выполнение другой программы (подпрограммы), соответствующей возникшему событию.

Различают внутренние и внешние (по отношению к процессору) события, требующие реакции подсистемы прерываний. К внутренним событиям относятся переполнение разрядной сетки при выполнении арифметических операций, попытка деления на 0, извлечение корня четной степени из отрицательного числа, появление несуществующего кода команды, обращение программы в область памяти, для нее не предназначенную, сбой при выполнении передачи данных или операции в АЛУ и многое другое. Внутренние прерывания должны обеспечиваться развитой системой аппаратного контроля процессора, поэтому они не получили широкого распространения в простых 8- и 16-разрядных МП.

Внешние прерывания могут возникать во внешней по отношению к процессору среде и отмечать как аварийные ситуации (кончилась бумага на принтере, температура в реакторе превысила допустимый уровень, исполнительный орган робота дошел до предельного положения и т. п.), так и нормальные рабочие события, которые происходят в случайные моменты времени (нажата клавиша, исчерпан буфер принтера или ВЗУ и т. п.). Во всех этих случаях

требуется прервать выполнение текущей программы и перейти на выполнение другой программы (подпрограммы), обслуживающей данное событие.

С точки зрения реализации внутренние и внешние прерывания функционируют одинаковым образом, хотя при работе подсистемы с внешними прерываниями возникают дополнительные проблемы идентификации источника прерывания. Поэтому ниже остановимся на рассмотрении внешних прерываний.

Анализ состояния внешней среды можно осуществлять путем программного сканирования — считывания через определенные промежутки времени слов состояния всех возможных источников прерываний, выделения признаков отслеживаемых событий и переход (при необходимости) на прерывающую подпрограмму (часто ее называют *обработчиком прерывания*).

Однако такой способ не обеспечивает для большинства применений приемлемого времени реакции системы на события, особенно при необходимости отслеживания большого числа событий. К тому же при коротком цикле сканирования большой процент процессорного времени тратится на проверку (чаще безрезультатную) состояния внешней среды.

Гораздо эффективней организовать взаимодействие с внешней средой таким образом, чтобы всякое изменение состояния среды, требующее реакции МПС, вызывало появление на специальном входе МП сигнала прерывания текущей программы. Организация прерываний должна быть обеспечена определенными аппаратными и программными средствами, которые мы и называем *подсистемой прерываний*.

Подсистема прерываний должна обеспечивать выполнение следующих функций:

- обнаружение изменения состояния внешней среды (запрос на прерывание);
- идентификация источника прерывания;
- разрешение конфликтной ситуации в случае одновременного возникновения нескольких запросов (приоритет запросов);
- определение возможности прерывания текущей программы (приоритет программ);
- фиксация состояния прерываемой (текущей) программы;
- переход к программе, соответствующей обслуживаемому прерыванию;
- возврат к прерванной программе после окончания работы прерывающей программы.

Рассмотрим варианты реализации в МПС перечисленных выше функций.

Обнаружение изменения состояния внешней среды

Фиксация изменения состояния внешней среды может осуществляться различными схемами: двоичными датчиками, компараторами, схемами формирования состояний и др. Будем полагать, что все эти схемы формируют в конечном итоге *логические сигналы* запроса на прерывание z, причем для определенности будем считать, что активное состояние этого сигнала передается *уровнем логической единицы* (Н-уровень).

Количество источников запросов в МПС может быть различно, в т. ч. и довольно велико. Дефицит внешних выводов МП в общем случае исключает возможность передачи каждого запроса от ВУ по "собственной" линии интерфейса. Обычно на одну линию запроса подключается несколько источников прерываний (по функции ИЛИ), а иногда и все источники запросов — на единственный вход.

Управляющий автомат процессора должен периодически анализировать состояние линии (линий) запросов на прерывания. Каким образом выбирается период проверки? С одной стороны, этот период должен быть коротким, чтобы обеспечить быструю реакцию системы на события. С другой стороны, при переходе на обслуживание прерывания требуется сохранить текущее состояние процессора на момент прерывания, с тем, чтобы, завершив программу-обработчик, продолжить выполнение прерванной программы "с того же места", на котором произошло прерывание.

Напомним, что в основе работы процессора лежит *командный цикл (см. разд. 2.1)*, состоящий, в свою очередь, из *машинных циклов*, каждый из которых длится несколько *тактов*. Осуществлять прерывание в произвольном такте невозможно, т. к. при этом пришлось бы сохранять в качестве контекста прерванной программы состояние всех элементов памяти процессора.

Прерывание по завершению текущего машинного цикла требует сохранение текущего состояния незавершенной команды. Например, при возникновении прерывания в том месте командного цикла, когда из памяти выбраны код команды и первый операнд, а для второго операнда только сформирован исполнительный адрес, следует сохранить: содержимое программного счетчика PC, код команды, первый операнд и адрес второго операнда. Сохранение этой информации требует как дополнительных аппаратных, так и временных затрат. Очевидно, при возврате к прерванной программе проще начать выполнение текущей команды заново. В этом случае сохранять при прерывании достаточно лишь значение PC. Поэтому в большинстве случаев процессоры анализируют состояние линий запросов *в конце каждого командного цикла*.

Идентификация источника прерывания

Различают два типа входов запросов на прерывания — *радиальные* и *векторные*. Получив запрос на прерывание, процессор должен идентифицировать

его источник, т. е. в конечном счете определить начальный адрес обслуживающей это прерывание программы. Способ идентификации зависит от типа входа, на который поступил запрос.

Каждый радиальный вход связан с определенным адресом памяти, по которому размещается указатель на обслуживающую программу или сама программа. Если радиальный вход связан с несколькими источниками запросов, то необходимо осуществить программную идентификацию путем последовательного (в порядке убывания приоритетов) опроса всех связанных с этим входом источников прерывания. Этот способ не требует дополнительных аппаратных затрат и одновременно решает проблему приоритета запросов, однако время реакции системы на запрос может оказаться недопустимо большим, особенно при большом числе источников прерываний.

Гораздо чаще в современных МПС используется т. н. *векторная подсистема прерываний*. В такой системе микропроцессор, получив запрос на векторном входе INT, выдает на свою выходную линию сигнал подтверждения прерывания INTA, поступающий на все возможные источники прерывания. Источник, не выставивший запроса, никак не реагирует на сигнал INTA. Источник, выставивший запрос, получая сигнал INTA, выдает на системную шину данных "*вектор прерывания*" — *свой номер или адрес обслуживающей программы* или, чаще, адрес памяти, по которому расположен указатель на обслуживающую программу. Время реакции МПС на запрос векторного прерывания минимально (1—3 машинных цикла) и не зависит от числа источников.

Приоритет запросов

Для исключения конфликтов при одновременном возникновении нескольких запросов на векторном входе ответный сигнал INTA подается на источники запросов не параллельно, а последовательно — в порядке убывания приоритетов запросов. Источник, не выставлявший запроса, транслирует сигнал INTA со своего входа на выход, а источник, выставивший запрос, блокирует дальнейшее распространение сигнала INTA. Таким образом, только один источник, выставивший запрос, получит от процессора сигнал INTA и выдаст по нему свой вектор на шину данных.

Более гибко решается проблема организации приоритетов запросов при использовании в МПС специальных *контроллеров прерываний*.

Конфликты на радиальном входе исключаются самим порядком программного опроса источников.

Приоритет программ

Прерывание в общем случае может возникать не только при решении "фоновой" задачи, но и в момент работы другой прерывающей программы, причем

не всякую прерывающую программу допустимо прерывать любым запросом. В фоновой задаче также могут встречаться участки, при работе которых прерывания (все или некоторые) недопустимы. В общем случае в каждый момент времени работы процессора должно быть выделено подмножество запросов, которым разрешено прерывать текущую программу.

В МПС эта задача решается на нескольких уровнях. В процессоре обычно предусматривается программно-доступный флаг разрешения/запрещения прерывания, значение которого определяет возможность или невозможность всех прерываний. Для создания более гибкой системы приоритетов программ на каждом источнике прерываний может быть предусмотрен специальный программно-доступный триггер разрешения формирования запроса. В таком случае возможно формирование произвольного подмножества разрешенных в данный момент источников прерываний.

При использовании контроллера внешних прерываний, в нем обычно предусматривают специальный программно-доступный регистр, разряды которого *маскируют* соответствующие линии запросов на прерывание, запрещая контроллеру вырабатывать сигнал прерывания процессору, если запросы от ВУ поступают по замаскированным линиям. Однако замаскированные запросы сохраняются в контроллере и в дальнейшем, при изменении состояния регистра маски, могут быть переданы на обслуживание.

Обработка прерывания

К обработке прерывания отнесем фиксацию состояния прерываемой программы, переход к программе, соответствующей обслуживаемому прерыванию, и возврат к прерванной программе после окончания работы прерывающей программы.

Выше мы определили, что большинство процессоров может прервать выполнение текущей программы и переключиться на реализацию обработчика прерывания только после завершения очередной команды. При этом в качестве контекста прерванной программы необходимо сохранить текущее состояние счетчика команд PC, а в PC загрузить новое значение — адрес программы-обработчика прерывания. Очевидно, адрес возврата в прерванную программу (содержимое PC на момент прерывания) следует размещать в стеке, что позволит при необходимости осуществлять вложенные прерывания (когда в процессе обслуживания одного прерывания получен запрос на обслуживание другого).

Можно вспомнить, что подобный механизм реализован в системах команд многих процессоров для выполнения команд вызовов подпрограммы (CALL, JSR). В этих командах адрес вызываемой подпрограммы содержится в коде команды.

В случае вызова обработчика прерывания его адрес необходимо связать либо со входом, на который поступил запрос (радиальные прерывания), либо с номером источника прерываний, сформировавшего запрос (векторные прерывания). В первом случае не требуется никаких внешних процедур для идентификации источника, сразу можно запускать связанный со входом обработчик. Понятно, здесь идет речь об отсутствии необходимости в аппаратных процедурах идентификации источника запроса. Если на радиальный вход "работают" несколько источников, то выбор осуществляется программными способами.

В случае векторных прерываний адрес перехода связывают с информацией, поступающей от источника запроса по шине данных в машинном цикле обслуживания прерывания — *вектором прерывания*.

Напомним, что любой командный цикл процессора начинается с чтения команды из памяти. В первом машинном цикле командного цикла процессор выдает на шину адреса содержимое PC, формирует управляющий сигнал RDM и помещенное памятью на шину данных слово интерпретирует как команду (или ее начальную часть, если длина команды превышает длину машинного слова).

Если в конце очередного командного цикла процессор обнаруживает (незамаскированный) запрос на векторном входе, он начинает следующий командный цикл с небольшими изменениями: содержимое PC по-прежнему выдается на шину адреса (чтобы не нарушать общности цикла), но вместо сигнала RDM формирует сигнал INTA. Источник запроса (чаще — контроллер прерываний) в ответ на сигнал INTA формирует на шину данных код команды вызова подпрограммы, в адресной части которой размещается адрес обработчика соответствующего прерывания.

Такой простой способ реализации векторных прерываний, с использованием уже существующего механизма вызова подпрограмм, был реализован, например, в микропроцессоре i8080 с контроллером прерываний i8259. Однако этот механизм, как, впрочем, и все остальное, допускает дальнейшее совершенствование.

Прежде всего, желание иметь возможность располагать подпрограммы в произвольной области памяти приводит к необходимости размещать в поле адреса команды вызова полноразрядный адрес (16 — 20 — 32 бита). В этом случае длина команды превышает длину машинного слова и ее ввод требует нескольких машинных циклов (например, в i8080 — трех), что увеличивает время реакции системы на запрос прерывания.

Для преодоления этого недостатка в систему команд процессора включают дополнительно "укороченные" команды вызова длиной в одно машинное слово. Эти команды в процессорах 8080 и *x*86 имеют мнемокод INT. В микро-

процессоре i8080 имеется 8 таких команд длиной в 1 байт, адресующих подпрограммы по фиксированным адресам памяти: 0000h, 0008h, 0010h, ..., 0038h.

В процессорах *x*86 имеется 256 вариантов двухбайтовых команд `INT 00h`, ..., `INT FFh`, байт поля адреса которых (называемый *вектором*) после умножения на 4 указывает на четырехбайтовую структуру, определяющую произвольный адрес в адресном пространстве памяти.

Напомним, что доступ в память процессоров *x*86 (в *реальном режиме*) осуществляется только в рамках сегментов размером в 64 Кбайт. Положение начала сегмента в адресном пространстве памяти определяется содержимым 16-разрядного сегментного регистра, а положение адресуемого байта внутри сегмента — 16-разрядным смещением. Среди команд передачи управления различают *короткие* и *длинные переходы* (вызовы). При коротком вызове подпрограмма должна располагаться в текущем сегменте кода, и ее вызов сопровождается только изменением счетчика команд (в *x*86 он обозначается, как IP). При длинном вызове новое значение загружается как в IP, так и в сегментный регистр кода CS. Таким образом, для осуществления длинного вызова (перехода) в адресном поле команды необходимо разместить 4 байта.

Механизм векторных прерываний в процессорах *x*86 в реальном режиме реализован следующим образом. В начальных адресах 00000h, ..., 003FFh пространства памяти размещается таблица векторов прерываний объемом 1 Кбайт, включающая 256 строк таблицы — четырехбайтовых структур CS:IP, которые определяют адреса соответствующих обработчиков прерываний. В цикле обработки векторного прерывания (запрос по входу INT), процессор получает от источника байт — номер строки таблицы векторов прерываний, из которой и загружаются новые значения CS и IP. Старые значения CS:IP (адрес возврата) размещаются в стеке.

Запросу по радиальному входу NMI соответствует вектор 2, поэтому появление активного значения не вызывает машинного цикла обслуживания прерывания, а сразу вызывается обработчик по адресу из ячеек памяти 00008h, ..., 0000Bh. Кстати, любой обработчик прерывания (независимо от значения маскирующих флагов) можно вызвать программно с помощью команды `INT` *nn*, где *nn* — номер строки таблицы векторов прерываний.

Таким образом, команда `INT` отличается от команды `CALL`, во-первых, способом адресации вызываемой подпрограммы (прямой адрес — в команде `CALL`, косвенный — в `INT`), во-вторых, при реализации `INT` в стек, помимо CS и IP, помещается содержимое регистра признаков процессора — FLAGS. Соответственно, завершаться подпрограмма, вызываемая командой `INT`, должна командой `IRET` ("возврат из прерывания"). Действие `IRET` отличается от действия `RET` извлечением из стека дополнительного слова в регистр FLAGS.

6.3.4. Прямой доступ в память

В процессе работы МПС с интерфейсом типа "общая шина" часто возникает необходимость передачи достаточно больших массивов данных между памятью и ВУ (например, копирование сектора диска, загрузка видеопамяти и т. п.). При наличии в системе единственного активного устройства — процессора возможен единственный путь решения этой задачи — программно-управляемый обмен "Память → Процессор → ВУ" (или "ВУ → Процессор → Память").

Рассмотрим вариант программно-управляемого обмена между памятью и внешним устройством в МПС на базе МП i8080 [13]. Пусть необходимо передать массив данных длиной L, начиная с адреса ADR на ВУ с адресом AIO. Положим, что начальный адрес массива загружен в регистровую пару HL, а длина массива — в регистр C. Тогда фрагмент программы обмена может иметь вид, представленный в табл. 6.2.

Таблица 6.2. Фрагмент программы обмена

Мнемокод	Комментарий	Количество МЦ
LM: MOV A,M	Чтение байта в Акк	2
OUT AIO	Выдача байта на ВУ	3
INX H	Модификация адреса	1
DCRC	Модификация счетчика	1
JNZ LM	Переход, если массив не исчерпан	3
	Всего машинных циклов:	10

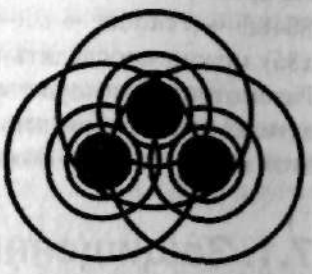
Таким образом, для того чтобы в рамках процедуры копирования массива данных переслать из памяти в ВУ один байт данных, потребуется десять машинных циклов. Процессоры с более совершенной системой команд (например, *x*86) могут использовать для этой цели меньшее число МЦ, но все равно их будет более одного.

Управляя обменом, микропроцессор "ведет" два счетчика — адресов массива и количества переданных байтов и формирует на магистраль сигналы управления. Если снабдить ВУ аппаратными счетчиками и схемой формирования управляющих сигналов (т. н. "канал прямого доступа в память" — ПДП), то передачу одного байта (слова) можно осуществить за один МЦ без участия процессора. Необходимо лишь на время передачи данных под управлением канала ПДП блокировать работу процессора, отключив его от системной шины. Для этого служит вход захвата шины HLD. Если подать на него активный

уровень, то МП по окончании текущего МЦ, безусловно, перейдет в режим ожидания, переведя все свои выходные линии, кроме HLDA, в высокоимпедансное состояние, а выход HLDA — в состояние логической 1. Выходной сигнал HLDA используется для отключения процессорного модуля от системной шины — перевода шинных формирователей, включенных между локальной и системной шиной, в высокоимпедансное состояние.

Если в МПС используется несколько ВУ, снабженных каналом ПДП, то целесообразно использовать специальный контроллер ПДП, который обеспечивает программирование каналов ПДП, подключение их к системной шине и дисциплину обслуживания.

ГЛАВА 7



Эволюция архитектур микропроцессоров и микроЭВМ

В *главе 6* была рассмотрена архитектура 16-разрядного микропроцессора i8086 и систем на его основе. Эту архитектуру мы (условно) будем считать базовой. Уже в ней по сравнению с первыми 8-разрядными системами (на базе i8080) реализован ряд новых архитектурных решений:

- расширена система команд (по набору операций и способам адресации);
- архитектура микропроцессора ориентирована на мультипроцессорную работу. Разработана группа вспомогательных БИС (контроллеров и специализированных процессоров) для организации мультимикропроцессорных систем различной конфигурации;
- начато движение в сторону совмещения во времени выполнения различных операций. Микропроцессор включает два параллельно работающих устройства: *обработки данных и связи с магистралью, что позволяет* совместить во времени процессы обработки информации и передачи ее по магистрали;
- введена новая (по сравнению с i8080) организация памяти, которая далее использовалась во всех старших моделях семейства Intel — *сегментация памяти.*

Можно сказать, что основная цель совершенствования микропроцессоров — это увеличение их производительности. Достигается эта цель различными путями: повышением тактовой частоты работы кристалла, совершенствованием операционных устройств (например, применение параллельного умножителя), организацией параллельной во времени работы нескольких устройств, совершенствованием системы команд (с ориентацией под конкретный класс задач), эффективной организацией иерархии памяти, опережающим выполнением ряда процедур командных циклов, организацией мультизадачных и мультипроцессорных систем и другими способами.

На примере микропроцессоров Intel линии 8080 → 8086 → 80286 → 80386 → 80486 → Pentium → ... → Pentium 4 (это семейство принято обозначать как *x*86) можно проследить реализацию многих из перечисленных выше путей. Рассмотрим некоторые из них, не придерживаясь хронологической последовательности нововведений. Более подробные сведения о рассматриваемых в этой главе вопросах можно найти в [3, 11, 12, 14].

7.1. Защищенный режим и организация памяти

Первый шаг для увеличения производительности систем на базе процессоров *x*86 был сделан в направлении мультипроцессорной конфигурации. В 8086 предусмотрены два режима работы — *минимальный* и *максимальный (см. разд. 6.1.1)*, причем последний ориентирован на организацию мультипроцессорных систем. Часть выводов микропроцессора в максимальном режиме вместо сигналов управления шиной передает коды внутренних состояний управляющего автомата; кроме того, в составе серии выпускались специализированные модули, которые обеспечивали доступ микропроцессора к системной шине — арбитраж шины.

Однако широкого распространения подобная архитектура не получила, поскольку при отсутствии на кристалле микропроцессора достаточно "вместительной" внутренней памяти процессоры постоянно ожидают в очереди на доступ к шине. Характерно, что в последующих моделях семейства — 80286, 80386, 80486 поддерживалась только однопроцессорная конфигурация, и лишь в Pentium вновь вернулись к возможности организации многопроцессорных систем.

7.1.1. Сегментная организация памяти

Как вы, очевидно, помните, в микропроцессоре 8086 в рамках адресного пространства объемом 1 Мбайт одновременно было доступно четыре сегмента по 64 Кбайт каждый.

В следующих моделях микропроцессоров семейства *x*86[1] в рамках т. н. *защищенного режима* (protect mode, P-режим) организовано *линейное адресное пространство* объемом 2^{32} байтов, в котором допускается создание практически любого числа сегментов.

[1] В рамках этой главы будем понимать под обозначением *x*86 микропроцессоры i80386 и старше.

Если в 8086 единственным атрибутом сегмента был его начальный адрес, то в P-режиме старших моделей семейства *x86* для описания многочисленных атрибутов предусмотрена специальная структура — дескриптор.

Дескриптор — это 8-байтовый блок, содержащий атрибуты области линейных адресов — *сегмента*. Дескриптор включает в себя информацию о положении сегмента в линейном адресном пространстве, размере сегмента, типе информации, хранящемся в сегменте и правах доступа к ней, а также другие атрибуты сегмента. Формат дескриптора представлен на рис. 7.1.

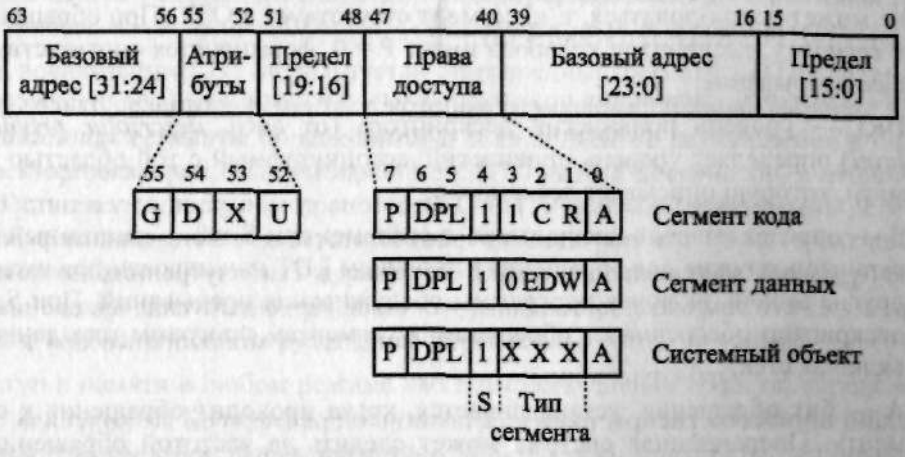


Рис. 7.1. Формат дескриптора

Назначение полей дескриптора:

- *базовый адрес*[1] [31:0] определяет место сегмента (начальный адрес) внутри 4-гигабайтного адресного пространства;
- *предел* [19:0] определяет размер сегмента с учетом бита гранулярности (см. далее).

Поле *атрибутов* включает следующие признаки:

- G — бит гранулярности. При значении G = 0 размер сегмента задается в байтах, а при G = 1 — в страницах по 4 Кбайт. В первом случае максимальный размер сегмента может достигать 1 Мбайт, во втором — 4 Гбайт;
- D — бит размера по умолчанию (от англ. *defaults size*) обеспечивает совместимость с процессором 80286. При *D = 0* находящиеся в сегменте операнды считаются имеющими размер 16 битов, иначе — 32 бита;

[1] Из рис. 7.1 видно, что дескриптор содержит ряд полей, имеющих название и размер. Например, 32-разрядный адрес "базовый адрес [31:0]"; но в формате данного дескриптора он расположен в разрядах [63:56] и [39:16]. То же относится и к полю "предел". Это связано с желанием фирмы Intel сохранить преемственность форматов младших моделей.

- X — зарезервирован Intel и не должен использоваться программистом (содержит 0);
- U — *бит пользователя* (*от англ. user*) *предназначен для использования* системным программистом. Процессор игнорирует этот бит.

Байт *права доступа* (AR) имеет несколько отличающуюся структуру для дескрипторов сегментов разных типов, но некоторые поля этого байта являются общими для всех дескрипторов:

- P — бит присутствия (от англ. *present*) сегмента, если P = 0, то дескриптор не может использоваться, т. к. сегмент отсутствует в ОЗУ. При обращении к сегменту, дескриптор которого имеет P = 0, формируется соответствующее прерывание;
- DRL — уровень привилегий дескриптора (от англ. *descriptor privilege level*) определяет уровень привилегий, ассоциируемый с той областью памяти, которую описывает дескриптор;
- S — определяет роль дескриптора в системе: при S = 0 — системный дескриптор, служит для обращения к таблицам LDT или шлюзам для входа в другие задачи, включая программы обслуживания прерываний. При S = 1 дескриптор обеспечивает обращение к сегментам программ или данных, включая стек;
- A — бит обращения, устанавливается, когда проходит обращение к сегменту. Операционная система может следить за частотой обращения к сегменту путем периодического анализа и очистки A.

Трехбитное поле *тип сегмента* определяет целевое использование сегмента, задавая допустимые в сегменте операции. Значение этого поля для системных дескрипторов (S = 0) безразлично. Для несистемных сегментов биты поля *тип сегмента* имеют следующие значения:

- бит 3 различает сегменты кода (1) и данных (0);
- для сегмента кода бит 2 (Conforming) отмечает при C = 1 т. н. "подчиненные сегменты" (см. далее), а бит 1 (Read) при R = 1 допускает чтение кода как данных с помощью префикса замены сегмента;
- для сегмента данных бит 2 (Expand Down) определяет т. н. "расширение вниз" — для сегментов стека ED = 1, а для сегментов собственно данных ED = 0;
- бит 1 (Write) показывает возможность записи в сегмент при W = 1.

Дескрипторы хранятся в памяти и группируются в дескрипторные таблицы:

- GDT — глобальная дескрипторная таблица;
- IDT — дескрипторная таблица прерываний;
- LDT — локальная дескрипторная таблица.

Причем, если GDT и IDT — общесистемные, присутствуют в системе в единственном экземпляре и являются общими для всех задач, то LDT может создаваться для каждой задачи.

Максимальный размер дескрипторной таблицы может составлять $2^{13} = 8192$ дескриптора ($2^{13} \times 8 = 65\,536$ байтов).

Дескрипторная таблица локализуется в памяти с помощью соответствующего регистра. 48-битовые регистры GDTR и IDTR содержат 32-битовое поле базового адреса таблицы и 16-битный предел (размер) таблицы с байтовой гранулярностью.

Для локализации LDT используется 16-разрядный регистр LDTR, содержащий только селектор сегмента, в котором размещена таблица. Таблицы LDT хранятся как сегменты, а дескрипторы этих сегментов размещаются в GDT. Селектор регистра LDTR выбирает из GDT нужный дескриптор, и атрибуты LDT становятся доступны процессору. С LDTR, как и с сегментными регистрами, ассоциируется соответствующий "теневой регистр", в который помещается выбранный из GDT дескриптор LDT текущей задачи. При переключении задачи достаточно заменить 16-разрядное содержимое LDTR, а процессор автоматически загрузит теневой регистр.

Доступ к памяти в любом режиме *x*86 возможен лишь в область, определенную как сегмент. Количество доступных в данный момент сегментов определяется числом сегментных регистров (CS, SS, DS, ES, FS, GS). Однако в защищенном режиме содержимое сегментного регистра не является базой сегмента, а рассматривается как селектор сегмента и имеет формат, приведенный на рис. 7.2.

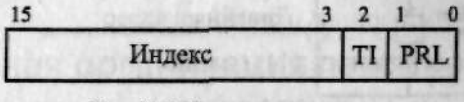


Рис. 7.2. Формат селектора

Индекс определяет смещение внутри дескрипторной таблицы, которая соответственно разрядности индекса может содержать 2^{13} 8-байтовых дескрипторов. Бит TI определяет *тип дескрипторной таблицы*: 0 — глобальная, 1 — локальная. Поле RPL определяет *запрашиваемый уровень привилегий*.

Итак, селектор адресует дескриптор сегмента в одной из дескрипторных таблиц. Всякий раз, когда производится перезагрузка сегментного регистра (замена селектора), адресуемый им дескриптор извлекается из соответствующей дескрипторной таблицы и помещается в "теневой регистр" дескриптора. Все последующие обращения к этому сегменту не требуют чтения из дескрипторной таблицы.

Логический адрес в защищенном режиме, как и в реальном, описывается парой RS:EA, где RS — содержимое выбранного сегментного регистра, EA — эффективный адрес, генерируемый программой (смещение в сегменте).

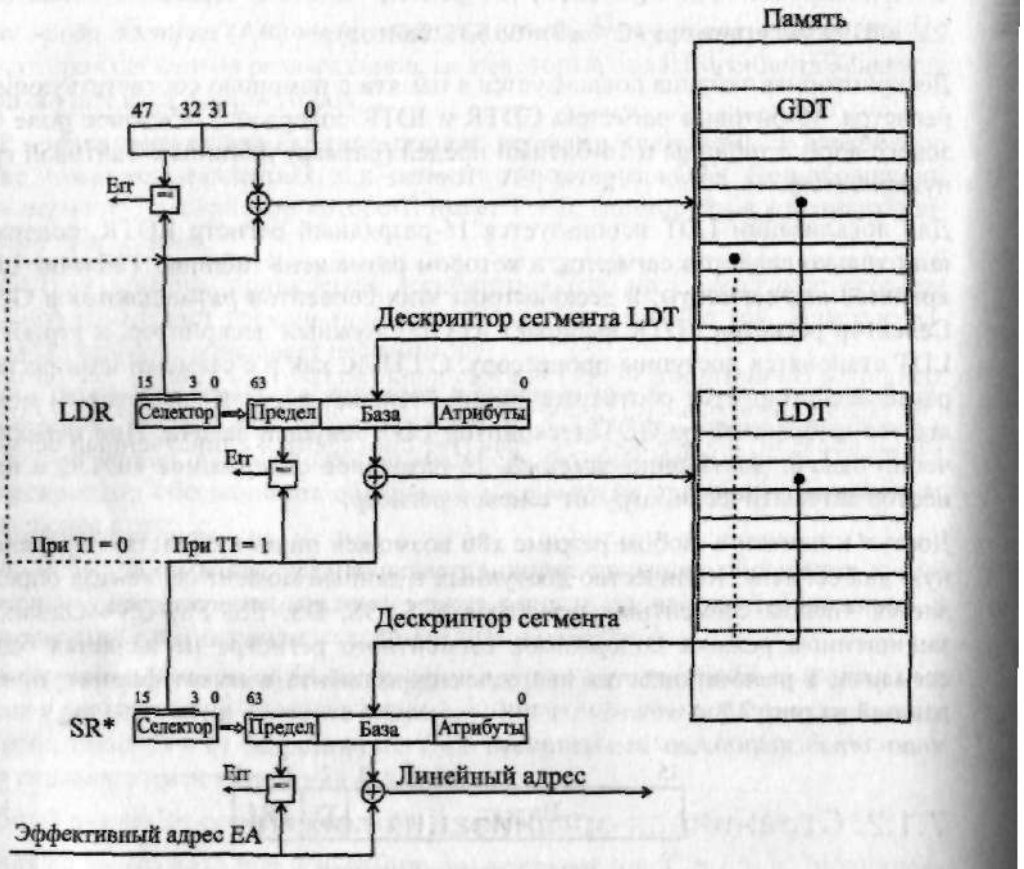


Рис. 7.3. Преобразование логического адреса в линейный

Процесс загрузки дескрипторных регистров и преобразования эффективного (логического) адреса в линейный протекает следующим образом (рис. 7.3):

1. При переходе в защищенный режим в памяти создается глобальная дескрипторная таблица, базовый адрес которой размещается в регистре GDTR.
2. Несколько сегментов определяется в памяти, и их дескрипторы помещаются в GDT.
3. При запуске очередной задачи можно определить дополнительно несколько сегментов и для хранения их дескрипторов создать локальную дескрипторную таблицу, как системный сегмент, дескриптор которого хранится в

GDT, а его положение в GDT определяется селектором в регистре LDTR. В теневой регистр LDTR автоматически помещается дескриптор сегмента LDT.

4. При загрузке в любой сегментный регистр нового содержимого в соответствующий теневой регистр автоматически помещается новый дескриптор из GDTR или LDTR.

5. При генерации программой очередного адреса EA из соответствующего теневого сегментного регистра выбирается базовый адрес сегмента и складывается со значением EA. Полученная сумма представляет собой линейный адрес.

В приведенной выше процедуре не отражены особые случаи, которые могут возникать при различных нарушениях (ошибках) в процессе формирования линейного адреса.

Механизм сегментации можно искусственно подавить, назначив все базовые адреса сегментов равными нулю и определив длину всех сегментов в 4 Гбайт. Таким образом, в адресном пространстве определится единственный сегмент размером 2^{32} байтов.

Сегмент в защищенном режиме — область памяти, снабженная рядом атрибутов: типом, размером, положением в памяти, уровнем привилегий и др. Сегмент может начинаться и кончаться, где угодно, и его размер — произвольный. Другой элемент памяти — страница — имеет строго фиксированный размер (4 Кбайт) и положение в линейном адресном пространстве: страница всегда выровнена по границе 4-килобайтовых фрагментов, т. е. 12 *младших разрядов адреса страницы — всегда нули.*

7.1.2. Страничная организация памяти

Наряду с сегментной организацией в микропроцессорах *x*86 возможна дополнительно *страничная организация памяти.* Механизм страничной организации памяти может включаться (выключаться) программно путем установки (сброса) флага PG регистра CR0.

Все линейное адресное пространство делится на разделы, число которых может достигать 1024. Каждый раздел, в свою очередь, может содержать до 1024 страниц (рис. 7.4), размер которых фиксирован — 4 Кбайт, причем начальные адреса страниц жестко фиксированы в физическом адресном пространстве: границы страниц совпадают с границами 4-килобайтовых блоков.

32-разрядный логический адрес, полученный на предыдущем этапе преобразования адреса, рассматривается состоящим из трех полей:

- [31:22] — номер раздела (TABLE);
- [21:12] — номер страницы в разделе (PAGE);
- [11:0] — номер слова на странице (смещение).

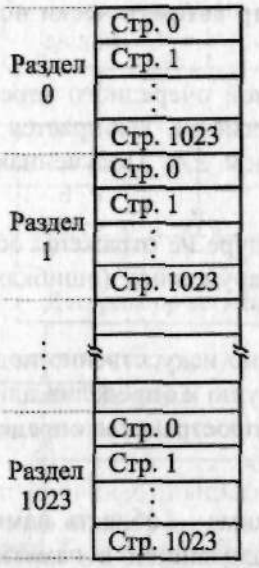


Рис. 7.4. Разделы в линейном адресном пространстве

Начальные адреса страниц данного раздела (вместе с атрибутами страницы) хранятся в памяти в страничной таблице, размер которой 1024 стр. × 4 байта = 4096 байтов.

Поскольку в задаче может быть несколько разделов и, следовательно, столько же страничных таблиц, то начальные адреса всех страничных таблиц одного сегмента хранятся в специальной таблице — *каталоге раздела*.

Линейный 32-разрядный адрес является исходной информацией для формирования 32-разрядного физического адреса (рис. 7.5) с помощью каталога раздела и страничной таблицы (СТ). Старшие 10 разрядов линейного адреса определяют номер строки каталога разделов, который локализуется содержимым системного регистра CR3.

Поскольку каталог разделов имеет размер 1 Кбайт × 4 байта, он занимает точно одну страницу (CR3[11:0] = 0) и содержит 4-байтовые поля, формат которых показан на рис. 7.6. Помимо базового адреса страничной таблицы, это поле хранит атрибуты страницы. Извлеченный из каталога базовый адрес страничной таблицы складывается (конкатенируется) с разрядами [21:12] линейного адреса для получения адреса строки страничной таблицы, из кото-

рой, в свою очередь, извлекается базовый адрес страницы. Конкатенацией базового адреса страницы с разрядами [11:0] линейного адреса получается *физический адрес*.

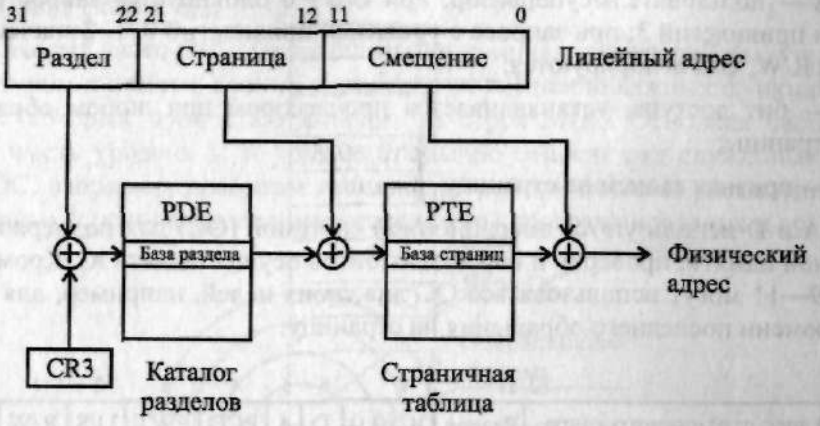


Рис. 7.5. Преобразование линейного адреса в физический

Такая двухуровневая организация страничной таблицы позволяет значительно экономить память для хранения страничных таблиц. Действительно, если рассматривать разряды [31:20] линейного адреса как номер строки страничной таблицы, то ее (таблицы) размер должен составлять $2^{20} \times 4$ байтов, т. е. 4 Мбайт. Абсолютное большинство задач никогда не использует такого количества страниц, однако, во избежание возникновения особого случая (внутреннего прерывания) необходимо поддерживать всю такую таблицу целиком.

При двухуровневой организации страничного преобразования (см. рис. 7.5) в памяти достаточно хранить каталог разделов и страничные таблицы только реально существующих разделов. Максимальное число разделов может достигать 1024, однако во многих случаях достаточно бывает двух-трех разделов, а то и единственного.

Каждая четырехбайтовая строка каталога разделов и страничной таблицы содержит, помимо 20-разрядного базового адреса, атрибуты страницы, определяющие ее назначение, положение в физической памяти, а также информацию, позволяющую аппаратно поддерживать некоторые алгоритмы замещения страниц при страничных сбоях. Формат строки этих таблиц представлен на рис. 7.6.

Атрибуты страницы (CT) :

- P — бит присутствия, при P = 0 страница отсутствует в оперативной памяти, попытка обращения к ней вызывает прерывание 14 — "страничный сбой";

- R/W — чтение/запись, если работает программа с уровнем привилегий 3 (низший), то при R/W = 0 разрешается только чтение, но не запись на страницу;
- U/S — пользователь/супервизор, при U/S = 0 блокируется запрос с уровнем привилегий 3; при запросе с уровнями привилегий 0, 1, 2 значения битов R/W, U/S игнорируются;
- A — бит доступа, устанавливается процессором при любом обращении к странице;
- D — признак записи на страницу.

Биты A и D используются операционной системой (ОС) для поддержки виртуальной памяти, проверку и сброс этих битов осуществляет ОС. Кроме того, биты 9—11 могут использоваться ОС для своих целей, например, для хранения времени последнего обращения на страницу.

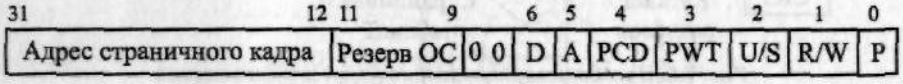


Рис. 7.6. Формат строки каталога разделов и страничных таблиц

В *x86* предусмотрена *ассоциативная память страничных таблиц*, которая называется *буфером ассоциативной трансляции* — TLB.

TLB представляет собой 32 ячейки АЗУ 1-го рода, поле признаков которого (теги) включают старшие 20 разрядов линейного адреса. Информационное поле ячейки включает 20 старших битов физического адреса страницы и ряд ее атрибутов. Биты D, U/S, R/W имеют тот же смысл, что в слове CT, а бит достоверности V сбрасывается при записи в CR3 нового слова (смена каталога). После преобразования очередного линейного адреса в физический бит V в этой ячейке устанавливается.

Наличие TLB позволяет при кэш-попадании избежать обращения к ОЗУ при преобразовании линейного адреса. При кэш-промахе микропроцессор выполняет процедуру формирования физического адреса по каталогу раздела и CT. Полученный из CT 20-разрядный базовый адрес вместе с 20-разрядным тегом заносятся в свободную ячейку TLB или занимают ячейку, в которой хранится адрес, введенный в TLB ранее других.

Так как TLB хранит адреса 32 страниц по 4 Кбайт, то непосредственно доступными становятся физические адреса 128 Кбайт памяти.

7.1.3. Защита памяти

В *x86* предусмотрены два вида защиты памяти: на уровне сегментов и на уровне страниц.

Защита памяти на уровне сегментов

В *x86* определено понятие *привилегии для сегмента* и установлены 4 уровня привилегий PL (рис. 7.7), которые задаются номерами от 0 (наиболее защищенный) до 3 (низший).

В ядро входит часть ОС, обеспечивающая инициализацию работы, управление доступом к памяти, защиту и ряд других жизненно важных функций, нарушение которых полностью выводит из строя МПС. Основная часть ОС должна иметь уровень 1. К уровню 2 обычно относят ряд служебных программ ОС, например, драйверы внешних устройств, системы управления базами данных, специализированные подсистемы программирования и др.

Рис. 7.7. Кольца защиты сегментов

Выше отмечалось, что основой организации памяти *x86* является сегмент. С каждым сегментом (данных, кода или стека) ассоциируется уровень привилегий DPL и все, что находится внутри этого сегмента, имеет данный уровень привилегий. DPL располагается в байте доступа дескриптора сегмента, поэтому его называют *уровнем привилегий дескриптора* (Descriptor Privilege Level), однако правильнее считать его уровнем привилегий сегмента.

Уровень привилегий выполняющегося кода называется *текущим уровнем привилегий* CPL (Current Privilege Level или Code Privilege Level) и он задается полем RPL селектора в сегментном регистре CS. Значение CPL можно считать уровнем привилегий процессора в текущий момент времени, т. к. при передаче управления сегменту кода с другим уровнем привилегий процессор будет работать на новом уровне привилегий.

Каждый селектор выбирает точно один дескриптор и, соответственно, один сегмент, но конкретный сегмент могут идентифицировать несколько селекторов ("альтернативное именование"). Младшие два бита селектора содержат поле *запрашиваемого уровня привилегий* RPL (Requested Privilege Level). Это поле не влияет на выбор дескриптора, но учитывается при контроле привилегий.

Таким образом, текущее состояние системы защиты характеризуется следующими признаками:

- CPL — уровень привилегий выполняемого кода, размещается в поле RPL сегментного регистра кода CS;
- DPL — уровни привилегий для каждого из восьми открытых сегментов, располагаются в байте доступа дескрипторов, помещенных в "теневые регистры";
- RPL — определяют уровни привилегий источника селектора, размещаются в полях RPL сегментных регистров.

Процессор постоянно контролирует, обладает ли текущая программа достаточным уровнем привилегий, чтобы:

- выполнять некоторые команды;
- обращаться к данным других программ;
- передавать управление внешнему (по отношению к программе) коду командами передачи управления типа FAR.

Рис. 7.8. Правила доступа к сегментам

В системе команд существуют специальные *привилегированные команды*, которые могут выполняться процессором, работающим только на уровне привилегий 0. При попытке выполнить их на другом уровне привилегий генерируется прерывание 13 — нарушение общей защиты.

К привилегированным относятся команды:

- останов процессора;
- сброс флага переключенной задачи;

- загрузка регистров дескрипторных таблиц;
- загрузка регистра задачи;
- загрузка слова состояния машины;
- модификация флага прерываний IF*;
- команды ввода/вывода*.

Последние две группы команд (отмеченные *) не обязательно выполняются на нулевом уровне, достаточно, чтобы уровень привилегий программы был выше уровня привилегий ввода/вывода, определяемого полем IOPL в регистре EFLAGS.

Защита доступа к данным

Данные из сегмента могут выбираться только программой, имеющей такой же или более высокий, чем сегмент, уровень привилегий. Программам не разрешается обращение к данным, которые имеют более высокий уровень привилегий, чем выполняемая программа. Программы могут использовать данные на своем и более низких уровнях привилегий. Ограничения на возможность доступа к данным иллюстрирует рис. 7.8.

Контроль реализуется двумя способами. Во-первых, проверка привилегий осуществляется при загрузке селектора в один из сегментных регистров данных — DS, ES, FS или GS. Если значение DPL того сегмента, который выбирает селектор, численно меньше CPL, процессор не загружает селектор и формирует *нарушение общей защиты*. Во-вторых, после успешной загрузки селектора при использовании его для фактического обращения к памяти процессор контролирует, разрешена ли для этого сегмента запрашиваемая операция (чтение или запись). Кроме того, при обращении контролируется значение запрашиваемого уровня привилегий RPL, причем обращение разрешается, если DPL > max(RPL, CPL), иначе формируется прерывание 13.

Обращение к сегменту стека возможно, если RPL = DPL = CPL, причем сегмент стека должен иметь разрешение на запись — бит W в байте доступа должен быть установлен.

Защита сегментов кода

Межсегментная передача управления происходит по командам `JMP`, `CALL`, `RET`, `INT`, `IRET`. Для передачи управления существуют жесткие ограничения: передавать управление в общем случае можно только в пределах своего уровня привилегий, т. е. DPL целевого дескриптора должен быть точно равен CPL (см. рис. 7.8).

Однако часто бывает необходимо обойти установленные ограничения (например, фрагменты операционной системы могут использоваться програм-

мами пользователя). В x86 предусмотрены два механизма передачи управления между уровнями привилегий: *подчиненные сегменты* и *шлюзы вызова*.

Если сегмент кода определен как подчиненный (установлен в 1 бит подчиненности C в байте доступа дескриптора), то для него вводятся другие правила защиты. С подчиненными сегментами не ассоциируется конкретный уровень привилегий, он устанавливается равным уровню привилегий вызывающей программы. Поэтому код подчиненных сегментов не должен содержать привилегированных команд.

Когда управление передается подчиненному сегменту, биты поля RPL регистра CS не изменяются на значение поля DPL дескриптора нового сегмента кода, а сохраняют прежнее значение. Только в этой единственной ситуации биты поля RPL регистра CS не соответствуют битам поля DPL дескриптора текущего выполняемого сегмента кода.

При использовании подчиненных сегментов сохраняется одно ограничение — значение DPL дескриптора подчиненного сегмента всегда должно быть меньше или равно текущему значению CPL. Другими словами, передача управления подчиненному сегменту разрешается только во внутренние, более защищенные сегменты. Если бы это ограничение нарушалось, то возврат в вызывающую программу был бы вызовом неподчиненного более защищенного сегмента, что никогда не разрешено.

Наличие подчиненных сегментов кода обеспечивает некоторую свободу передачи управления между уровнями привилегий. Для реализации фактического изменения уровня привилегий привлекаются особые системные объекты, называемые *шлюзами вызова* (рис. 7.9).

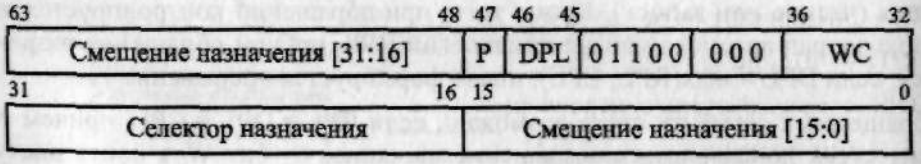


Рис. 7.9. Шлюз вызова

Хотя шлюзы вызова и располагаются в дескрипторной таблице, они, по существу, дескрипторами не являются, т. к. не определяют никакого сегмента. Поэтому в дескрипторе шлюза отсутствует база и граница сегмента, а содержится лишь селектор вызываемого сегмента программы и относительный адрес шлюза — задается фактически адрес *селектор : смещение* точки входа той процедуры (назначения), которой шлюз передает управление. Байт доступа имеет тот же смысл, что и в обычных дескрипторах, а пятибитовое поле WC указывает количество параметров, переносимых из стека текущей программы в стек новой программы.

При этом, если вызываемая программа имеет более высокий уровень привилегий, чем текущая, то для нее по команде CALL создается новый стек, позиция которого определяется из сегмента состояния задачи TSS. В этот стек последовательно записываются: старые значения SS и ESP, параметры, переносимые из старого стека, старые CS и EIP. По команде RET происходит возврат к старому стеку.

Дескриптор шлюза вызова действует как своеобразный интерфейс между сегментами кода на разных уровнях привилегий. Шлюзы вызова идентифицируют *разрешенные точки входа* в более привилегированные программы, которым может быть передано управление.

Селектор, определяющий шлюз вызова, можно загружать только в сегментный регистр CS для передачи управления сегменту кода на другом уровне привилегий.

Защита памяти на уровне страниц

В отличие от 80386, процессоры 80486 и Pentium имеют дополнительные поля в элементе страничной таблицы [3]. Формат строки CT i80486 представлен на рис. 7.10, сравните его с форматом на рис. 7.6.

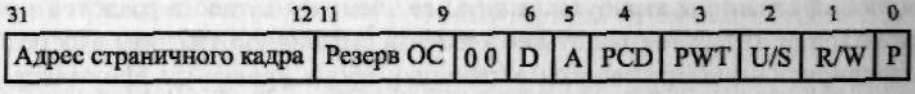


Рис. 7.10. Формат строки CT процессора 80486

Кроме используемых в i80386 полей:

- P — бит присутствия;
- R/W — чтение/запись;
- U/S — пользователь/супервизор;
- A — бит доступа;
- D — признак записи на страницу;
- резерв ОС

введены биты управления кэшированием:

- PCD — запрет кэширования страницы;
- PWT — сквозная запись.

На уровне страниц в 80486 предусмотрены две разновидности контроля:

- ограничение адресуемой области;
- контроль типа.

Ограничение адресуемой области. Для страниц и сегментов привилегии интерпретируются по-разному: для сегментов — 4 уровня, для страниц — только 2, определяемые битом U/S. При U/S = 0 страница имеет уровень супервизора, иначе — уровень пользователя. На уровне супервизора работают обычно операционные системы, драйверы ВУ, а также располагаются защищенные данные (например, страничные таблицы). Уровни привилегий сегментов отображаются на уровень привилегий страниц: если значение CPL равно 0, 1 или 2, то процессор работает на уровне супервизора, при CPL = 3 — на уровне пользователя. На уровне супервизора доступны все страницы, а на уровне пользователя — только страницы уровня пользователя.

Контроль типа. Механизм защиты распознает только два типа страниц: с доступом только по считыванию (R/W = 0) и с доступом по считыванию/записи (R/W = 1), причем в 80386 ограничение по записи действительно только для уровня пользователя. Программа уровня супервизора игнорирует значение бита R/W и может записывать на любые страницы. В отличие от 80386, процессор 80486 разрешает защитить от записи страницы уровня пользователя в режиме супервизора: установка в регистре CR0 бита WR = 1 обеспечивает чувствительность режима супервизора к защищенным от записи страницам режима пользователя.

Для любой страницы атрибуты защиты ее элемента каталога разделов могут отличаться от атрибутов защиты ее элемента страничной таблицы. Процессор контролирует атрибуты защиты в таблицах обоих уровней и принимает решение таким образом, что всякое разночтение в уровне привилегий раздела и страницы всегда разрешается в сторону большей защиты (предоставления меньших прав пользователю).

7.2. Мультизадачность

Под *мультизадачностью* понимают способность процессора выполнять несколько задач "одновременно". Конечно, процессор традиционной архитектуры не может выполнять строго одновременно более одного потока команд, однако он может некоторое время выполнять один поток команд, потом быстро переключиться на выполнение другого потока команд, потом третьего, потом — снова первого и т. д. Такая организация вычислительных процессов при высоком быстродействии процессора создает иллюзию одновременности (параллельности) выполнения нескольких задач.

Для реализации мультизадачности необходимо:

- располагать быстродействующим процессором;
- процессор должен аппаратно поддерживать механизм быстрого переключения задач;

- процессор должен аппаратно поддерживать механизм защиты памяти;
- использовать специальную мультипрограммную операционную систему.

Под *задачей* в мультизадачной системе понимается программа, которая выполняется или ожидает выполнения, пока выполняется другая задача, причем в определение задачи обычно включают ресурсы, требуемые для ее решения (объем памяти, процессорное время, дисковое пространство и др.).

Рассмотрим, как реализуется механизм переключения задач в процессорах *x*86.

7.2.1. Сегмент состояния задачи

Переключение задач в мультизадачной системе предполагает сохранение состояния приостанавливаемой задачи на момент ее останова. Информация о задаче, сохраняемая для последующего восстановления прерванного процесса, называется ее *контекстом*. В системе выделяется область оперативной памяти, доступная только ОС, в которой хранятся контексты задач. Для минимизации времени переключения контекста следует сохранять и восстанавливать минимальную информацию о каждой задаче.

В какой-то степени процесс переключения задачи напоминает вызов процедуры. Отличие состоит в том, что при вызове процедуры информация о точке возврата (автоматически) и содержимое некоторых РОН (программно) помещается в стек, что определяет свойство реентерабельности процедур (возможность вызова самой себя). Задачи не являются реентерабельными, т. к. контексты сохраняются не в стеке, а в фиксированной (для каждой задачи) области памяти в специальной структуре данных, называемой *сегментом состояния задачи* (Task State Segment, TSS), причем каждой задаче соответствует один TSS.

Сегмент TSS определяется дескриптором, который может находиться только в GDT. Формат дескриптора TSS похож на дескриптор сегмента кода и содержит обычные для дескриптора сегмента поля: базового адреса, предела, DPL, биты гранулярности (G = 0) и присутствия P, бит S = 0 — признака системного сегмента. В поле типа бит занятости B показывает, занята задача или нет. Занятая задача выполняется сейчас или ожидает выполнения. Процессор использует бит занятости для обнаружения попытки вызова задачи, выполнение которой прервано. Поле предела должно содержать значение, не меньшее 67h, что на один байт меньше минимального размера TSS. Формат 32-разрядного TSS представлен на рис. 7.11.

Процедура, которая обращается к дескриптору TSS, может вызвать переключения задачи. В большинстве случаев поле DPL дескрипторов сегментов TSS должно содержать 00, поэтому переключение задач могут проводить только привилегированные программы (на нулевом уровне).

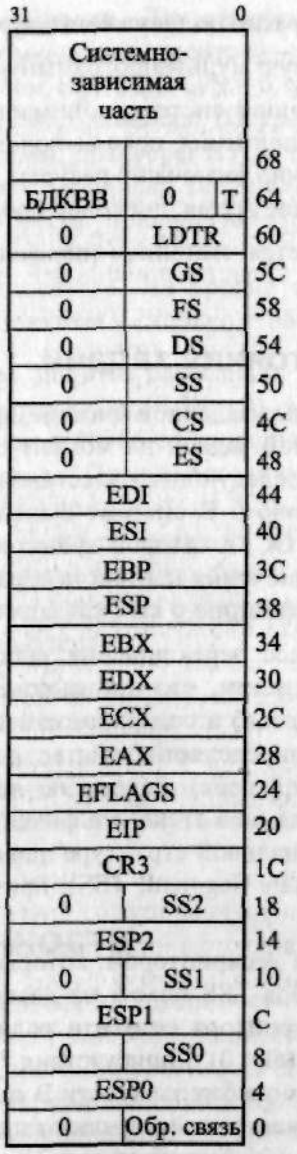


Рис. 7.11. Сегмент TSS

Сегмент TSS не является ни сегментом кода, ни сегментом данных. Доступ к нему имеет только процессор, но не задача, даже на нулевом уровне! Если предполагается программно использовать сегмент TSS, то следует применить *альтернативное именование*.

Обращение к дескриптору TSS не предоставляет возможность процедуре считать или модифицировать сегмент TSS. Загрузка селектора дескриптора

TSS в сегментный регистр вызывает особый случай. Доступ к сегменту TSS возможен только с помощью альтернативного именования, когда сегмент данных отображен на ту же область памяти.

Сегмент состояния задачи TSS (рис. 7.11) включает в себя содержимое всех пользовательских регистров процессора, причем 8 регистров общего назначения хранятся в сегменте в том же порядке, в каком они помещаются в стек командой PUSHAD. Кроме того, в TSS сохраняются значения трех указателей стека $SSi:ESPi$ для трех уровней привилегий — $i \in \{0, 1, 2\}$. Сохранение в TSS регистров CS и EIP позволяет осуществлять рестарт задачи, при этом гарантируется правильное действие команд условных переходов, т. к. в TSS сохраняется и EFLAGS. Сохранение в TSS содержимого регистров CR3 и LDTR позволяет для каждой задачи образовывать свой каталог разделов и локальную дескрипторную таблицу.

В сегменте TSS имеется также несколько дополнительных полей. Поле *обратной связи* содержит селектор TSS той задачи, которая выполнялась перед данной; с его помощью можно организовать цепь вложенных задач. Поле *базы двоичной карты разрешения ввода/вывода* (БДКВВ) содержит 16-битовое смещение в данном сегменте TSS, с которого начинается сама двоичная карта ввода/вывода. Эта карта позволяет определить произвольное подмножество адресов в пространстве ввода/вывода, по которым данной задаче разрешено обращаться независимо от уровня привилегий. Если в этом поле — 00h, то карта отсутствует. *Бит ловушки T* применяется для отладки: когда в TSS T = 1, при переключении на данную задачу генерируется особый случай отладки (прерывание 1).

При переключении задач между ними не передается никакой информации, т. е. они максимально изолированы друг от друга. Этим исключается искажения задач и обеспечивается возможность прекращения и запуска любой задачи в любой момент времени и в любом порядке.

С целью экономии времени на процедуру переключения задач все поля TSS разделяются на "статические" и "динамические". К статическим относятся поля указателей стека трех уровней и содержимое регистра LDTR — они остаются неизменными в течение всего времени существования задачи. Содержимое статических полей TSS определяется ОС при создании задачи. Статические поля процессор только считывает при переключении задачи. Поля регистров и поле обратной связи модифицируются при каждом переключении задачи.

До перехода в мультипрограммный режим необходимо определить дескрипторы TSS, разместить сами сегменты TSS в адресном пространстве и правильно инициировать их. Напомним, что селекторы TSS нельзя загружать в сегментные регистры, поэтому для работы с TSS следует пользоваться аль-

тернативным именованием, т. е. псевдонимами этих сегментов. При загрузке начальных значений полей TSS в CS : EIP указывают точку старта программы (задачи), а в регистр SS — селектор сегмента стека с правильным уровнем привилегий. Если предполагается работа задачи на разных уровнях привилегий, следует инициализировать поля SS*i* : ESP*i*, а если задача рассчитана на использование локальной дескрипторной таблицы и страничного преобразования, в сегменте TSS потребуется инициировать поля LDTR и CR3.

В сегменте TSS отсутствуют поля для регистров CR0 и CR2, следовательно, их значение не изменяется при переключениях задач. Поэтому страничное преобразование и условия работы с устройством "плавающей арифметики" (определяются полями CR0 и CR2) являются глобальными для всех задач. Для каждой задачи может быть свой каталог разделов, но страничное преобразование может быть разрешено или запрещено только для всей системы. Переключение задач не затрагивает регистры GDTR и IDTR, а также регистры отладки и проверки.

Минимальный размер сегмента TSS должен быть 104 байта (68h). Однако пользователь может увеличить размер сегмента TSS для размещения дополнительной информации, например, состояние устройства регистров сопроцессора "плавающей арифметики" FPU, списка открытых файлов, двоичной карты ввода/вывода и др. Однако когда процессор привлекает TSS для переключения задач, он игнорирует все данные сверх аппаратно поддерживаемых 104 байтами, и эту дополнительную информацию из TSS считывают программно.

7.2.2. Переключение задачи

Переключение задачи в *x*86 могут вызвать следующие четыре события:

- старая задача выполняет команду `FAR CALL` или `FAR JMP`, и селектор выбирает *шлюз задачи*;
- старая задача выполняет команду `FAR CALL` или `FAR JMP`, и селектор выбирает *дескриптор TSS*;
- старая задача выполняет команду `IRET` для возврата в предыдущую задачу; эта команда приводит к переключению задачи, если в регистре EFLAGS *бит вложенной задачи* NT = 1;
- возникло аппаратное или программное прерывание и соответствующий элемент дескрипторной таблицы прерываний IDT содержит *шлюз задачи*.

Под термином "старая задача" ("выходящая задача") будем понимать ту задачу, выполнение которой прекращается; под термином "новая задача" ("входящая задача") будем понимать ту задачу, которую начинает выполнять процессор.

Таким образом, селекторами в командах переходов и вызовов могут быть как селекторы TSS (прямое переключение задачи), так и селекторы шлюзов задачи (косвенное переключение задачи). В последнем случае дескриптор шлюза задачи обязательно содержит селектор TSS.

Формат дескриптора шлюза задачи приведен на рис. 7.12.

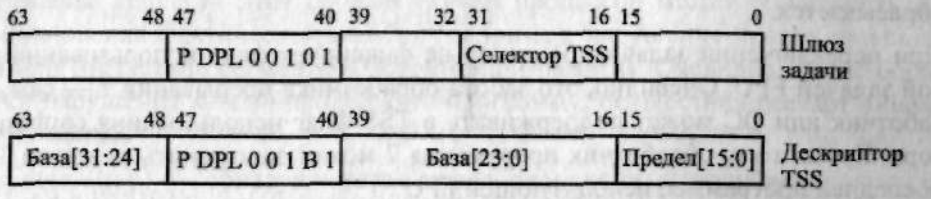


Рис. 7.12. Форматы шлюза задачи и дескриптора TSS

Старая задача должна быть достаточно привилегированна для доступа к шлюзу задачи или к сегменту TSS. Правила привилегий обычные:

- max(CPL, RPL) > DPL шлюза задачи:
- max(CPL, RPL) > DPL сегмента TSS.

Процедура возврата из прерываний IRET всегда возвращает управление прерванной программе. Если флаг NT сброшен в 0, производится обычный возврат, а если он установлен в 1 — происходит переключение задачи. При этом процессор сохраняет свое состояние в сегменте TSS старой задачи, загружает в регистр TR содержимое поля обратной связи — селектор новой задачи ("задачи-предка", т. к. осуществляется возврат) и восстанавливает из сегмента TSS контекст новой задачи. Благодаря наличию в каждом сегменте TSS поля обратной связи можно поддерживать многократные вложения задач. Характерно, что команда возврата из подпрограммы RET не чувствительна к значению флага NT и не может осуществить переключение задачи.

После модификации TR и загрузки нового контекста из сегмента TSS процессор отмечает этот сегмент как занятый (устанавливает бит 41 занятости Busy в его дескрипторе). Занятый TSS может относиться либо к выполняющейся, либо ко вложенной задаче. Переключение на задачу, отмеченную как занятая, не производится! В частности, это исключает возможность реализации реентерабельных задач. Исключение представляет только команда IRET, которая возвращает управление задаче-предку (очевидно, будучи вложенной, она отмечена как запятая).

При переключении задачи процессор устанавливает также флаг переключения задачи TS в регистре CR0. Сброс этого флага может осуществиться только привилегированной командой CLTS. TS применяется для правильного использования некоторых системных ресурсов, в частности — устройства пла-

вающей арифметики (Float Point Unit, FPU). Если при каждом переключении задачи сохранять состояние FPU, то на это уйдет много времени, причем новая задача может вообще не использовать ресурсы FPU, и тогда такое сохранение окажется напрасным. В процессоре 80486 команды FPU анализируют состояние флага TS и если TS = 1, формируется *особый случай 7* и вызывается системная процедура сохранения состояния FPU. После этого флаг TS сбрасывается.

При переключении задачи процессор не фиксирует факт использования новой задачей FPU. Очевидно, это забота обработчика прерывания 7 — сам обработчик или ОС может поддерживать в TSS флаг использования сопроцессора. Кроме того, обработчик прерывания 7 может запоминать селектор TSS последней программы, использующей FPU.

Итак, процесс переключения задачи можно представить следующим образом.

Имеется TR с теневым регистром дескриптора TSS, определяющий TSS старой задачи. Если селектор в командах FAR JMP, FAR CALL, IRET (NT = 1), INT указывает прямо (дескриптор TSS) или косвенно (шлюз задачи) в GDT на системный объект переключения задачи, то производится переключение задачи:

- процессор сохраняет контекст старой задачи в сегменте TSS старой задачи;
- процессор загружает в TR селектор сегмента новой задачи;
- процессор загружает в сегмент TSS новой задачи селектор TSS старой задачи (в поле обратной связи);
- получив доступ к сегменту TSS новой задачи, процессор загружает контекст новой задачи в регистры (в том числе CS : EIP — точка старта);
- процессор устанавливает флаги NT (в регистре EFLAGS) и TS (в CR0 для анализа командами FPU), устанавливает бит занятости задачи в дескрипторе TSS новой задачи.

7.3. Прерывания и особые случаи

Прерывания текущей программы могут возникать по следующим трем причинам:

- внешний сигнал по входам INTR или NMI;
- аномальная ситуация, сложившаяся при выполнении конкретной команды и зафиксированная аппаратурой контроля;
- *находящаяся в программе команда прерывания* INT *n*.

Первая из указанных выше причин относится к аппаратным прерываниям, а две другие — к программным.

Программные прерывания, вызываемые причинами 2 и 3, называют обычно *особыми случаями* (иногда используют термин *исключения*). Особые случаи возникают, например, при нарушении защиты по привилегиям, превышении предела сегмента, делении на ноль и т. д.

Все особые случаи классифицируются как нарушения, ловушки или аварии.

Нарушение (fault) — этот особый случай процессор может обнаружить до возникновения фактической ошибки (например, нарушение правил привилегий или отсутствие сегмента в оперативной памяти). Очевидно, после обработки нарушения можно продолжить программу, осуществив рестарт виновной команды.

Ловушка (trap) — обнаруживается после окончания выполнения виновной команды. После ее обработки процессор возобновляет действия с той команды, которая следует за "захваченной" (например, прерывание при переполнении или команда INT *n*). Большинство отладочных контрольных точек также интерпретируются как ловушки.

Авария (abort) — приводит к потере контекста программы, ее продолжение невозможно. Причину аварии установить нельзя, поэтому осуществить рестарт программы не удается, ее необходимо прекратить. К авариям ("выходам из процесса") относятся аппаратные ошибки, а также несовместимые или недопустимые значения в системных таблицах.

Общая реакция процессора на прерывания или особые случаи состоит в сохранении минимального контекста прерываемой программы (в стеке — адрес возврата и, может быть, некоторую дополнительную информацию), идентификации источника прерывания или особого случая и передаче управления соответствующему обработчику (программе, подпрограмме, задаче). Однако имеются принципиальные различия по формированию сохраняемого контекста.

При возникновении нарушений в стек обработчика особого случая в качестве адреса возврата включается CS : EIP команды, вызвавшей нарушение.

При распознавании ловушки (к ним относятся и большинство внешних прерываний) процессор включает в стек адрес возврата, относящийся к следующей за ловушкой команде.

Наконец, при авариях содержательный адрес возврата отсутствует, поэтому рестарт задачи при авариях невозможен.

Принцип реализации прерываний (внешних) и особых случаев (внутренних) в микропроцессорах фирмы Intel сохранился практически неизменным с МП 8086.

В 8086 сигналы запросов на обработку прерываний формировались либо аппаратурой контроля процессора (внутренние прерывания), либо поступали из

внешней среды на входы процессора INTR или NMI. При обнаружении (разрешенного) запроса в стек помещались текущие значения FLAGS, CS и IP.

В процессе идентификации источника запроса ему ставился в соответствие восьмиразрядный двоичный код *n* — вектор прерывания, причем за каждым внутренним прерыванием и за внешним NMI жестко закреплялся свой вектор, а для запросов, поступивших по входу INTR, реализовывалась процедура ввода вектора с внешней шины. Далее, определенный вектор прерывания рассматривался как номер строки таблицы, располагающейся с нулевого физического адреса памяти. Четырехбайтовыми элементами этой таблицы были адреса CS : IP точек входа в подпрограммы — обработчики прерываний. Таким образом, в системе поддерживалось до 256 различных обработчиков прерываний, причем векторы 0—31 резервировались за внутренними прерываниями (и NMI), а остальные — для внешних прерываний.

При идентификации источника запроса INTR выполняются следующие действия (при условии, что флаг IF = 1, иначе запрос INTR игнорируется):

1. Генерируется два цикла шины для ввода вектора внешнего прерывания.
2. Помещается в стек содержимое регистра FLAGS.
3. Помещается в стек содержимое регистра CS.
4. Помещается в стек содержимое регистра IP.
5. Сбрасывается в 0 флаг разрешения внешних прерываний IF, запрещая восприятие новых запросов по входу INTR до явной установки флага IF в 1 командой `STI`.
6. По значению вектора *n* обращаются к *n*-му элементу таблицы векторов прерываний и из нее загружаются новые значения регистров CS : IP.
7. Начинается выполнения обработчика прерывания с точки входа, определяемой CS : IP.

Сохраненное в стеке старое содержимое регистров CS : IP образует адрес возврата. Когда обработчик прерываний заканчивает свои действия, он должен выполнить команду возврата `IRET`, которая, извлекая из стека содержимое FLAGS, CS, IP, возвращает управление прерванной программе.

Механизм реализации внешних и внутренних прерываний МП 80486 и Pentium в R-режиме аналогичен описанному выше, однако в P-режиме он значительно усовершенствован:

- таблица векторов прерываний трансформирована в дескрипторную таблицу прерываний IDT (рис. 7.13);
- более сложен процесс перехода к обработчику прерывания или особого случая;

- обработчику передается дополнительная информация о причине возникновения особого случая.

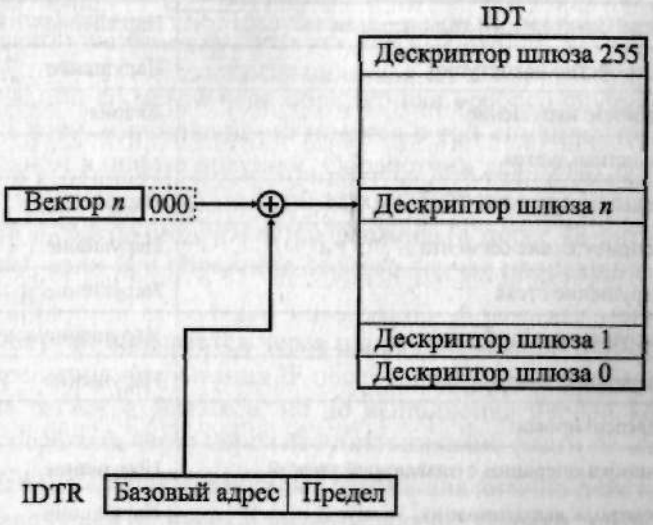


Рис. 7.13. Дескрипторная таблица прерываний

Механизм передачи управления обработчику особого случая (прерывания) соответствует обычному способу передачи управления через шлюз вызова. При этом процессор аппаратно включает в стек значения EFLAGS, CS : EIP (адрес возврата) прерываемой программы и, кроме того, в некоторых случаях — код ошибки и текущие значения SS, ESP (последние — при смене привилегий).

Точное значение адреса возврата зависит от того, является ли особый случай нарушением, ловушкой или аварией. Первые 32 вектора зарезервированы за особыми случаями P-режима (табл. 7.1).

Таблица 7.1. Особые случаи

Вектор	Причина	Тип
0	Ошибка деления	Нарушение
1	Отладка	Нарушение/ловушка
2	Немаскируемое прерывание NMI	Ловушка
3	Контрольная точка	Ловушка
4	Переполнение	Ловушка
5	Нарушение границы массива	Нарушение

Таблица 7.1 *(окончание)*

Вектор	Причина	Тип
6	Недействительный код операции	Нарушение
7	Устройство недоступно	Нарушение
8	Двойное нарушение*	Авария
9	Не используется	
10	Недействительный TSS*	Нарушение
11	Неприсутствие сегмента*	Нарушение
12	Нарушение стека*	Нарушение
13	Нарушение общей защиты*	Нарушение/ловушка
14	Страничное нарушение*	Нарушение
15	Зарезервирован	
16	Ошибка операции с плавающей точкой	Нарушение
17	Контроль выравнивания*	Нарушение
18—31	Зарезервированы	

Примечание. * — включает в стек код ошибки.

7.3.1. Дескрипторная таблица прерываний

Дескрипторная таблица прерываний IDT является прямой заменой таблицы векторов прерываний процессора 8086. Она должна определять 256 обработчиков прерываний и особых случаев, поэтому ее максимальный размер составляет 256 × 8 = 2048 байтов. Таблица IDT может находиться в любой области памяти, процессор локализует ее с помощью 48-битного регистра IDTR, который содержит базовый адрес и предел таблицы. Таблицу не рекомендуется объявлять короче максимального размера, т. к. любое обращение за пределы таблицы вызывает нарушение общей защиты, а вектор внешнего прерывания, вообще говоря, может иметь любое значение в диапазоне 00—FFh.

В таблице IDT разрешается применять только три вида дескрипторов: *шлюз ловушки*, *шлюз прерывания* и *шлюз задачи* (но не дескриптор TSS). Шлюзы прерывания и ловушки имеют большое сходство со шлюзом вызова (см. рис. 7.9). Единственное отличие состоит в отсутствии в этих шлюзах 5-битового поля счетчика WC, которое в шлюзе вызова определяет число параметров, передаваемых в вызываемую подпрограмму через стек. Соответствующее поле в шлюзах прерывания и ловушки зарезервировано.

Напомним, что шлюз вызова (а также ловушки и прерывания) содержит селектор сегмента кода и смещение внутри него, которые однозначно определяют точку передачи управления. Шлюз задачи содержит лишь селектор сегмента состояния задачи TSS (разумеется, каждый дескриптор содержит и байт доступа).

После локализации сегмента кода обработчика особого случая и включения информации в стек, выполнение начинается с той команды, которая определяется смещением в шлюзе ловушки. Обработчик действует до тех пор, пока *не достигнет команды* IRET. *По этой команде* процессор извлекает из стека адрес возврата и содержимое регистра флажков (а также 48-битный указатель внешнего стека, если при обработке особого случая происходила смена уровней привилегий).

Если особый случай вызывается через шлюз прерывания, процессор сбрасывает флаг разрешения прерывания IF после включения в стек адреса возврата и содержимое регистра флажков, но до выполнения первой команды обработчика. При переходе через шлюз ловушки никакие флаги не изменяются.

Обработка особого случая через шлюз задачи аналогична действию команды FAR CALL, приводящей к переключению задачи. Однако здесь невозможен прямой переход через дескриптор TSS, а требуется промежуточный шлюз задачи. С дескриптором шлюза задачи ассоциируется уровень привилегий, который не должен быть выше уровня привилегий прерываемой задачи. По существу, прерываемая задача как бы выполняет команду FAR CALL вызова другой задачи через шлюз задачи, и здесь действуют стандартные правила защиты по привилегиям.

Обработка особого случая через шлюз задачи, т. е. в другой задаче, имеет определенные преимущества:

- автоматически сохраняется весь контекст прерванной задачи;
- обработчик особого случая не может исказить прерванную задачу, т. к. он полностью изолирован от нее;
- обработчик *прерывания может работать на любом уровне привилегий и в* заведомо правильной среде; он может иметь свое локальное адресное пространство благодаря наличию отдельной локальной дескрипторной таблицы (при необходимости).

К недостаткам применения шлюза задачи для вызова обработчика можно отнести:

- замедленную реакцию процессора на особый случай;
- *в шлюзе задачи невозможно определить начальную точку выполнения* задачи;
- сложность получения информации о прерванной задаче.

Переключение задачи, инициируемое особым случаем, производит вложение задачи обработчика в прерванную задачу. Старая задача остается занятой, а новая — обработчик особого случая — отмечается как занятая, причем в ней будет установлен флажок NT и загружено поле обратной связи. Состояние флага IF в новой задаче не меняется и определяется тем значением бита регистра EFLAGS, которое хранилось в TSS. Поэтому, если обработчик особого случая через шлюз задачи предназначен для обработки внешних прерываний, следует в TSS установить IF = 0, "аппаратно" запретив внешние прерывания на время работы обработчика, или пока он явно не установит IF = 1.

7.3.2. Учет уровня привилегий

Все особые случаи должны обрабатываться через шлюзы. В дескрипторах шлюзов всех трех типов; содержащихся в IDT, имеется поле уровня привилегий дескриптора DPL, определяющее минимальный уровень привилегий, необходимый для использования шлюза. Для обработчиков прерываний рекомендуется устанавливать DPL = 3, чтобы обработка особого случая не зависела от уровня привилегий текущей задачи.

Шлюзы ловушек или прерываний должны передавать управление сегменту кода с более высоким или равным уровнем привилегий. Обработчику не разрешается работать на уровне привилегий, который ниже уровня прерываемой задачи (если, конечно, он не является отдельной задачей). Из-за непредсказуемости возникновения прерываний и особых случаев требуется гарантировать невозможность нарушения правил защиты по привилегиям при обработке особого случая. Этого можно достичь двумя способами:

- определить все обработчики особых случаев, не вызывающие переключения задачи, в сегментах кода с уровнем привилегий 0; такие обработчики будут действовать всегда, независимо от значения CPL программы;
- определить все обработчики особых случаев, не вызывающие переключения задачи, в подчиненные сегменты кода.

7.3.3. Код ошибки

В некоторых особых случаях процессор включает в стек 4 байта кода ошибки (error code), причем действительными являются только 2 младших байта, остальные включаются лишь для выравнивания стека. Когда процессор обнаруживает

- недействительный сегмент TSS;
- нарушение неприсутствия;

- нарушение стека;
- нарушение общей защиты,

он включает в стек обработчика особого случая информацию, идентифицирующую "виновный" дескриптор.

Формат кода ошибки напоминает селектор, т. к. большинство особых случаев связано с ошибками дескрипторов и содержит следующие поля:

- биты [15:3] — индекс (номер строки дескрипторной таблицы);
- бит [2] — TI, как и в других селекторах, определяет принадлежность дескриптора к локальной (TI = 1) или глобальной (TI = 0) дескрипторной таблице;
- бит [1] — I. Если I = 1, то индекс в старших битах [15:3] кода ошибки относится к дескрипторной таблице прерываний IDT;
- бит [0] — EXT = 1 означает, что особый случай был вызван аппаратным прерыванием (внешним) или возник, когда процессор обрабатывал другой особый случай.

Если процессор не может сформировать содержательный код ошибки, он включает в стек код, равный 0.

Помимо кода ошибки в некоторых особых случаях дополнительная диагностическая информация находится в других регистрах процессора. Например, при страничном нарушении в регистре CR2 содержится линейный адрес, преобразование которого привело к ошибке. Обработчик этого особого случая может обратиться к соответствующим элементам PDE и PTE. Для особого случая отладки полезная информация содержится в регистре состояния отладки DR6.

7.3.4. Описание особых случаев

Далее приводится краткое описание действий процессора *x*86 при возникновении особых случаев [3].

Ошибка деления (0) — автоматически формируется, когда в команде `DIV` или `IDIV` делитель равен нулю или частное слишком велико для получателя (AL/AX/EAX).

Отладка (1) — формируется в следующих случаях (может быть нарушением или ловушкой):

- нарушение контрольной точки по адресу команд;
- ловушка контрольной точки по адресу данных;
- нарушение общей защиты;

- ловушка покомандной работы (флаг TF =1);
- ловушка контрольной точки по переключению задачи (в сегменте TSS бит T = 1).

Немаскируемое прерывание NMI (2) — единственное внешнее радиальное прерывание.

Контрольная точка (3) — формируется при выполнении команды INT3 (код операции — CCh). Передача управления обработчику особого случая является частью команды INT3, адрес возврата в стеке относится к началу следующей команды. Тот же обработчик вызывается при выполнении внешнего прерывания с вектором 03 или двухбайтовой команды INT 03.

Переполнение (4) — возникает при выполнении команды INTO при условии установки в 1 флага переполнения OF. Как и для INT3, передача управления обработчику особого случая является частью команды INTO, адрес возврата в стеке относится к началу следующей команды. Обычно команда INTO применяется в компиляторах для выявления переполнения в арифметике знаковых чисел. Тот же обработчик вызывается при выполнении внешнего прерывания с вектором 04 или команды INT 04.

Нарушение границы массива (5) — возникает при выполнении команды BOUND, если контрольная проверка дает отрицательный результат, т. е. проверяемый (первый) операнд не попадает в диапазон значений, определенных вторым (нижняя граница) и третьим (верхняя граница) операндами команды.

Недействительный код операции (6) — генерируется, когда операционное устройство процессора обнаруживает неверный код операции, несоответствие типа операндов коду операции, попытку выполнения привилегированных команд в R-режиме, неверные байты mod r/m или sib, использование префикса блокировки LOCK с командами, которые нельзя блокировать. Характерно, что существует несколько одно- и двухбайтовых кодов, зарезервированных фирмой Intel для развития системы команд, и хотя им в 80486 не соответствуют никакие команды, зарезервированные коды не вызывают особого случая.

Устройство[1] *недоступно* (7) — возникает в двух ситуациях:

- процессор выполняет команду ESC и бит EM (эмуляция сопроцессора) в регистре CR0 установлен в 1;
- процессор выполняет команду WAIT или ESC и бит TS (переключение задачи) в регистре CR0 установлен в 1.

В первом случае программист намерен выполнить операции плавающей арифметики программно.

---

[1] Под устройством здесь понимается аналог математического сопроцессора — устройство с плавающей точкой FPU.

Второй случай может возникнуть после переключения задачи. Бит TS аппаратно устанавливается в 1 при переключении задачи, а первая же встретившаяся в новой задаче команда сопроцессора вызывает особый случай 7, ибо контекст устройства с плавающей точкой старой задачи не сохранен. Обработчик особого случая 7 сохраняет старое состояние устройства с плавающей точкой в сегменте TSS старой задачи и загружает новое состояние устройства из сегмента TSS новой задачи. (В случае работы цепочки вложенных задач обработчик должен программно отследить ту старую задачу, которая последней использовала FPU.) После этого привилегированной командой CLTS сбрасывается флажок TS и осуществляется возврат на команду устройства с плавающей точкой. Так как теперь флаг TS сброшен, команда сопроцессора будет выполнена.

Двойное нарушение (8) — обычно, когда процессор обнаруживает особый случай при попытке вызвать обработчик предыдущего особого случая, два особых случая обрабатываются последовательно. Если процессор не может обрабатывать их последовательно, он сигнализирует о двойном нарушении.

Для определения того, когда о двух нарушениях следует сообщать как о двойном нарушении, процессор подразделяет все особые случаи на три класса:

- легкие особые случаи — векторы 1, 2, 3, 4, 5, 6, 7,16;
- тяжелые особые случаи — векторы 0, 10, 11, 12, 13;
- страничные нарушения — вектор 14.

Когда возникают два легких особых случая или один легкий и один тяжелый, эти два события допускают последовательную обработку. При появлении двух тяжелых событий их обработать нельзя, поэтому процессор формирует особый случай двойного нарушения. Аналогичное состояние наступает, если после страничного нарушения возникает тяжелый особый случай или второе страничное нарушение, хотя если, наоборот, после тяжелого особого случая возникает страничное нарушение, эти события могут быть обработаны.

Если при попытке вызвать обработчик двойного нарушения возникает любое другое нарушение, процессор переходит в режим отключения. Этот режим аналогичен состоянию процессора после выполнения команды HLT. До восприятия сигналов NMI или RESET никакие команды не выполняются, причем если отключение возникло при выполнении обработчика немаскируемого прерывания, запустить процессор может только сигнал RESET.

Недействительный сегмент TSS (10) — возникает при попытке переключения на задачу с неверным сегментом TSS. Поскольку сегмент TSS может определять LDT, сегменты кода, стека, данных, к особому случаю 10 относятся ситуации с нарушением границ этих сегментов, нарушением прав доступа,

запретом записи в сегмент стека и др. Нарушения могут возникать как в контексте старой, так и новой задачи, поэтому обработчик особого случая 10 должен сам быть задачей и вызываться через шлюз задачи (десятая строка IDT должна содержать шлюз задачи).

Неприсутствие сегмента (11) — формируется, когда бит присутствия сегмента в дескрипторе P = 0. Это нарушение допускает рестарт, если обработчик особого случая 11 реализует механизм виртуальной памяти на уровне сегментов.

Нарушение стека (12) — возникает в двух ситуациях:

- в результате нарушения предела любой операции, которая обращается к регистру SS (`POP`, `PUSH`, `ENTER`, неявное использование стека при обращении к памяти, например, `MOV AX, [BP+6]`);
- при попытке загрузить в регистр SS дескриптор, который отмечен неприсутствующим.

Нарушение общей защиты (13) — все нарушения защиты, которые не служат причиной конкретного особого случая, вызывают особый случай общей защиты:

- превышение предела сегмента (кроме стека);
- передача управления сегменту, который не является выполняемым;
- запись в защищенный от записи сегмент;
- считывание из выполняемого сегмента;
- загрузка в SS селектора сегмента, защищенного от записи;
- загрузка в регистры SS, DS, ES, FS, GS селектора системного сегмента;
- загрузка в регистры SS, DS, ES, FS, GS селектора выполняемого сегмента;
- обращение к памяти через DS, ES, FS, GS, когда в них пустой селектор;
- переключение на занятую задачу;
- нарушение правила привилегий

и др.

Страничное нарушение (14) — возникает, когда разрешено страничное преобразование и имеет место одна из следующих ситуаций:

- в элементе каталога разделов или таблицы страниц, используемом для преобразования линейного адреса в физический, сброшен бит присутствия;
- процедура не имеет достаточного уровня привилегий для доступа к адресуемой странице.

Ошибка операции с плавающей точкой (16) — сигнализирует об ошибке, возникшей в команде устройства с плавающей точкой.

Контроль выравнивания (17) — возникает при нарушении выравнивания операндов. Операнды считаются выровненными, если адрес двухбайтового слова является четным (младший разряд равен 0), адрес четырехбайтового двойного слова кратен 4, а адрес восьмибайтовой структуры данных кратен 8. Для разрешения контроля выравнивания должны выполняться три условия:

- бит AM в регистре CR3 установлен;
- флаг AC установлен;
- выполняется программа на уровне привилегий 3.

7.4. Средства отладки

Традиционно средства отладки микропроцессоров ограничивались наличием:

- короткой команды программного прерывания, которую можно было устанавливать вместо первого байта любой команды;
- аппаратной реализации пошагового (покомандного) режима, который инициировался установкой специального бита T в регистре флагов.

По мере усложнения МПС возможности внешних аппаратных средств по наблюдению операций, происходящих внутри процессора, уменьшаются. Поэтому в схемах мощных процессоров стали предусматривать разнообразные средства отладки.

Основу средств отладки в процессорах *x*86 старших моделей составляют специализированные регистры отладки — программируемые регистры задания контрольных точек, регистры управления и состояния отладки. Они заменяют собой средства аппаратных внутрисхемных эмуляторов. Процессоры *x*86 старших моделей обеспечивают не только покомандную работу, но и регистрацию переключения на конкретную задачу, установку контрольных точек по адресам команд и фиксацию модификации значений переменных в памяти.

Регистры отладки поддерживают контрольные точки по командам и данным. В общем, под *контрольной точкой* понимается адрес, при использовании которого программой возникает особый случай отладки. Установка контрольной точки по команде обеспечивает регистрацию команды по любому линейному адресу. Задание контрольной точки по данным позволяет узнать, когда производится обращение к конкретной переменной.

Отладочные средства *x*86 включают в себя:

- однобайтовую команду контрольной точки INT3, которую можно вставлять в программу по любому адресу; при выполнении этой команды генерируется особый случай отладки;

- флаг пошагового режима TF в регистре EFLAGS, позволяющий выполнить программу по командам;
- четыре регистра отладки DR0—DR3, которые определяют четыре независимые контрольные точки по командам или данным; регистр управления отладкой DR7 и регистр состояния отладки DR6;
- флаг ловушки T в TSS, который вызывает особый случай отладки при переключении на задачу с установленным в 1 битом T;
- флаг возобновления RF в регистре EFLAGS, с помощью которого подавляются многократные особые случаи в одной и той же команде.

Все эти средства действуют как *ловушки*, следя за возникновением условий, представляющих интерес для программиста. Когда возникает такое условие, формируется особый случай отладки (с вектором 1); только команда INT3 генерирует прерывание с вектором 3. Зарезервированный вектор отладки 1 упрощает процедуру вызова отладчика.

Итак, отладчик по вектору 1 вызывается в следующих случаях:

- при TF = 1 после каждой команды;
- при TTSS = 1 в момент переключения на задачу;
- ловушка контрольной точки по данным;
- нарушение контрольной точки по команде.

Команда INT3 предоставляет альтернативный способ задания контрольной точки и особенно удобна, если контрольные точки размещаются в исходном коде или требуется установить более четырех контрольных точек.

Программные контрольные точки задаются путем замены обычных команд на команды INT3 — при этом требуется осуществлять запись в сегмент кода (создавать альтернативный сегмент данных), а после отладки — восстанавливать исходный код. Использование аппаратных контрольных точек исключает необходимость модификации кода (можно отлаживать программу, размещенную в ПЗУ), а так же допускает контроль обращения к данным.

Рассмотренные средства позволяют вызывать отладчик как процедуру в контексте текущей задачи или как отдельную задачу при выполнении одного из следующих условий:

- выполнение команды контрольной точки INT3;
- выполнение любой команды (при TF = 1);
- выполнение команды по указанному адресу;
- считывание или запись байта, слова или двойного слова по указанному адресу;

- запись байта, слова или двойного слова по указанному адресу;
- переключение на конкретную задачу;
- попытка изменить содержимое регистра отладки.

7.4.1. Регистры отладки

Регистры отладки, форматы которых приведены на рис. 7.14, включают в себя:

- четыре регистра DR0—DR3, предназначенные для хранения линейных адресов четырех контрольных точек, каждая из которых независимо может быть определена как контрольная точка по команде или по данным;
- регистр DR7 управления отладкой, включающий поля, которые определяют свойства контрольных точек и некоторые параметры процесса отладки;
- регистр DR6 состояния отладки, предназначенный для идентификации причины прерывания отладки;
- наконец, зарезервированные регистры DR5, DR4.

LEN3	RW3	LEN2	RW2	LEN1	RW1	LEN0	RW0	GD	GE	LE	G3	L3	G2	L2	G1	L1	G0	L0	DR7
								BT	BS	BD	000000000				B[3:0]				DR6
Зарезервирован																			DR5
Зарезервирован																			DR4
Линейный адрес контрольной точки 3																			DR3
Линейный адрес контрольной точки 2																			DR2
Линейный адрес контрольной точки 1																			DR1
Линейный адрес контрольной точки 0																			DR0

Рис. 7.14. Форматы регистров отладки

Рассмотрим форматы регистров состояния и управления.

Выше отмечалось, что все события отладки, кроме INT3, вызывают прерывание с вектором 1. Следовательно, при возникновении особого случая отладки встает вопрос о причине прерывания. Именно *для идентификации причин прерывания 1* предусмотрен регистр состояния отладки DR6.

Младшие четыре бита B0—B3 относятся к четырем контрольным точкам и единичное состояние B*i* означает достижение контрольной точки, линейный адрес которой находится в регистре DR*i*.

Флаг BS (Step) устанавливается в 1, когда процессор начинает отрабатывать особый случай, вызванный ловушкой покомандной работы, т. е. при BS = 1 причиной особого случая является состояние TF = 1 регистра EFLAGS. Этот

случай имеет высший приоритет среди всех случаев отладки, когда BS = 1, могут быть установлены и другие биты состояния отладки.

Флаг BT (Task) устанавливается в 1, когда особый случай отладки вызван переключением на задачу, в TSS которой установлен бит ловушки T =1.

Регистры отладки доступны (по записи или чтению) только в реальном режиме или в защищенном режиме по привилегированной (т. е. разрешенной к выполнению только на нулевом уровне) команде MOV:

```
mov eax, dr6
mov dr1, eax
```

В регистре DR7 предусмотрен флаг GD, который, будучи установленным, обеспечивает "сверхзащиту" всех обращений к регистрам отладки, вызывая при любой попытке обращения к этим регистрам *прерывание 1*. Для идентификации этой ситуации в DR6 предусмотрен бит BD. Он устанавливается в 1, если следующая команда будет считывать или записывать в один из восьми регистров отладки. Характерно, что при вызове процедуры обработчика с вектором 1 бит GD автоматически сбрасывается, что обеспечивает обработчику возможность доступа к регистрам отладки.

Процессор никогда не сбрасывает биты регистра состояния отладки DR6, поэтому обработчик особого случая должен сбрасывать их программно, иначе причины особых случаев отладки будут накапливаться.

Возможно, при возникновении особого случая отладки в состоянии 1 будут находиться несколько битов DR6 (что возможно при разрешении аппаратных контрольных точек) или, наоборот, в DR6 не окажется единичных битов. Последнее возможно, если возникло внешнее прерывание с вектором 1 или выполняется команда INT1.

Регистр управления отладкой DR7 содержит для каждой из четырех контрольных точек следующие поля, определяющие ее характеристики: L*i*, G*i*, RW*i* и LEN*i*, а также два однобитовых поля LE и GE, определяющие свойства, общие для всех контрольных точек. Назначение этих полей приведены в табл. 7.2. Кроме того, в регистре DR7 бит 13 — GD — предназначен для включения режима защиты регистров отладки от любого обращения со стороны программ пользователя.

Таблица 7.2. Назначение полей регистра DR7

Поле	Назначение
L*i*	Локальное разрешение контрольной точки $i \in \{0, 1, 2, 3\}$. Когда бит L*i* находится в 1, разрешена аппаратная контрольная точка, линейный адрес которой находится в DR1, но только в текущей задаче. При переключении задачи сбрасываются все биты L*i*

Таблица 7.2 *(окончание)*

Поле	Назначение
G*i*	Глобальное разрешение контрольной точки *i*. Функционирует аналогично биту L*i*, но на него не действует переключение задач. Сбросить бит G*i* можно только программно
LE	Локальная точность
GE	Глобальная точность
RW*i*	Определяет вид контрольной точки *i*: • 00 — контрольная точка по команде; • 01 — контрольная точка по данным с обращением по записи; • 10 — не определена и не используется; • 11 — контрольная точка по данным с обращением по считыванию или записи (но не выбор команды)
LEN*i*	Для контрольной точки *i* по данным определяет длину данных: • 00 — однобайтовое поле (и все команды); • 01 — двухбайтовое поле (слово); • 10 — не определена и не используется; • 11 — четырехбайтовое поле (двойное слово)

Контрольная точка может быть локальной (в пределах одной задачи) или глобальной — в зависимости от значений битов L*i* и G*i*.

Допускается одновременное значение L*i* = G*i* = 1 или L*i* = G*i* = 0. В первом случае это эквивалентно G*i* = 1, а во втором — контрольная точка запрещена и соответствие линейного адреса из регистра DR*i* адресу команды не вызывает особого случая (но B*i* в регистре DR6 устанавливается в 1).

Аппаратные контрольные точки по командам устанавливаются путем загрузки в один из регистров DR*i* линейного адреса требуемой команды, установки в 00 соответствующих полей RW*i* и LEN*i*, установки в 1 бита L*i* и/или G*i*. После этого процессор начинает контролировать устройство предвыборки команд. Когда фиксируется равенство адреса команды и содержимого одного из "разрешенных" регистров DR*i*, хранящих контрольную точку по командам, формируется особый случай отладки, причем в DR6 устанавливается в 1 бит B*i*.

Адреса команд в регистрах DR0—DR3 должны быть 32-разрядными линейными, а не логическими *(селектор : смещение)* или физическими. Линейный адрес не зависит от страничного преобразования, поэтому контрольная точка действует даже тогда, когда целевая команда участвует в свопинге и отображается на различные адреса физической памяти.

Если контрольная точка установлена как локальная, она сбрасывается при переключении задачи, причем значение DR7 не сохраняется в TSS, поэтому при восстановлении задачи контрольные точки не возобновляются. При необходимости следует предусмотреть программное возобновление контрольных точек. Для этого можно в расширении сегмента TSS той задачи, в которой определены локальные контрольные точки, записать значения DR0—DR3 и DR7, а так же установить в TSS бит T = 1. При переходе к такой задаче вызывается обработчик особого случая, который и восстановит значения регистров DR из сегмента TSS.

Аппаратные контрольные точки по командам являются *нарушениями*, т. е. процессор включает в стек адрес команды, вызвавшей нарушение, и обработчик особого случая возвращает управление на ту же команду. Поскольку аппаратные контрольные точки по команде проверяются до выполнения самой команды, процессор должен вновь сформировать особый случай. Необходимо обойти данную команду, для этого используется флаг возобновления RF в регистре EFLAGS. Процессор автоматически устанавливает RF = 1 при возникновении любого нарушения, включая и аппаратные контрольные точки по командам. Аппаратные прерывания по входам INTR и NMI, а так же программные ловушки и аварии не воздействуют на флаг RF. Когда RF = 1, процессор игнорирует особый случай аппаратной контрольной точки по командам, а после первой же команды, которая выполнена без особых случаев, процессор сбрасывает RF.

Аппаратные контрольные точки по данным устанавливаются с помощью тех же регистров отладки DR. Процессор формирует особый случай отладки как *ловушку*, когда происходит обращение к данным по установленным в DR адресам. Разрешается совместное применение контрольных точек по командам и по данным, причем возможно произвольное назначение типа контрольной точки (поле RW*i* в регистре DR7).

Процессор контролирует выравнивание данных, если их длина, указанная в поле LEN*i*, равна слову или двойному слову. В этом случае при сравнении текущего адреса сегмента данных с адресом контрольной точки игнорируются один или два младших бита регистра DR*i*.

Конвейерная архитектура старших моделей *x*86 обеспечивает одновременную обработку нескольких команд. Случается, что контрольная точка по данным фиксируется только после выполнения нескольких следующих команд. В регистре управления отладкой DR6 предусмотрены биты, задающие локальную LE и глобальную GE "точность" определения контрольной точки по данным. Будучи установленными в 1, эти биты замедляют внутренние операции процессора таким образом, что сообщают об обращении по контролируемому адресу данных точно в тот момент, когда происходит обращение к памяти.

Биты LE, GE действуют только на контрольные точки по данным, являются общими для всех таких точек, причем LE автоматически сбрасывается при переключении задачи, а GE может быть сброшен лишь программно.

Регистрация нескольких особых случаев

Если команда, на которую настроена контрольная точка, вызывает данные по контролируемому адресу, процессор правильно сформирует два особых случая отладки. Первый — по команде — является нарушением, и обработчик этого особого случая возвращает управление той же команде. При выполнении команды второй особый случай не возникает (см. выше назначение бита RF), зато фиксируется ловушка по данным. Причина текущего особого случая фиксируется в DR6.

7.5. Увеличение быстродействия процессора

Одним из самых распространенных способов определения производительности процессора является оценка времени T решения некоторой (тестовой) задачи. Очевидно,

$$T = \frac{N \cdot S}{f}, \tag{7.1}$$

где N — количество выполненных при решении задачи машинных команд; S — среднее количество тактов, приходящихся на выполнение одной команды; f — тактовая частота процессора.

Если длительности различных команд (в тактах) существенно отличаются друг от друга, то более точно можно оценить значение T по выражению

$$T = \frac{1}{f} \cdot \sum_{i=1}^{N} S_i, \tag{7.2}$$

где S_i — число тактов i-й команды.

Используются и более точные (и, соответственно, более сложные) методы оценки производительности [7, 11, 12], однако и из выражений (7.1), (7.2) видны пути увеличения производительности процессора:

- увеличение тактовой частоты (решения лежат в области технологии СБИС);
- сокращение длины программы (совершенствование технологии программирования, разработка оптимизирующих компиляторов);
- сокращение числа тактов, приходящихся на выполнение одной команды.

Последнее возможно за счет усложнения схемы процессора, при этом значительное усложнение может привести к сокращению числа тактов команды при увеличении "глубины схемы", что повлечет за собой увеличение длительности такта, так что выигрыш может обернуться проигрышем.

Магистральным путем увеличения производительности ЭВМ можно считать *параллелизм* на различных уровнях.

Существуют две основные формы параллелизма [11]:

- параллелизм на уровне процессов;
- параллелизм на уровне команд.

В первом случае над одной задачей могут одновременно работать несколько процессоров или других устройств ЭВМ.

Во втором случае параллелизм реализуется в пределах отдельных команд.

Обычно стремятся совмещать во времени процедуры обращения к памяти и обработки информации, параллельно выполнять арифметические операции сразу над несколькими (или даже всеми) разрядами операндов, одновременно выполнять несколько последовательных команд программы (разумеется, на разных стадиях) и т. п.

Уже в младшей модели семейства *x*86 — микропроцессоре 8086 предусматривалась одновременная работа двух основных устройств — *обработки данных* и *связи с магистралью*. Подобный механизм (с модификациями) сохранился и в старших моделях семейства.

Особенно эффективным способом организации параллельных операций в компьютерной системе является *конвейерная обработка команд*.

Далее мы кратко рассмотрим некоторые из перечисленных методов увеличения производительности процессора. Более подробную информацию по этим вопросам можно найти, например, в [11, 12].

7.5.1. Конвейеры

При отсутствии конвейера процессор выполняет программу, по очереди выбирая из памяти и активизируя ее команды.

Процесс обработки команды может быть разбит, например, на следующие шаги (стадии):

- F — выборка (от англ. *fetch*) — чтение команды из памяти;
- D — декодирование (от англ. *decode*) — декодирование команды;
- A — формирование адресов (от англ. *address generate*) и выборка операндов;

- E — выполнение (от англ. *execute*) — выполнение заданной в команде операции;
- W — запись (от англ. *write*) — сохранение результата по целевому адресу.

Приведенное разбиение не является единственно возможным — в некоторых случаях рассматривают четырехстадийный командный цикл, иногда (например, для процессоров, реализующих команды над числами с плавающей запятой) — восьмистадийный и др.

Для реализации каждой из стадий командного цикла в процессоре предусмотрено соответствующее оборудование (регистры, дешифраторы, сумматоры, управляющие автоматы или их фрагменты), причем операционные элементы разных стадий обычно слабо пересекаются между собой. Поэтому когда очередная команда завершает действия на одной стадии, например F, и переходит на следующую — D, то оборудование стадии F "простаивает" и может быть использовано для чтения следующей команды. Таким образом, очередная команда может начинать выполнение, не дожидаясь окончания командного цикла предыдущей команды.

При рассмотрении пятистадийного командного цикла одновременно на разных стадиях может выполняться до пяти команд. Организация *пятистадийного конвейера* потребует дублирования некоторых операционных элементов на разных стадиях (например, регистра команд PC) и усложнение схемы управления, однако игра стоит свеч.

Очевидно, очередная команда может перейти с одной стадии командного цикла на другую при выполнении двух условий:

- действия команды на текущей стадии завершены;
- предыдущая команда освободила оборудование следующей стадии.

При условии, что каждая стадия выполняется в любой команде одинаковое количество тактов (например, один), конвейер работает идеально, и одновременно всегда выполняются пять команд (для пятистадийного конвейера).

Для большинства процессоров т. н. CISC[1]-архитектуры (к ним относятся, в частности, процессоры семейства *x*86) такая идеальная ситуация складывается далеко не всегда.

Действительно, команда, извлекаемая из памяти на стадии F, может иметь разную длину и, следовательно, извлекаться из памяти за разное число машинных циклов.

В зависимости от заданного в команде способа адресации операндов время выполнения стадии A может быть существенно различным (сравните непо-

[1] Complex Instruction Set Computer — компьютер с полным набором команд.

средственную адресацию и косвенно-автоинкрементную). Расположение адресуемых операндов (и размещение результата) в памяти разного уровня так же существенно влияет на время реализации стадии A (и стадии W).

Наконец, на стадии E время выполнения операции зависит не только от типа операции (короткие — сложение, конъюнкция, ..., длинные — умножение, деление), но даже иногда и от значений операндов.

Учитывая отмеченные выше обстоятельства, можно сказать, что конвейеры процессоров с классической CISC-архитектурой редко работают "на полную мощность", находясь значительную часть времени в ожидании завершения "длинных" операций.

Желание увеличить производительность конвейеров привело к появлению процессоров т. н. RISC[1]-архитектуры, из систем команд которых были исключены все факторы, тормозящие реализацию командного цикла — длинные команды, сложные способы адресации, размещение операндов в ОЗУ. К особенностям RISC-архитектуры можно отнести:

- форматы всех команд имеют одинаковую длину, в крайнем случае, разнообразие длин форматов ограничивается двумя вариантами;
- все операции выполняются за одинаковый промежуток времени (обычно 1 или 2 такта);
- операнды всех арифметических и логических операций располагаются только в регистрах, к оперативной памяти обращаются только команды загрузки и сохранения;
- сверхоперативная память представлена большим числом регистров (32—256).

Реализация этих особенностей, с одной стороны, позволяет приблизить работу конвейера к идеальной, с другой стороны — существенно ограничивает возможности системы команд процессора. Действительно, из системы операций исключаются "длинные" операции — умножение, деление, операции над числами с плавающей запятой и др. Исключаются сложные (и эффективные) способы адресации, например, автоиндексные. Это приводит к значительному увеличению длины программ, увеличению времени на выборку команд из памяти, при этом среднее число команд, выполняемых в единицу времени, в RISC-процессорах значительно больше, чем в CISC-процессорах.

По мере совершенствования интегральной технологии появилась возможность ценой значительных аппаратных затрат (теперь разработчики уже могли их себе позволить) резко сократить время выполнения "длинных" опера-

[1] Reduced Instruction Set Computer — компьютер с сокращенным набором команд.

ций, например, за счет реализации матричного умножителя, табличной арифметики и др. В этом случае длинные операции можно включать в систему команд RISC-процессоров, не нарушая принципов их организации, но увеличивая эффективность системы команд.

В настоящее время понятия "RISC-" и "CISC-архитектура" скорее являются обозначением некоторых принципов проектирования, но не характеристиками конкретных процессоров. Современные процессоры, как правило, реализованы по "гибридному" принципу: содержат ядро RISC, которое выполняет простые и самые распространенные команды за один такт на стадию конвейера, а сложные команды интерпретируются как некая последовательность простых (на уровне микрокоманд). Пользователь же в этом случае имеет дело с системой команд привычной CISC-архитектуры, а внутренние вопросы реализации командного цикла его, как правило, не интересуют.

При реализации конвейеров возникает еще одна проблема, связанная с его оптимальной загрузкой — команды условной передачи управления (в среднем каждая 5—6-я команда программы). Действительно, когда такая команда передается со стадии F на стадию D, на стадию F надо ставить следующую команду, но какую? Условие перехода будет проверено лишь на стадии E, тогда же определится адрес следующей команды. Здесь возможны два пути решения:

- приостановить загрузку конвейера до завершения командой перехода стадии E;
- загрузить конвейер "наугад" командой по одному из двух возможных адресов, а на стадии E проверить правильность выбора и, если он оказался неверным — очистить весь конвейер и начать загрузку заново по правильному адресу.

Второй путь представляется предпочтительным, т. к. конвейер не останавливается, и (в среднем) в половине случаев мы избежим потери времени на перезагрузку. Результаты работы конвейера будут еще лучше, если мы научимся правильно предсказывать адрес перехода, чтобы вероятность угадывания адреса приближалась к 1.

В современных процессорах часто предусматривают специальные аппаратные блоки предсказания переходов. В разных процессорах реализуются различные алгоритмы предсказания, основанные на анализе результатов выполнения предыдущих команд переходов [12].

7.5.2. Динамический параллелизм

Один конвейер хорошо, а два лучше? Почему бы, если позволяют ресурсы интегральной технологии, не построить два конвейера и ставить на них одно-

временно пару команд программы. В процессоре Pentium именно так и сделали — предусмотрели два 5-уровневых конвейера, которые могли работать одновременно и выполнять две целочисленные команды за машинный такт.

Однако возможность одновременной постановки на два разных конвейера пары последовательных команд программы ограничивается рядом условий. Очевидно, что нельзя ставить на разные конвейеры две последовательные команды, если вторая использует в качестве операнда результат работы первой или, во всяком случае, необходимо гарантировать, что к началу стадии выборки операндов второй команды первая (на другом конвейере) уже завершит стадию размещения результата. Существуют и другие ограничения, которые определяют т. н. условия "спаривания" последовательных команд (pairing), позволяющие размещать их одновременно на разных конвейерах.

В процессоре Pentium два конвейера не являются равноправными. Один из них (U) может принять любую команду, а другой (V) — только удовлетворяющую условиям "спаривания" (довольно сложным) с командой, поставленной на U. Если эти условия не соблюдаются, следующая команда так же помещается на U-конвейер, а V-конвейер пропускает такт. Некоторые команды могут появляться только на U-конвейере.

Разумеется, производительность двухконвейерного процессора, при прочих равных условиях, превышает производительность одноконвейерного, но далеко не в два раза. Эффективность во многом определяется, насколько часто будут встречаться в программе пары последовательных команд, допускающих "спаривание".

Очевидно, движение в направлении увеличения в процессоре числа конвейеров, работающих по описанным выше принципам, бесперспективно. Условия "страивания" (если можно так сказать) и т. д. будут настолько сложны, что редко будут выполняться.

Следующим шагом на пути увеличения производительности было решение, которое принято называть *динамическим параллелизмом*, а процессоры, реализующие этот принцип, называют *суперскалярными*.

Рассмотрим фрагмент программы, написанный на некотором условном языке:

```
MOV R1, R4
ADD R1, @R0
MOV R5, R6
SUB R5, R7
CLR R6
. . .
```

Вторая команда этого фрагмента использует в качестве операнда содержимое ячейки памяти (косвенно-регистровая адресация) и, следовательно, попав на

конвейер, будет "тормозить" его[1] на стадии А. В то же время следующие три команды этого фрагмента никак не связаны с результатами работы второй команды, выполняют действия только над регистрами и могли бы выполняться еще до завершения предыдущей команды, однако в этом случае пришлось бы изменить порядок выполнения команд, определенный программой, чего конвейер не предусматривает.

Суперскалярная архитектура процессора предполагает, что команды (на некотором ограниченном участке программы) могут выполняться не только в порядке их размещения в программе, но и по мере возможности их выполнения независимо от порядка следования. Возможность выполнения определяется, во-первых, отсутствием зависимостей от ранее расположенных, но еще не завершенных команд, во-вторых, наличием свободных ресурсов процессора, необходимых для выполнения команды.

Одним из первых микропроцессоров, реализующих механизм динамического параллелизма, был процессор Pentium Pro (Pentium II) фирмы Intel (рис. 7.15).

На кристалле процессора размещаются два блока кэш-памяти первого уровня, в одном из которых (кэш-C) размещается программа, а в другом (кэш-D) — данные.

Рис. 7.15. Структура процессора Pentium Pro

[1] Здесь мы не обсуждаем возможности кэш-памяти.

Устройство выборки/декодирования выбирает очередную команду из кэш-С (в порядке их размещения в программе), при необходимости заменяет сложные команды на последовательность микрокоманд, снабжает каждую команду полем признаков (тегом) и помещает в специальным образом организованную память — *пул команд*.

Устройство диспетчирования постоянно анализирует, с одной стороны, содержимое пула команд и выявляет команды, готовые к выполнению на какой-нибудь стадии, с другой стороны — свободные в данный момент операционные устройства. При совпадении "желания" (команда завершила предыдущую стадию и готова к выполнению следующей) и "возможностей" (свободны соответствующие ресурсы) устройство диспетчирования отправляет команду на выполнение независимо от порядка поступления команд в пул. После завершения обработки на очередной стадии команда возвращается в пул с соответствующей пометкой в поле тега.

Устройство отката размещает результаты выполнения команд по адресам назначения. Оно просматривает содержимое пула команд, отыскивает команды, завершившие работу, и извлекает их из пула с размещением результата в строгом соответствии с порядком расположения команд в программе.

Небольшое количество регистров в архитектуре процессоров Intel приводит к интенсивному использованию каждого из них и, как следствие, к возникновению множества мнимых зависимостей между командами, использующими один и тот же регистр. Поэтому, чтобы исключить задержку в выполнении команд из-за мнимых зависимостей, устройство диспетчирования/выполнения работает с дублями регистров, находящимися в пуле команд (одному регистру может соответствовать несколько дублей).

Реальный набор регистров контролируется устройством отката, и результаты выполнения команд отражаются на состоянии вычислительной системы только после того, как выполненная команда удаляется из пула команд в соответствии с истинным порядком команд в программе.

Таким образом, принятая в Pentium Pro технология динамического выполнения может быть описана как оптимальное выполнение программы, основанное на предсказании будущих переходов, анализе графа потоков данных с целью выбора наилучшего порядка исполнения команд и на опережающем выполнении команд в выбранном оптимальном порядке. Однако следует иметь в виду, что процессор оптимизирует выполнение только ограниченного участка программы, который в текущий момент располагается в пуле.

Суперскалярная архитектура предполагает наличие на кристалле процессора нескольких параллельно работающих операционных устройств (в т. ч. и нескольких одинаковых). Так, например, RISC-процессор PowerPC содержит шесть параллельно работающих исполнительных устройств: блок предсказа-

ния ветвлений, два устройства для выполнения простых целочисленных операций (сложение, вычитание, сравнение, сдвиги, логические операции), одно устройство для выполнения сложных целочисленных операций (умножение, деление), устройство обработки чисел с плавающей запятой и блок обращения к внешней памяти. При этом обеспечивается одновременное выполнение четырех команд.

Все операции обработки данных выполняются с регистровой адресацией. При этом для хранения целочисленных операндов используется блок, включающий тридцать два 32-разрядных регистра, а для хранения операндов с плавающей запятой — блок из тридцати двух 64-разрядных регистров.

Выборка данных из памяти производится только командами пересылки, которые выполняются блоком обращения к памяти и осуществляют загрузку данных в регистры или запись их содержимого в память.

При параллельной работе исполнительных устройств возможно их одновременное обращение к одним регистрам. Чтобы *избежать ошибок*, возникающих при этом в случае записи нового содержимого до того, как другим устройством будет считано предыдущее, введены буферные регистры — 12 для целочисленных регистров и 8 — для регистров с плавающей запятой. Эти регистры служат для промежуточного хранения операндов, дублируя основные регистры блоков, используемые при выполнении текущих операций. После завершения операций производится перезапись полученных результатов в основные регистры (обратная запись).

7.5.3. VLIW-архитектура

Для реализации динамического параллелизма в процессорах с традиционной системой команд и способами компиляции программного кода требуются весьма сложные схемы организации пула, планировщики, схемы "отката" и др. Процессоры такой архитектуры имеют несколько операционных блоков различного, а иногда и одинакового назначения, которые могут работать параллельно, например, 1—2 блока вычисления адресов, 2—3 блока АЛУ для *чисел с фиксированной* запятой, блок обработки чисел с плавающей запятой, блок размещения результата, блок предсказания переходов и др.

Поскольку процессор может планировать и формировать последовательность выполнения команд на ограниченном (размером пула) участке программы, то для эффективной загрузки операционных блоков требуются не только сложные и эффективные процедуры планирования, но и некоторое "везение" — хорошо, если в пул загружены команды, для выполнения которых нужны различные операционные блоки, а если нет?

Один из путей дальнейшего повышения эффективности подобных систем лежит в области разработки *специальных компиляторов*, которые упаковы-

вают несколько простых команд в "очень длинное командное слово" (VLIW — аббревиатура от Very Long Instruction Word) таким образом, чтобы в одной "очень длинной команде" можно было использовать все существующие в процессоре операционные блоки. В этом случае командное слово соответствует набору функциональных устройств.

VLIW-архитектуру можно рассматривать как статическую суперскалярную, поскольку распараллеливание кода производится на этапе компиляции, а не динамически во время исполнения, т. е. в машинном коде VLIW присутствует явный параллелизм.

Одним из примеров воплощения идей VLIW может служить предложенная Intel в содружестве с HP концепция 64-разрядной архитектуры микропроцессора IA-64 (Intel 64-bit Architecture, 64-разрядная архитектура Intel). Для ее обозначения использована аббревиатура EPIC (Explicitly Parallel Instruction Computing, вычисления с явным параллелизмом команд).

Процессор, разработанный на базе этой концепции, отличающийся следующими особенностями:

- большое количество регистров: 128 64-разрядных регистров общего назначения (целочисленных), плюс 128 80-разрядных регистров арифметики плавающей запятой, плюс 64 1-разрядных предикатных регистра;
- масштабируемость архитектуры до большого количества функциональных устройств. Это свойство представители фирм Intel и HP называют "наследственно масштабируемым набором команд" (inherently scaleable instruction set);
- явный параллелизм в машинном коде: поиск зависимостей между командами производит не процессор, а компилятор;
- предикация (predication): команды из разных ветвей условного ветвления снабжаются предикатными полями (полями условий) и запускаются на выполнение параллельно;
- загрузка по предположению (speculative loading): данные из медленной основной памяти загружаются заранее.

Формат команды IA-64 включает код операции, три 7-разрядных поля операндов — 1 приемник и 2 источника (операндами могут быть только регистры), особые поля для вещественной и целой арифметики, 6-разрядное предикатное поле.

Команды IA-64 упаковываются (группируются) компилятором в "связку" длиною в 128 разрядов. Связка содержит 3 команды и шаблон, в котором указаны зависимости между командами в связке (можно ли с командой k_1 запустить параллельно k_2, или же k_2 должна выполниться только после k_1), а

также между другими связками (можно ли с командой k_3 из связки c_1 запустить параллельно команду k_4 из связки c_2).

Одна такая связка, состоящая из трех команд, соответствует набору из трех функциональных устройств процессора. Процессоры IA-64 могут содержать разное количество таких блоков, оставаясь при этом совместимыми по коду. Ведь благодаря тому, что в шаблоне указана зависимость и между связками, процессору с N одинаковыми блоками из трех функциональных устройств будет соответствовать командное слово из $N \times 3$ команд (N связок). Таким образом, обеспечивается *масштабируемость* IA-64.

Предикация — способ обработки условных ветвлений. Суть этого способа — компилятор указывает, что обе ветви выполняются на процессоре параллельно. Если в исходной программе встречается условное ветвление, то команды из разных ветвей помечаются различными предикатными регистрами (команды имеют для этого предикатные поля), далее они выполняются совместно, но их результаты не записываются, пока значения предикатных регистров не определены. Когда, наконец, вычисляется значение условия ветвления, предикатный регистр, соответствующий "правильной" ветви, устанавливается в 1, а другой — в 0. Перед записью результатов процессор будет проверять предикатное поле и записывать результаты только тех команд, предикатное поле которых содержит предикатный регистр, установленный в 1.

Загрузка по предположению — это механизм, который предназначен снизить простои процессора, связанные с ожиданием выполнения команд загрузки из относительно медленной основной памяти. Компилятор перемещает команды загрузки данных из памяти так, чтобы они выполнились как можно раньше. Следовательно, когда данные из памяти понадобятся какой-либо команде, процессор не будет простаивать. Перемещенные таким образом команды называются командами загрузки по предположению и помечаются особым образом. Непосредственно перед командой, использующей загружаемые по предположению данные, компилятор вставляет команду проверки предположения.

Выводы

Основная особенность EPIC — распараллеливанием потока команд занимается компилятор, а не процессор.

Достоинства данного подхода:

- упрощается архитектура процессора; вместо распараллеливающей логики на EPIC-процессоре можно разместить больше регистров, функциональных устройств;

- процессор не тратит время на анализ потока команд на предмет возможности их параллельного выполнения — эту работу уже выполнил компилятор;
- возможности процессора по анализу программы во время выполнения ограничены сравнительно небольшим участком программы, тогда как компилятор способен произвести анализ по всей программе;
- если некоторая программа должна запускаться многократно, выгоднее распараллелить ее один раз (при компиляции), а не каждый раз, когда она исполняется на процессоре.

Недостатки:

- компилятор производит статический анализ программы, раз и навсегда планируя вычисления. Однако даже при небольшом изменении начальных данных путь выполнения программы может сколь угодно сильно измениться;
- серьезно увеличивается сложность компиляторов. Значит, увеличится число ошибок в них, время компиляции;
- еще более увеличивается сложность отладки, т. к. отлаживать придется оптимизированный параллельный код;
- производительность EPIC будет всецело зависеть от качества компилятора;
- проблематичным пока видится преемственность программного обеспечения при переходе на новые поколения микропроцессоров (скомпилированный код очень сильно "привязан" к конкретной архитектуре процессора).

Тем не менее, представители Intel и HP называют EPIC концепцией следующего поколения и противопоставляют ее CISC и RISC. По мнению Intel, традиционные архитектуры имеют фундаментальные свойства, ограничивающие производительность.

Производители RISC-процессоров не разделяют подобного пессимизма. Кстати, в 1980-х годах, когда возникла концепция RISC, прозвучало много заявлений, что концепция CISC устарела, имеет фундаментальные свойства, ограничивающие производительность. Но процессоры, причисляемые к CISC (например, семейство *x*86), широко используются до сих пор, их производительность растет.

В действительности же, все эти аббревиатуры — CISC, RISC, VLIW, EPIC — обозначают только *идеализированные концепции*. Реальные современные микропроцессоры трудно подвести исключительно под какой-либо из перечисленных выше классов. Просто в наиболее совершенных современных

процессорах заложено большое число удачных идей, использующих многие рассмотренные здесь концепции.

7.6. Однокристальные микроЭВМ

При рассмотрении процессов эволюции современных процессоров и ЭВМ, прежде всего, обращают внимание на увеличение производительности системы (быстродействие процессора). Некоторые из путей повышения быстродействия были обсуждены в предыдущих разделах, другие, реализующие параллелизм на уровне процессов (мультипроцессоры, векторные, массивно-параллельные, компьютерные сети [11, 12], нейроматричные процессоры, и др.), выходят за рамки настоящей книги.

Однако всегда существовали и существуют задачи, для решения которых вовсе не требуется высокое быстродействие процессора и мощная система команд. На первый план здесь выступают другие характеристики: стоимость, надежность, малые габариты, способность работать в экстремальных климатических условиях, при значительных перепадах питающего напряжения и на фоне высокого уровня электромагнитных помех, с автономным питанием.

Для получения таких характеристик растущие возможности интегральной технологии можно использовать не для увеличения разрядности и вычислительной мощности процессора, а размещая на кристалле, наряду с простым (на первых порах — восьмиразрядным) процессором, все другие устройства, входящие в состав ЭВМ: регистры, различные типы памяти (на первых порах — небольшого объема), тактовый генератор, порты параллельного и последовательного обмена, различные внешние устройства (таймеры, АЦП и др.).

При этом получается полностью "самодостаточный" кристалл БИС (СБИС) *однокристальной микроЭВМ* (некоторые авторы используют термин *микроконтроллер*, учитывая, что основная сфера применения подобных изделий — управляющие системы, работающие в реальном времени).

В настоящее время многие фирмы (Motorola, Intel, MicroChip, Zilog и др.) выпускают широкую номенклатуру подобных однокристальных микроЭВМ (ОМЭВМ), отличающихся разрядностью, системой команд, типами и объемом памяти, составом и характеристиками внешних устройств. Большинство ОМЭВМ выпускается с 8-разрядным процессором, но на рынке присутствуют и 16- и даже 32-разрядные ОМЭВМ.

Базовые принципы архитектуры ОМЭВМ можно проиллюстрировать на примере 8-разрядных ОМЭВМ. Далее кратко отметим некоторые особенности этих контроллеров. Подробнее архитектура ряда ОМЭВМ и примеры их применения описаны в [4].

Контроллеры разных фирм, несмотря на кажущиеся различия, имеют много общих черт, определяющих тенденции развития современных ОМЭВМ малой и средней производительности. Попробуем отметить некоторые из них.

- Все без исключения контроллеры имеют *встроенные тактовые генераторы*; для запуска большинства из них используют одну из четырех возможных внешних цепей: источник внешних тактовых импульсов, кварцевый резонатор, LC-цепь, RC-цепь. Последние две можно использовать лишь в системах, где точностью временных привязок можно пренебречь. Большинство контроллеров имеют в своем составе динамические элементы памяти, что ограничивает допустимую тактовую частоту не только сверху, но и снизу (обычно — до 1 МГц). ОМЭВМ Motorola используют на кристалле только статические элементы памяти, что позволяет работать на произвольно низких системных тактовых частотах. Снижение тактовой частоты контроллера целесообразно при управлении инерционными объектами, если необходимо отслеживать достаточно длительные временные задержки (от долей секунды до десятков секунд).
- Процессоры большинства ОМЭВМ реализуют классическую ("интелловскую") систему команд, включающую одно- и двухадресные команды с операциями над ячейками памяти и регистрами, с использованием разнообразных способов адресации (прямая, регистровая, косвенно-регистровая, индексная, непосредственная). Предусмотрен широкий выбор команд передачи управления, в т. ч. вызовы подпрограмм. Во многих контроллерах реализовано умножение и деление. Процессоры семейств *MCS-51* и *MC68HC11* имеют развитую систему операций с битами.
- Память большинства ОМЭВМ организована по *гарвардской архитектуре*, предполагающей различные адресные пространства для памяти программ и памяти данных (исключение составляют лишь контроллеры фирмы Motorola, традиционно поддерживающие единое адресное пространство). Такое решение снижает риск потери управления при выполнении программы, но ограничивает возможности по распределению ресурсов памяти системы.
- На кристалле могут располагаться различные типы памяти: масочное или однократно программируемое ПЗУ, ППЗУ со стиранием ультрафиолетовым излучением, электрически стираемое ППЗУ (флэш-память), ОЗУ (регистры).
- Многие параллельные порты контроллеров допускают двунаправленный обмен, часто возможно независимое программирование линий порта на ввод или вывод. Допускается выбор типа выхода — обычный ТТЛ-вывод или вывод с открытым коллектором (стоком). Большинство линий портов

имеют одну или несколько альтернативных функций, выбор которых осуществляется программно.

- Важным элементом ОМЭВМ являются системы контроля времени, представленные различными счетчиками с управляемыми коэффициентами пересчета и возможностью выбора источника счетных импульсов: тактовый генератор — в режиме таймера и внешний вывод — в режиме счетчика внешних событий. Более мощные контроллеры имеют в своем составе таймерные системы, включающие несколько модулей сравнения, автозахвата, ШИМ (широтно-импульсная модуляция) и др.
- Последовательные каналы включаются обычно в старшие модели семейств. Предусматриваются либо универсальные синхронно-асинхронные приемопередатчики, программируемые на работу в определенном режиме, либо отдельные блоки SCI (UART) — асинхронный приемопередатчик и SPI — синхронный периферийный интерфейс, работающие независимо друг от друга.
- Средства работы с аналоговыми сигналами включают в себя компараторы, многоканальные 8-разрядные аналого-цифровые и реже — цифроаналоговые преобразователи.
- Подсистема прерываний включает несколько внешних радиальных входов и большое число внутренних прерываний, которые генерируются в системе контроля времени, АЦП, последовательных и параллельных каналах.

В современных микроконтроллерах предусматривается широкий набор специальных средств, повышающих надежность и эффективность функционирования систем управления. Прежде всего, это т. н. *сторожевой таймер* WDT (Watch-Dog Timer), который предотвращает аварийное зацикливание программы. В некоторых контроллерах предусматриваются специальные схемы, следящие за правильной работой тактового генератора. В системах с автономным питанием большое значение имеют средства энергосбережения. Большинство контроллеров программно можно переводить в специальные *режимы пониженного энергопотребления* (в состоянии ожидания) с остановкой основных подсистем (в т. ч. иногда и тактового генератора), но с сохранением контекста задачи. Выход из таких режимов возможен по разрешенному прерыванию или по сбросу (часто системный сброс реализуется как одно из прерываний).

16- и 32-разрядные ОМЭВМ построены обычно по модульному принципу. На внутрикристальный системный интерфейс могут подключаться процессоры различной вычислительной мощности, различные модули памяти, контроллеры параллельного и последовательного обмена, модули АЦП и ЦАП, таймерные сопроцессоры и сопроцессоры ввода/вывода. В зависимости от тре-

бований решаемой задачи пользователь может выбрать подходящую конфигурацию кристалла ОМЭВМ. В качестве примера коротко рассмотрим семейство 32-разрядных микроконтроллеров фирмы Motorola.

Отличительной особенностью ОМЭВМ фирмы Motorola является модульная технология построения многофункциональных устройств на одном кристалле. Определен стандарт внутрикристальной межмодульной шины IMB и множество наборов системных модулей, из которых собирается ОМЭВМ:

- *процессорные ядра* (CPU), включающие 16- и 32-разрядные микропроцессоры различной вычислительной мощности, но относящиеся к одному семейству;
- *системные интеграционные модули* (SIM), контролирующие внешнюю шину, запуск, инициализацию и конфигурацию микроконтроллера. Они включают в себя тактовый генератор, блок системной конфигурации и защиты, блок тестирования и интерфейс с внешней шиной. Модули отличаются друг от друга, главным образом, разрядностью шин адреса и данных;
- *модули памяти*, отличающиеся типом и объемом запоминающих устройств: ОЗУ, ПЗУ(включая однократно программируемое пользователем), ЭСППЗУ;
- *модули последовательных портов* включают различные варианты синхронных и асинхронных программируемых контроллеров последовательного обмена. Предусмотрены модули, содержащие несколько различных интерфейсов;
- *таймерные системы* представлены различными вариантами таймерных сопроцессоров (TPU). TPU способен независимо от процессора выполнять как простые, так и сложные таймерные функции, его можно считать отдельным специализированным микропроцессором, который осуществляет две основные операции — проверку на совпадение (от англ. *match* — сравнение) и сохранение значения счетчика-таймера в момент изменения состояния какого-либо входа (от англ. *capture* — захват) над одним операндом — временем. Выполнение любой из них называется *событием*. Обслуживание событий сопроцессором замещает обработку прерываний центральным процессором. С помощью двух основных операций TPU может реализовать значительный набор функций: счет внешних событий, захват по внешнему входу, сравнение временных интервалов, широтно-импульсную модуляцию, измерение периода входного сигнала, программируемую генерацию импульсов и многие другие, программируемые пользователем. TPU, естественно, имеет собственную систему команд, его программы хранятся на общей памяти системы или в специализированной памяти программ TPU;

- *системы аналогового ввода* реализованы на различных вариантах АЦП (ADC), отличающихся разрядностью получаемого кода и способом (а следовательно, и временем) аналого-цифрового преобразования.

На рис. 7.16 приведена структура 32-разрядного микроконтроллера MC68332.

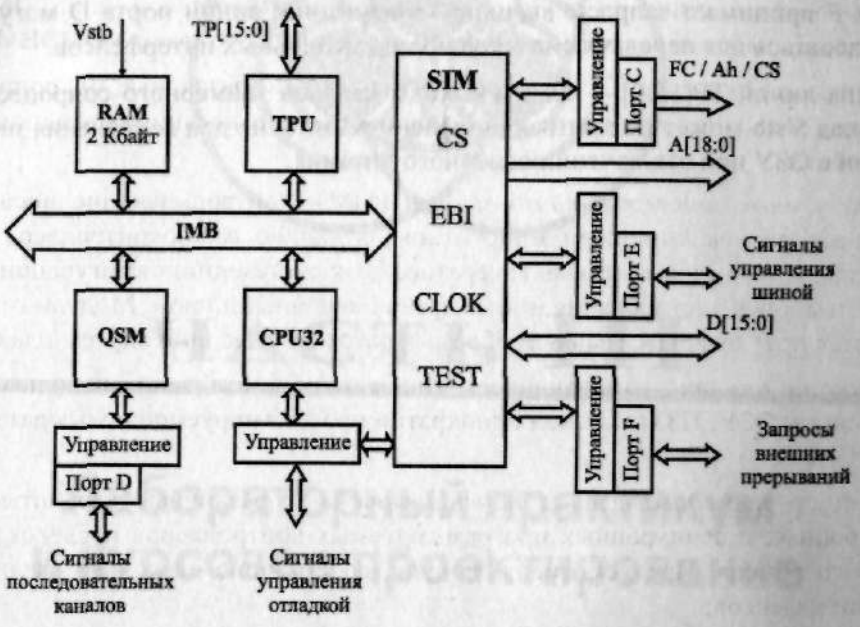


Рис. 7.16. Структура MC68332

Кристалл микроконтроллера содержит следующие модули:

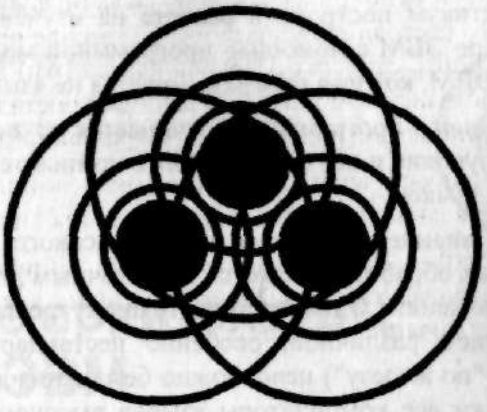
- CPU32 — 32-разрядный центральный процессор семейства MC68000;
- RAM — 2 Кбайт ОЗУ с независимым питанием для размещения программ и данных (напомним, что большинство изделий Motorola поддерживают архитектуру фон Неймана с единым адресным пространством программ и данных);
- QSM — подсистемы последовательного ввода/вывода, включающие универсальный асинхронный приемопередатчик (UART) и дуплексный синхронный последовательный интерфейс (SPI);
- TPU — шестнадцатиканальный таймерный сопроцессор;
- SIM — системный интеграционный модуль.

Внешняя память может при необходимости подключаться к контроллеру к шинам *адреса* A[18:0], *данных* D[15:0], *управления шиной*.

Линии порта C могут использоваться для выдачи функционального кода, идентифицирующего состояние процессора и адресное пространство текущего цикла шины (FC[2:0]), старших разрядов адреса (Ah) в режиме расширенного адресного пространства и сигналов выбора кристалла (CS), разрешающих работу периферийных устройств по запрограммированным адресам.

Порт F принимает запросы внешних прерываний, линии порта D могут использоваться для передачи сигналов последовательных интерфейсов.

Группа линий TP[15:0] — входы/выходы каналов таймерного сопроцессора. На вход Vstb может подаваться независимое питание для сохранения информации в ОЗУ при отключении основного питания.



ЧАСТЬ III

Лабораторный практикум и курсовое проектирование

Лабораторный практикум построен в расчете на изучение взаимодействия устройств в структуре ЭВМ с помощью программной модели некоторой абстрактной учебной ЭВМ, которая программируется на языке *ассемблера*.

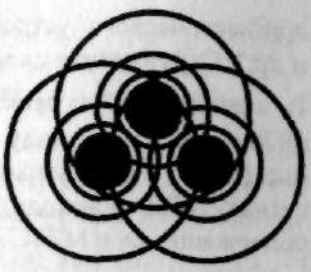
Часто путь современного программиста начинается со знакомства с языком (языками) высокого уровня и все его общение с компьютером проходит с использованием таких языков.

Во многих случаях знание операторов языка высокого уровня, структуры данных и способов их обработки является достаточным для создания различных полезных приложений. Однако по-настоящему решать проблемы, связанные с управлением различной, особенно нестандартной, аппаратурой (программирование "по железу") невозможно без знания ассемблера [14]. Не случайно практически все компиляторы языков высокого уровня содержат средства связи своих модулей с модулями на ассемблере либо поддерживают выход на ассемблерный уровень программирования.

Однако проводить начальное обучение программированию на низком уровне с рассмотрением механизмов взаимодействия устройств на реальном языке, например *x*86 на персональной ЭВМ, не всегда удобно. В этом случае между пользователем и аппаратурой ЭВМ присутствует операционная система (ОС), которая существенно ограничивает желания пользователя экспериментировать с аппаратными средствами. Для преодоления этих ограничений необходимо обладать глубокими знаниями как ОС, так и аппаратных средств ЭВМ.

Предлагаемая для использования программная модель учебной ЭВМ отражает все основные особенности систем команд и структур современных простых ЭВМ, включает в себя, помимо процессора и памяти, модели нескольких типичных внешних устройств. Модель позволяет изучить основы программирования на низком уровне, вопросы взаимодействия различных уровней памяти в составе ЭВМ и способы взаимодействия процессора с внешними устройствами.

ГЛАВА 8



Описание архитектуры учебной ЭВМ

Современные процессоры и операционные системы — не слишком благоприятная среда для начального этапа изучения архитектуры ЭВМ.

Одним из решений этой проблемы может быть создание программных моделей учебных ЭВМ, которые, с одной стороны, достаточно просты, чтобы обучаемый мог освоить базовые понятия архитектуры (система команд, командный цикл, способы адресации, уровни памяти, способы взаимодействия процессора с памятью и внешними устройствами), с другой стороны — архитектурные особенности модели должны соответствовать тенденциям развития современных ЭВМ.

Программная модель позволяет реализовать доступ к различным элементам ЭВМ, обеспечивая удобство и наглядность. С другой стороны, модель позволяет игнорировать те особенности работы реальной ЭВМ, которые на данном уровне рассмотрения не являются существенными.

Далее приводится описание программной модели учебной ЭВМ[1], предназначенной для начальных этапов изучения архитектуры (в т. ч. на младших курсах вуза и даже в школе). Именно этим объясняется использование в модели десятичной системы счисления для кодирования команд и представления данных.

8.1. Структура ЭВМ

Моделируемая ЭВМ включает процессор, оперативную (ОЗУ) и сверхоперативную память, устройство ввода (УВв) и устройство вывода (УВыв). Процессор, в свою очередь, состоит из центрального устройства управления (УУ),

[1] Программная модель учебной ЭВМ находится на компакт-диске, прилагаемом к книге.

арифметического устройства (АУ) и системных регистров (CR, PC, SP и др.). Структурная схема ЭВМ показана на рис. 8.1.

В ячейках ОЗУ хранятся команды и данные. Емкость ОЗУ составляет 1000 ячеек. По сигналу MWr выполняется запись содержимого регистра данных (MDR) в ячейку памяти с адресом, указанным в регистре адреса (MAR). По сигналу MRd происходит считывание — содержимое ячейки памяти с адресом, содержащимся в MAR, передается в MDR.

Сверхоперативная память с прямой адресацией содержит десять регистров общего назначения R0—R9. Доступ к ним осуществляется (аналогично доступу к ОЗУ) через регистры RAR и RDR.

АУ осуществляет выполнение одной из арифметических операций, определяемой кодом операции (COP), над содержимым аккумулятора (Acc) и регистра операнда (DR). Результат операции всегда помещается в Acc. При завершении выполнения операции АУ вырабатывает сигналы признаков результата: Z (равен 1, если результат равен нулю); S (равен 1, если результат отрицателен); OV (равен 1, если при выполнении операции произошло переполнение разрядной сетки). В случаях, когда эти условия не выполняются, соответствующие сигналы имеют нулевое значение.

В модели ЭВМ предусмотрены внешние устройства двух типов. Во-первых, это регистры IR и OR, которые могут обмениваться с аккумулятором с помощью безадресных команд IN (Acc := IR) и OUT (OR := Acc). Во-вторых, это набор моделей внешних устройств, которые могут подключаться к системе и взаимодействовать с ней в соответствии с заложенными в моделях алгоритмами. Каждое внешнее устройство имеет ряд программно-доступных регистров, может иметь собственный *обозреватель* (окно видимых элементов). Подробнее эти внешние устройства описаны в *разд. 8.6*.

УУ осуществляет выборку команд из ОЗУ в последовательности, определяемой естественным порядком выполнения команд (т. е. в порядке возрастания адресов команд в ОЗУ) или командами передачи управления; выборку из ОЗУ операндов, задаваемых адресами команды; инициирование выполнения операции, предписанной командой; останов или переход к выполнению следующей команды.

В качестве сверхоперативной памяти в модель включены регистры общего назначения (РОН), и может подключаться модель кэш-памяти.

В состав УУ ЭВМ входят:

- PC — счетчик адреса команды, содержащий адрес текущей команды;
- CR — регистр команды, содержащий код команды;
- RB — регистр базового адреса, содержащий базовый адрес;

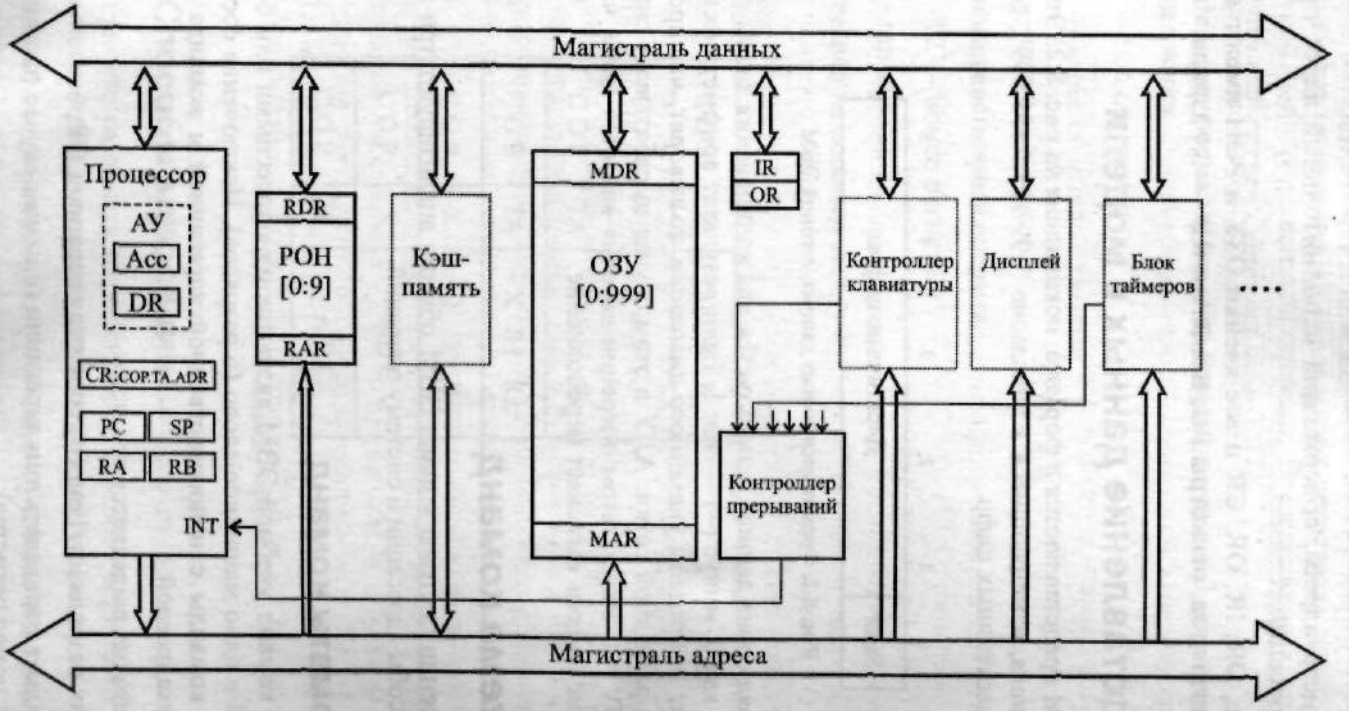


Рис. 8.1. Общая структура учебной ЭВМ

- SP — указатель стека, содержащий адрес верхушки стека;
- RA — регистр адреса, содержащий исполнительный адрес при косвенной адресации.

Регистры Acc, DR, IR, OR, CR и все ячейки ОЗУ и РОН имеют длину 6 десятичных разрядов, регистры PC, SP, RA и RB — 3 разряда.

8.2. Представление данных в модели

Данные в ЭВМ представляются в формате, показанном на рис. 8.2. Это целые десятичные числа, изменяющиеся в диапазоне *"–99 999…+99 999"*, содержащие знак и 5 десятичных цифр.

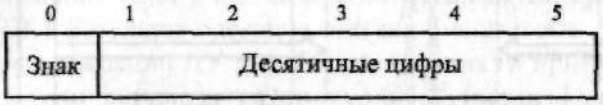


Рис. 8.2. Формат десятичных данных учебной ЭВМ

Старший разряд слова данных используется для кодирования знака: плюс (+) *изображается как 0*, минус (–) — как 1. *Если результат арифметической операции* выходит за пределы указанного диапазона, то говорят, что произошло переполнение разрядной сетки. АЛУ в этом случае вырабатывает сигнал переполнения OV = 1. Результатом операции деления является целая часть частного. Деление на ноль вызывает переполнение.

8.3. Система команд

При рассмотрении системы команд ЭВМ обычно анализируют три аспекта: *форматы, способы адресации и систему операций.*

8.3.1. Форматы команд

Большинство команд учебной ЭВМ являются одноадресными или безадресными, длиной в одно машинное слово (6 разрядов). Исключение составляют двухсловные команды с непосредственной адресацией и команда MOV, *являющаяся двухадресной.*

В форматах команд выделяется три поля:

- два старших разряда [0:1] определяют код операции COP;
- разряд 2 может определять тип адресации (в одном случае (формат 5а) он определяет номер регистра);

- разряды [3:5] могут определять прямой или косвенный адрес памяти, номер регистра (в команде MOV номера двух регистров), адрес перехода или короткий непосредственный операнд. В двухсловных командах непосредственный операнд занимает поле [6:11].

Полный список форматов команд показан на рис. 8.3, где приняты следующие обозначения:

- COP — код операции;
- ADR — адрес операнда в памяти;
- ADC — адрес перехода;
- I — непосредственный операнд;
- R, R1, R2 — номер регистра;
- TA — тип адресации;
- X — разряд не используется.

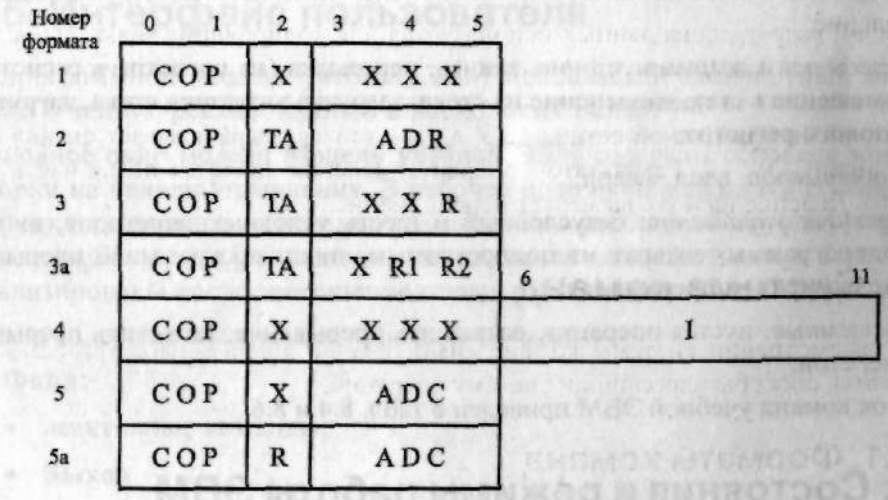


Рис. 8.3. Форматы команд учебной ЭВМ

8.3.2. Способы адресации

В ЭВМ принято различать пять основных способов адресации: *прямая*, *косвенная*, *непосредственная*, *относительная*, *безадресная*.

Каждый способ имеет разновидности. В модели учебной ЭВМ реализованы семь способов адресации, приведенные в табл. 8.1.

Таблица 8.1. Адресация в командах учебной ЭВМ

Код ТА	Тип адресации	Исполнительный адрес
0	Прямая (регистровая)	ADR (R)
1	Непосредственная	—
2	Косвенная	ОЗУ(ADR)[3:5]
3	Относительная	ADR + RB
4	Косвенно-регистровая	РОН(R)[3:5]
5	Индексная с постинкрементом	РОН(R)[3:5], R:= R + 1
6	Индексная с преддекрементом	R:= R – 1, РОН(R)[3:5]

8.3.3. Система операций

Система команд учебной ЭВМ включает команды следующих классов:

- *арифметико-логические* и *специальные*: сложение, вычитание, умножение, деление;
- *пересылки* и *загрузки*: чтение, запись, пересылка (из регистра в регистр), помещение в стек, извлечение из стека, загрузка указателя стека, загрузка базового регистра;
- *ввода/вывода*: ввод, вывод;
- *передачи управления*: безусловный и шесть условных переходов, вызов подпрограммы, возврат из подпрограммы, цикл, программное прерывание, возврат из прерывания;
- *системные*: пустая операция, разрешить прерывание, запретить прерывание, стоп.

Список команд учебной ЭВМ приведен в табл. 8.4 и 8.6.

8.4. Состояния и режимы работы ЭВМ

Ядром УУ ЭВМ является управляющий автомат (УА), вырабатывающий сигналы управления, которые инициируют работу АЛУ, РОН, ОЗУ и УВВ, передачу информации между регистрами устройств ЭВМ и действия над содержимым регистров УУ.

ЭВМ может находиться в одном из двух состояний: **Останов** и **Работа**.

В состояние **Работа** ЭВМ переходит по действию команд **Пуск** или **Шаг**. Команда **Пуск** запускает выполнение программы, представляющую собой последовательность команд, записанных в ОЗУ, в автоматическом режиме до

команды HLT или точки останова. Программа выполняется по командам, начиная с ячейки ОЗУ, на которую указывает PC, причем изменение состояний объектов модели отображается в окнах обозревателей.

В состояние **Останов** ЭВМ переходит по действию команды **Стоп** или автоматически в зависимости от установленного режима работы.

Команда **Шаг**, в зависимости от установленного режима работы, запускает выполнение одной команды или одной микрокоманды (если установлен **Режим микрокоманд**), после чего переходит в состояние **Останов**.

В состоянии **Останов** допускается просмотр и модификация объектов модели: регистров процессора и РОН, ячеек ОЗУ, устройств ввода/вывода. В процессе модификации ячеек ОЗУ и РОН можно вводить данные для программы, в ячейки ОЗУ — программу в кодах. Кроме того, в режиме **Останов** можно менять параметры модели и режимы ее работы, вводить и/или редактировать программу в мнемокодах, ассемблировать мнемокоды, выполнять стандартные операции с файлами.

8.5. Интерфейс пользователя

В программной модели учебной ЭВМ использован стандартный интерфейс Windows, реализованный в нескольких окнах.

Основное окно модели **Модель учебной ЭВМ** содержит основное меню и кнопки на панели управления. В рабочее поле окна выводятся сообщения о функционировании системы в целом. Эти сообщения группируются в файле logfile.txt (по умолчанию), сохраняются на диске и могут быть проанализированы после завершения сеанса работы с моделью.

Меню содержит следующие пункты и команды:

- **Файл**:
  - неактивные команды;
  - **Выход**.
- **Вид**:
  - **Показать все**;
  - **Скрыть все**;
  - **Процессор**;
  - **Микрокомандный уровень**;
  - **Память**;
  - **Кэш-память**;

- **Программа**;
- **Текст программы**.

❒ **Внешние устройства**:

- **Менеджер ВУ**;
- окна подключенных ВУ;

❒ **Работа**:

- **Пуск**;
- **Стоп**;
- **Шаг**;
- **Режим микрокоманд**;
- **Кэш-память**;
- **Настройки**.

Команды меню **Вид** открывают окна соответствующих обозревателей, описанные далее. Менеджер внешних устройств позволяет подключать/отключать внешние устройства, предусмотренные в системе. Команда вызова менеджера внешних устройств выполняется при нажатии кнопки на панели инструментов. Подробнее о внешних устройствах и их обозревателях смотрите в *разд. 8.6*.

Команды меню **Работа** позволяют запустить программу в автоматическом (команда **Пуск**) или шаговом (команда **Шаг**) режиме, остановить выполнение программы в модели процессора (команда **Стоп**). Эти команды могут выполняться при нажатии соответствующих одноименных кнопок на панели инструментов основного окна.

Команда **Режим микрокоманд** включает/выключает микрокомандный режим работы процессора, а команда **Кэш-память** подключает/отключает в системе модель этого устройства.

Команда **Настройки** открывает диалоговое окно **Параметры системы**, позволяющее установить задержку реализации командного цикла (при выполнении программы в автоматическом режиме), а так же установить параметры файла logfile.txt, формируемого системой и записываемого на диск.

8.5.1. Окна основных обозревателей системы

Окно *Процессор*

Окно **Процессор** (рис. 8.4) обеспечивает доступ ко всем регистрам и флагам процессора.

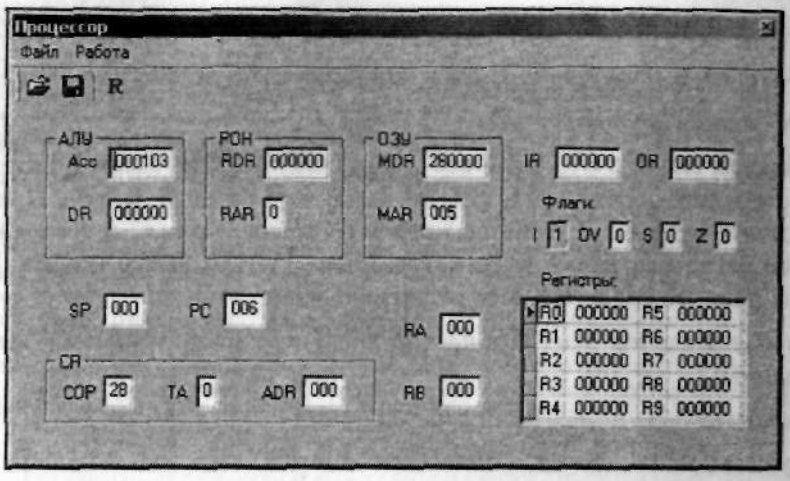


Рис. 8.4. Окно **Процессор**

- Программно-доступные регистры и флаги:
  - Acc — аккумулятор;
  - PC — счетчик адреса команды, содержащий адрес текущей команды;
  - SP — указатель стека, содержащий адрес верхушки стека;
  - RB — регистр базового адреса, содержащий базовый адрес;
  - RA — регистр адреса, содержащий исполнительный адрес при косвенной адресации;
  - IR — входной регистр;
  - OR — выходной регистр;
  - I — флаг разрешения прерываний.
- Системные регистры и флаги:
  - DR — регистр данных АЛУ, содержащий второй операнд;
  - MDR — регистр данных ОЗУ;
  - MAR — регистр адреса ОЗУ;
  - RDR — регистр данных блока РОН;
  - RAR — регистр адреса блока РОН;
  - CR — регистр команд, содержащий поля:
    - COP — код операции;
    - TA — тип адресации;
    - ADR — адрес или непосредственный операнд;

- Z — флаг нулевого значения Acc;
- S — флаг отрицательного значения Acc;
- OV — флаг переполнения.

Регистры Acc, DR, IR, OR, CR и все ячейки ОЗУ и РОН имеют длину 6 десятичных разрядов, регистры PC, SP, RA и RB — 3 разряда. В окне **Процессор** отражаются текущие значения регистров и флагов, причем в состоянии **Останов** все регистры, включая регистры блока РОН, и флаги (кроме флага I) доступны для непосредственного редактирования.

Элементы управления окна **Процессор** включают меню и кнопки, вызывающие команды:

- **Сохранить**;
- **Загрузить**;
- **Reset**;
- **Reset R0-R9** (только команда меню **Работа**).

Команды **Сохранить**, **Загрузить** позволяют сохранить текущее значение регистров и флагов процессора в файле и восстановить состояние процессора из файла. Команда **Reset** и кнопка **R** устанавливают все регистры (в т. ч. блок РОН) в начальное (нулевое) значение. Содержимое ячеек памяти при этом не меняется. Выполняемая лишь из меню **Работа** команда **Reset R0-R9** очищает только регистры блока РОН.

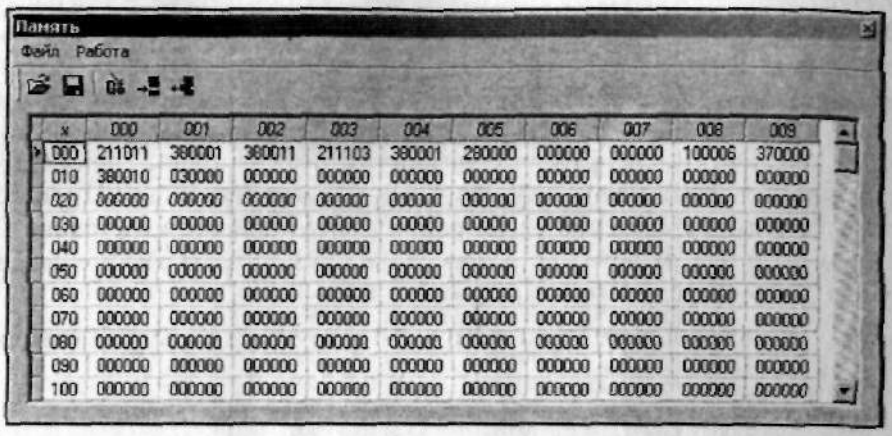
Окно *Память*

Окно **Память** (рис. 8.5) отражает текущее состояние ячеек ОЗУ. В этом окне допускается редактирование содержимого ячеек, кроме того, предусмотрена возможность выполнения (через меню или с помощью кнопок панели инструментов) пяти команд: **Сохранить**, **Загрузить**, **Перейти к**, **Вставить**, **Убрать**.

Команды **Сохранить**, **Загрузить** во всех окнах, где они предусмотрены, работают одинаково — сохраняют в файле текущее состояние объекта (в данном случае памяти) и восстанавливают это состояние из выбранного файла, причем файл в каждом окне записывается по умолчанию с характерным для этого окна расширением.

Команда **Перейти к** открывает диалоговое окно, позволяющее перейти на заданную ячейку ОЗУ.

Команда **Убрать** открывает диалог, в котором указывается диапазон ячеек с m по n. Содержимое ячеек в этом диапазоне теряется, а содержимое ячеек $[(n+1):999]$ перемещается в соседние ячейки с меньшими адресами. Освободившиеся ячейки с адресами 999, 998, ... заполняются нулями.



x	000	001	002	003	004	005	006	007	008	009
000	211011	380001	380011	211103	380001	280000	000000	000000	100006	370000
010	380010	030000	000000	000000	000000	000000	000000	000000	000000	000000
020	000000	000000	000000	000000	000000	000000	000000	000000	000000	000000
030	000000	000000	000000	000000	000000	000000	000000	000000	000000	000000
040	000000	000000	000000	000000	000000	000000	000000	000000	000000	000000
050	000000	000000	000000	000000	000000	000000	000000	000000	000000	000000
060	000000	000000	000000	000000	000000	000000	000000	000000	000000	000000
070	000000	000000	000000	000000	000000	000000	000000	000000	000000	000000
080	000000	000000	000000	000000	000000	000000	000000	000000	000000	000000
090	000000	000000	000000	000000	000000	000000	000000	000000	000000	000000
100	000000	000000	000000	000000	000000	000000	000000	000000	000000	000000

Рис. 8.5. Окно **Память**

Команда **Вставить**, позволяющая задать номера ячеек, перемещает содержимое всех ячеек, начиная от m-й на $n-m$ позиций в направлении больших адресов, ячейки заданного диапазона $[m:n]$ заполняются нулями, а содержимое последних ячеек памяти теряется.

Окно *Текст программы*

Окно ***Текст программы*** (рис. 8.6) содержит стандартное поле текстового редактора, в котором можно редактировать тексты, загружать в него текстовые файлы и сохранять подготовленный текст в виде файла.

Команды меню ***Файл:***

- **Новая** — открывает новый сеанс редактирования;
- **Загрузить** — открывает стандартный диалог загрузки файла в окно редактора;
- **Сохранить** — сохраняет файл под текущим именем;
- **Сохранить как** — открывает стандартный диалог сохранения файла;
- **Вставить** — позволяет вставить выбранный файл в позицию курсора.

Все перечисленные команды, кроме последней, дублированы кнопками на панели инструментов окна. На той же панели присутствует еще одна кнопка — ***Компилировать***, которая запускает процедуру ассемблирования текста в поле редактора.

Ту же процедуру можно запустить из меню **Работа**. Команда **Адрес вставки** позволяет задать адрес ячейки ОЗУ, начиная с которой программа будет размещаться в памяти. По умолчанию этот адрес принят равным 0.

Ниже области редактирования в строку состояния выводится позиция текущей строки редактора — номер строки, в которой находится курсор.

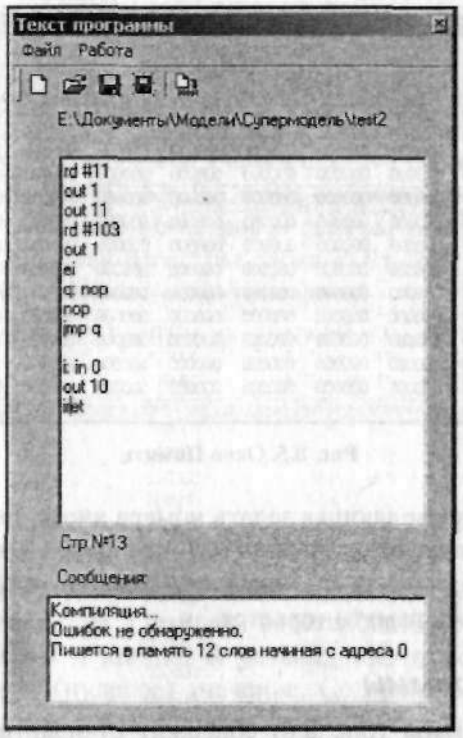


Рис. 8.6. Окно **Текст программы**

В случае обнаружения синтаксических ошибок в тексте программы диагностические сообщения процесса компиляции выводятся в окно сообщений и запись в память кодов (даже безошибочного начального фрагмента программы) не производится.

После исправления ошибок и повторной компиляции выдается сообщение об отсутствии ошибок, о расположении и размере области памяти, занятой под ассемблированную программу.

Набор текста программы производится по стандартным правилам языка ассемблера. В каждой строке может содержаться метка, одна команда и комментарий. Метка отделяется от команды двоеточием, символы после знака "точка с запятой" до конца строки игнорируются компилятором и могут рассматриваться как комментарии. Строка может начинаться с ; и, следовательно, содержать только комментарии.

Окно *Программа*

Окно **Программа** (рис. 8.7) отображает таблицу, имеющую 300 строк и 4 столбца. Каждая строка таблицы соответствует дизассемблированной ячей-

ке ОЗУ. Второй столбец содержит адрес ячейки ОЗУ, третий — дизассемблированный мнемокод, четвертый — машинный код команды. В первом столбце может помещаться указатель --> на текущую команду (текущее значение PC) и точка останова — красная заливка ячейки.

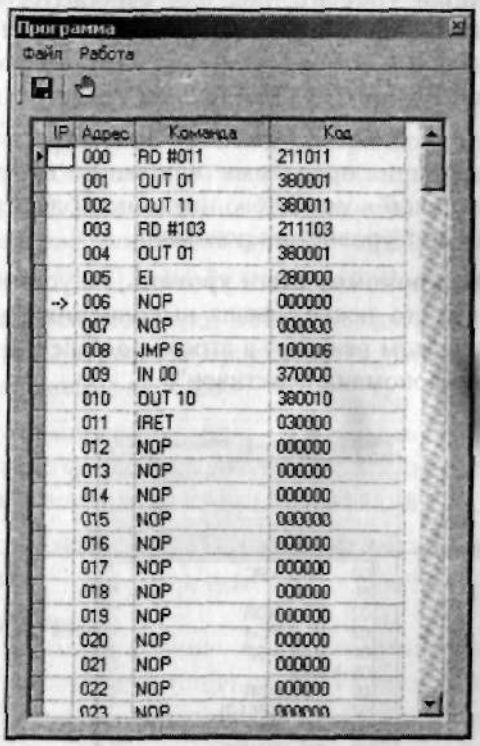


Рис. 8.7. Окно **Программа**

Окно **Программа** позволяет наблюдать процесс прохождения программы. В этом окне ничего нельзя редактировать. Органы управления окна позволяют сохранить содержимое окна в виде текстового файла, выбрать начальный адрес области ОЗУ, которая будет дизассемблироваться (размер области постоянный — 300 ячеек), а также установить/снять точку останова. Последнее можно проделать тремя способами: командой **Точка останова** из меню **Работа**, кнопкой на панели инструментов или двойным щелчком мыши в первой ячейке соответствующей строки. Характерно, что прочитать в это окно ничего нельзя. Сохраненный текстовый asm-файл можно загрузить в окно **Текст программы**, ассемблировать его и тогда дизассемблированное значение заданной области памяти автоматически появится в окне **Программа**. Такую процедуру удобно использовать, если программа изначально пишется или редактируется непосредственно в памяти в машинных кодах.

Начальный адрес области дизассемблирования задается в диалоге командой **Начальный адрес** меню **Работа**.

Окно *Микрокомандный уровень*

Окно **Микрокомандный уровень** (рис. 8.8) используется только в режиме микрокоманд, который устанавливается командой **Режим микрокоманд** меню **Работа**. В это окно выводится мнемокод выполняемой команды, список микрокоманд, ее реализующих, и указатель на текущую выполняемую микрокоманду.

Шаговый режим выполнения программы или запуск программы в автоматическом режиме с задержкой командного цикла позволяет наблюдать процесс выполнения программы на уровне микрокоманд.

Если открыть окно **Микрокомандный уровень**, не установив режим микрокоманд в меню **Работа**, то после начала выполнения программы в режиме **Шаг** (или в автоматическом режиме) в строке сообщений окна будет выдано сообщение "Режим микрокоманд неактивен".

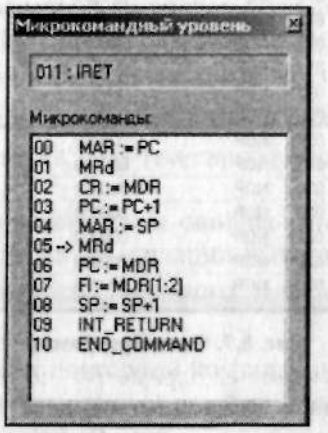


Рис. 8.8. Окно **Микрокомандный уровень**

Окно *Кэш-память*

Окно **Кэш-память** используется в режиме с подключенной кэш-памятью. Подробнее смотрите об этом режиме в *разд. 8.8*.

8.6. Внешние устройства

Модели внешних устройств (ВУ), используемые в описываемой системе, реализованы по единому принципу. С точки зрения процессора они представляют

собой ряд программно-доступных регистров, лежащих в адресном пространстве ввода/вывода. Размер регистров ВУ совпадает с размером ячеек памяти и регистров данных процессора — шесть десятичных разрядов.

Доступ к регистрам ВУ осуществляется по командам `IN aa`, `OUT aa`, где `aa` — двухразрядный десятичный адрес регистра ВУ. Таким образом, общий объем адресного пространства ввода/вывода составляет 100 адресов. Следует помнить, что адресные пространства памяти и ввода/вывода в этой модели разделены.

Разные ВУ содержат различное число программно-доступных регистров, каждому из которых соответствует свой адрес, причем нумерация адресов всех ВУ начинается с 0. При создании ВУ ему ставится в соответствие *базовый адрес* в пространстве ввода/вывода, и все адреса его регистров становятся *смещениями* относительно этого базового адреса.

Если в системе создаются несколько ВУ, то их базовые адреса следует выбирать с учетом величины адресного пространства, занимаемого этими устройствами, исключая наложение адресов.

Если ВУ способно формировать запрос на прерывание, то при создании ему ставится в соответствие *вектор прерывания* — десятичное число. Разным ВУ должны назначаться различные векторы прерываний.

Программная модель учебной ЭВМ комплектуется набором внешних устройств, включающим:

- контроллер клавиатуры;
- дисплей;
- блок таймеров;
- тоногенератор,

которым по умолчанию присвоены параметры, перечисленные в табл. 8.2.

Таблица 8.2. Параметры внешних устройств

Внешнее устройство	Базовый адрес	Адреса регистров	Вектор прерывания
Контроллер клавиатуры	0	0, 1, 2	0
Дисплей	10	0, 1, 2, 3	Нет
Блок таймеров	20	0, 1, 2, 3, 4, 5, 6	2
Тоногенератор	30	0, 1	Нет

При создании устройств пользователь может изменить назначенные по умолчанию базовый адрес и вектор прерывания.

В описываемой версии системы не предусмотрена возможность подключения в систему нескольких одинаковых устройств.

Большинство внешних устройств содержит регистры *управления* CR и *состояния* SR, причем обычно регистры CR доступны только по записи, а SR — по чтению.

Регистр CR содержит флаги и поля, определяющие режимы работы ВУ, а SR — флаги, отражающие текущее состояние ВУ. Флаги SR устанавливаются аппаратно, но сбрасываются программно (или по внешнему сигналу). Поля и флаги CR устанавливаются и сбрасываются программно при записи кода данных в регистр CR или специальными командами.

Контроллер ВУ интерпретирует код, записываемый по адресу CR как команду, если третий разряд этого кода равен 1, или как записываемые в CR данные, если третий разряд равен 0. В случае получения командного слова запись в регистр CR не производится, а пятый разряд слова рассматривается как код операции.

8.6.1. Контроллер клавиатуры

Контроллер клавиатуры (рис. 8.9) представляет собой модель внешнего устройства, принимающего ASCII-коды[1] от клавиатуры ПЭВМ.

Символы помещаются последовательно в *буфер символов*, размер которого установлен равным 50 символам, и отображаются в окне обозревателя (рис. 8.10).

В состав контроллера клавиатуры входят три программно-доступных регистра:

- DR (адрес 0) — регистр данных;
- CR (адрес 1) — регистр управления, определяет режимы работы контроллера и содержит следующие флаги:
  - E — флаг разрешения приема кодов в буфер;
  - I — флаг разрешения прерывания;
  - S — флаг режима посимвольного ввода.
- SR (адрес 2) — регистр состояния, содержит два флага:
  - Err — флаг ошибки;
  - Rd — флаг готовности.

[1] Сокр. от American Standard Code for Information Interchange — американский стандартный код обмена информацией.

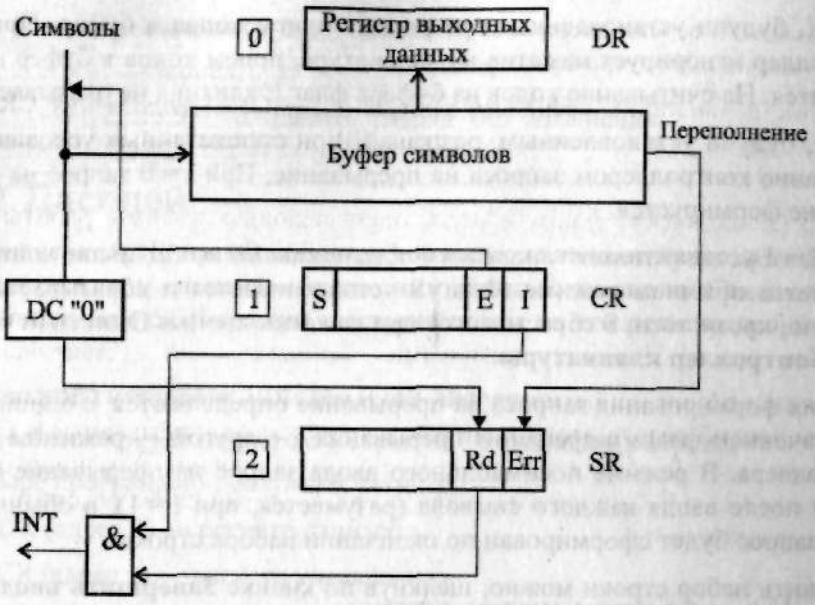


Рис. 8.9. Контроллер клавиатуры

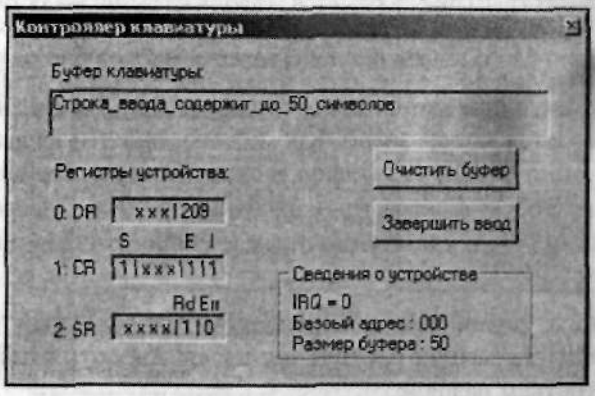


Рис. 8.10. Окно обозревателя контроллера клавиатуры

Регистр данных DR доступен только для чтения, через него считываются ASCII-коды из буфера, причем порядок чтения кодов из буфера соответствует порядку их записи в буфер — каждое чтение по адресу 0 автоматически перемещает указатель чтения буфера. В каждый момент времени DR содержит код символа по адресу указателя чтения буфера.

Флаги *регистра управления CR* устанавливаются и сбрасываются программно.

Флаг E, будучи установленным, разрешает прием кодов в буфер. При E = 0 контроллер игнорирует нажатие на клавиатуре, прием кодов в буфер не производится. На считывание кодов из буфера флаг E влияния не оказывает.

Флаг I, будучи установленным, разрешает при определенных условиях формирование контроллером запроса на прерывание. При I = 0 запрос на прерывание не формируется.

Флаг S = 1 устанавливает т. н. *режим посимвольного ввода*, иначе контроллер работает в обычном режиме. Флаг S устанавливается и сбрасывается программно, кроме того, S сбрасывается при нажатии кнопки **Очистить буфер** в окне **Контроллер клавиатуры**.

Условия формирования запроса на прерывание определяются, с одной стороны, значением флага разрешения прерывания I, с другой — режимом работы контроллера. В режиме посимвольного ввода запрос на прерывание формируется после ввода каждого символа (разумеется, при I = 1), в обычном режиме запрос будет сформирован по окончании набора строки.

Завершить набор строки можно, щелкнув по кнопке **Завершить ввод** в окне **Контроллер клавиатуры** (см. рис. 8.10). При этом устанавливается флаг готовности Rd (от англ. *ready*) в регистре состояния SR. Флаг ошибки Err (от англ. *error*) в том же регистре устанавливается при попытке ввода в буфер 51-го символа. Ввод 51-го и всех последующих символов блокируется.

Сброс флага Rd осуществляется автоматически при чтении из регистра DR, флаг Err сбрасывается программно. Кроме того, оба эти флага сбрасываются при нажатии кнопки **Очистить буфер** в окне **Контроллер клавиатуры**; одновременно со сбросом флагов производится очистка буфера — весь буфер заполняется кодами 00h, и указатели записи и чтения устанавливаются на начало буфера.

Для программного управления контроллером предусмотрен ряд командных слов. Все команды выполняются при записи по адресу регистра управления CR кодов с 1 в третьем разряде.

Контроллер клавиатуры интерпретирует следующие командные слова:

- xxx101 — очистить буфер (действие команды эквивалентно нажатию кнопки **Очистить буфер**);
- xxx102 — сбросить флаг Err в регистре SR;
- xxx103 — установить флаг S в регистре CR;
- xxx104 — сбросить флаг S в регистре CR.

Если по адресу 1 произвести запись числа xxx0*nn*, то произойдет изменение 4-го и 5-го разрядов регистра CR по следующему правилу:

$$n = \begin{cases} 0 — \text{записать } 0; \\ 1 — \text{записать } 1; \\ 2, \ldots, 9 — \text{сохранить разряд без изменения.} \end{cases} \quad (8.1)$$

8.6.2. Дисплей

Дисплей (рис. 8.11) представляет собой модель внешнего устройства, реализующую функции символьного дисплея. Дисплей может отображать символы, задаваемые ASCII-кодами, поступающими на его регистр данных. Дисплей включает:

- видеопамять объемом 128 слов (ОЗУ дисплея);
- символьный экран размером 8 строк по 16 символов в строке;
- четыре программно-доступных регистра:
  - DR (адрес 0) — регистр данных;
  - CR (адрес 1) — регистр управления;
  - SR (адрес 2) — регистр состояния;
  - AR (адрес 3) — регистр адреса.

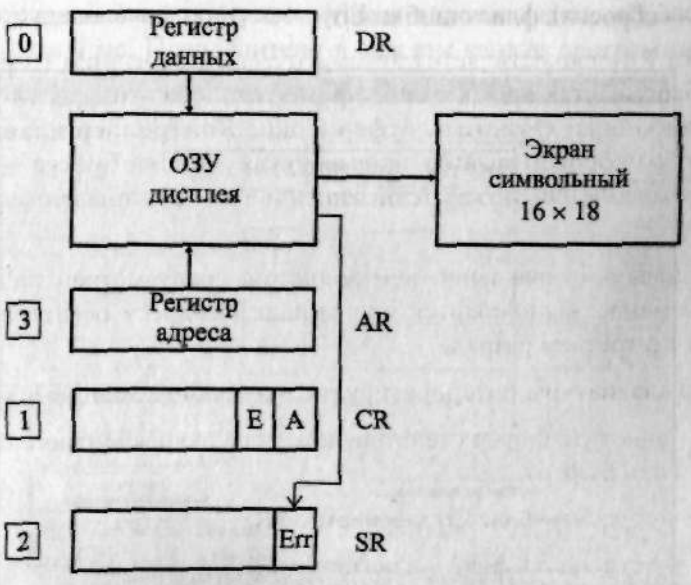


Рис. 8.11. Контроллер дисплея

Через *регистры адреса* AR и *данных* DR по записи и чтению осуществляется доступ к ячейкам видеопамяти. При обращении к регистру DR по записи со-

держимое аккумулятора записывается в DR и в ячейку видеопамяти, адрес которой установлен в регистре AR.

Регистр управления CR доступен только по записи и содержит в 4-м и 5-м разрядах соответственно два флага:

- E — флаг разрешения работы дисплея; при E = 0 запись в регистры AR и DR блокируется;
- A — флаг автоинкремента адреса; при A = 1 содержимое AR автоматически увеличивается на 1 после любого обращения к регистру DR — по записи или чтению.

Изменить значения этих флагов можно, если записать по адресу CR (по умолчанию — 11) код xxx0*nn*, при этом изменение 4-го и 5-го разрядов регистра CR произойдет согласно выражению (8.1).

Для программного управления дисплеем предусмотрены две команды, коды которых должны записываться по адресу регистра CR, причем в третьем разряде командных слов обязательно должна быть 1:

- xxx101 — очистить дисплей (действие команды эквивалентно нажатию кнопки **Очистить** в окне **Дисплей**), при этом очищается видеопамять (в каждую ячейку записывается код пробела — 032), устанавливается в 000 регистр адреса AR и сбрасываются флаги ошибки Err и автоинкремента A;
- xxx102 — сбросить флаг ошибки Err.

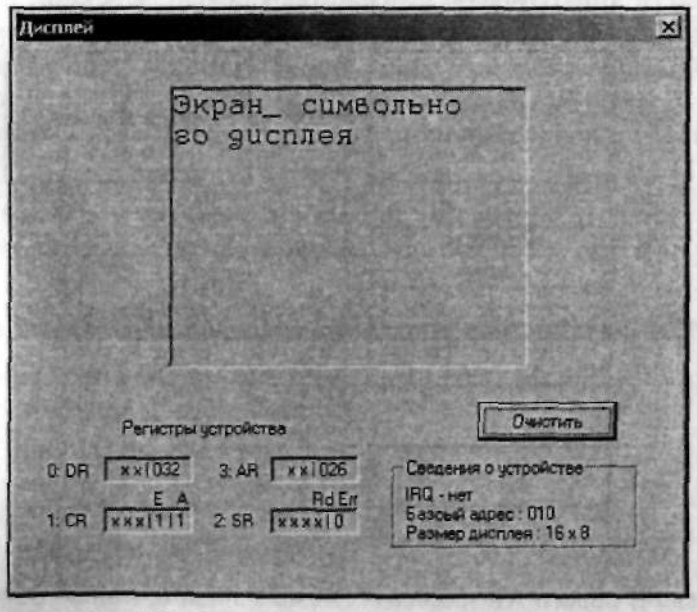


Рис. 8.12. Окно обозревателя контроллера дисплея

Регистр состояния SR доступен только по чтению и содержит единственный флаг (в пятом разряде) ошибки Err. Этот флаг устанавливается аппаратно при попытке записать в регистр адреса число, большее 127, причем как в режиме прямой записи в AR, так и в режиме автоинкремента после обращения по адресу 127. Сбрасывается флаг Err программно или при нажатии кнопки **Очистить** в окне **Дисплей** (рис. 8.12).

8.6.3. Блок таймеров

Блок таймеров (рис. 8.13) включает в себя три однотипных канала, каждый из которых содержит:

- пятиразрядный десятичный реверсивный счетчик T, на вход которого поступают метки времени (таймер);
- программируемый предделитель D;
- регистр управления таймером CTR;
- флаг переполнения таймера FT.

Регистры таймеров T доступны по записи и чтению (адреса 1, 3, 5 соответственно для T1, T2, T3). Программа в любой момент может считать текущее содержимое таймера или записать в него новое значение.

На входы предделителей поступает общие для всех каналов метки времени CLK с периодом 1 мс. Предделители в каждом канале программируются независимо, поэтому таймеры могут работать с различной частотой.

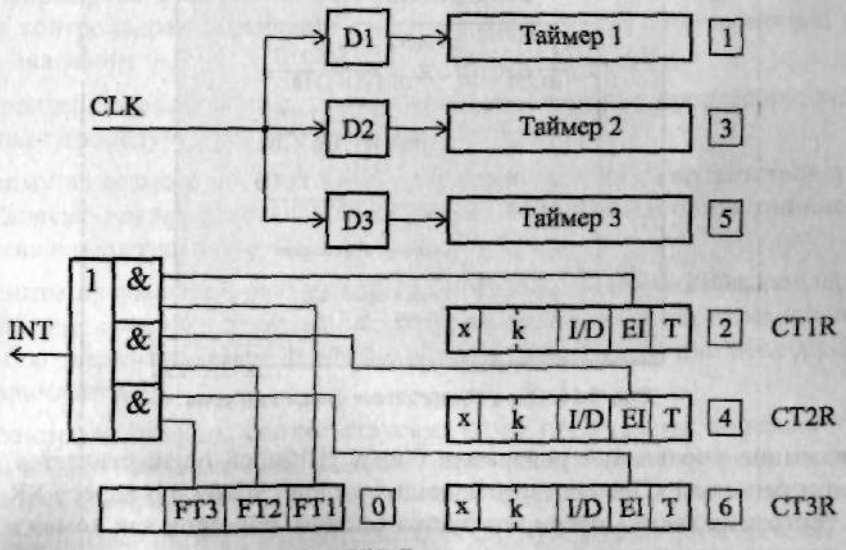


Рис. 8.13. Блок таймеров

Регистры управления CTR доступны по записи и чтению (адреса 2, 4, 6) и содержат следующие поля:

- T (разряд 5) — флаг включения таймера;
- EI (разряд 4) — флаг разрешения формирования запроса на прерывание при переполнении таймера;
- I/D (разряд 3) — направление счета (инкремент/декремент), при I/D = 0 таймер работает на сложение, при I/D = 1 — на вычитание;
- *k* (разряды [1:2]) — коэффициент деления предделителя (от 1 до 99).

Флаги переполнения таймеров собраны в один регистр — доступный только по чтению регистр состояния SR, имеющий адрес 0. Разряды регистра (5, 4 и 3 для T1, T2, T3 соответственно) устанавливаются в 1 при переполнении соответствующего таймера. Для таймера, работающего на сложение, переполнение наступает при переходе его состояния из 99 999 в 0, для вычитающего таймера — переход из 0 в 99 999.

В окне обозревателя (рис. 8.14) предусмотрена кнопка **Сброс**, нажатие которой сбрасывает в 0 все регистры блока таймеров, кроме CTR, которые устанавливаются в состояние 001000. Таким образом, все три таймера обнуляются, переключаются в режим инкремента, прекращается счет, запрещаются прерывания, сбрасываются флаги переполнения и устанавливаются коэффициенты деления предделителей равными 01.

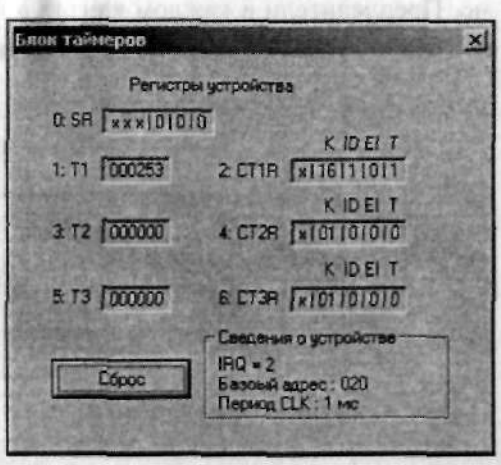


Рис. 8.14. Окно обозревателя блока таймеров

Программное управление режимами блока таймеров осуществляется путем записи в регистры CTR соответствующих кодов. Запись по адресу SR числа с 1 в третьем разряде интерпретируется блоком таймеров как команда, причем младшие разряды этого числа определяют код команды:

- xxx100 — общий сброс (эквивалентна нажатию кнопки **Сброс** в окне обозревателя);
- xxx101 — сброс флага переполнения таймера FT1;
- xxx102 — сброс флага переполнения таймера FT2;
- xxx103 — сброс флага переполнения таймера FT3.

8.6.4. Тоногенератор

Модель этого простого внешнего устройства не имеет собственного обозревателя, содержит всего два регистра, доступных только для записи:

- FR (адрес 0) — регистр частоты звучания (Гц):
- LR (адрес 1) — регистр длительности звучания (мс).

По умолчанию базовый адрес тоногенератора — 30. Сначала следует записать в FR требуемую частоту тона в герцах, затем в LR — длительность звучания в миллисекундах. Запись числа по адресу регистра LR одновременно является командой на начало звучания.

8.7. Подсистема прерываний

В модели учебной ЭВМ предусмотрен механизм векторных внешних прерываний. Внешние устройства формируют запросы на прерывания, которые поступают на входы *контроллера прерываний*. При подключении ВУ, способного формировать запрос на прерывание, ему ставится в соответствие номер входа контроллера прерываний — вектор прерывания, принимающий значение в диапазоне 0—9.

Контроллер передает вектор, соответствующий запросу, процессору, который *начинает процедуру обслуживания прерывания*.

Каждому из возможных в системе прерываний должен соответствовать т. н. *обработчик прерывания* — подпрограмма, вызываемая при возникновении события конкретного прерывания.

Механизм прерываний, реализованный в модели учебной ЭВМ, поддерживает *таблицу векторов прерываний*, которая создается в оперативной памяти *моделью операционной системы* (если она используется) или непосредственно пользователем.

Номер строки таблицы соответствует вектору прерывания, а элемент таблицы — ячейка памяти, в трех младших разрядах которой размещается начальный адрес подпрограммы, обслуживающей прерывание с этим вектором.

Таблица прерываний в рассматриваемой модели жестко фиксирована — она занимает ячейки памяти с адресами 100—109. Таким образом, адрес обработ-

чика с вектором 0 должен располагаться в ячейке 100, с вектором 2 — в ячейке 102. При работе с прерываниями не рекомендуется использовать ячейки 100—109 для других целей.

Процессор начинает обработку прерывания (если они разрешены), завершив текущую команду. При этом он:

1. Получает от контроллера вектор прерывания.
2. *Формирует и помещает в верхушку стека слово, три младших разряда* ([3:5]) которого — текущее значение PC (адрес возврата из прерывания), а разряды [1:2] сохраняют десятичный эквивалент шестнадцатеричной цифры, определяющей значение вектора флагов (I, OV, S, Z). Например, если I = 1, OV = 0, S = 1, Z = 1, то в разряды [1:2] запишется число $11_{10} = 1011_2$.
3. Сбрасывает в 0 флаг разрешения прерывания I.
4. Извлекает из таблицы векторов прерываний адрес обработчика, соответствующий обслуживаемому вектору, и помещает его в PC, осуществляя тем самым переход на подпрограмму обработчика прерывания.

Таким образом, вызов обработчика прерывания, в отличие от вызова подпрограммы, связан с помещением в стек не только адреса возврата, но и текущего значения вектора флагов. Поэтому последней командой подпрограммы обработчика должна быть команда `IRET`, которая не только возвращает в PC три младшие разряда ячейки — верхушки стека (как `RET`), но и восстанавливает те значения флагов, которые были в момент перехода на обработчик прерывания.

Не всякое событие, которое может вызвать прерывание, приводит к прерыванию текущей программы. В состав процессора входит программно-доступный флаг I разрешения прерывания. При I = 0 процессор не реагирует на запросы прерываний. После сброса процессора флаг I так же сброшен и все прерывания запрещены. Для того чтобы разрешить прерывания, следует в программе выполнить команду `EI` (от англ. *enable interrupt*).

Выше отмечалось, что при переходе на обработчик прерывания флаг I автоматически сбрасывается, в этом случае прервать обслуживание одного прерывания другим прерыванием нельзя. По команде `IRET` значение флагов восстанавливается, в т. ч. вновь устанавливается I = 1, следовательно, в основной программе прерывания опять разрешены.

Если требуется разрешить другие прерывания в обработчике прерывания, достаточно в нем выполнить команду `EI`. Контроллер прерываний и процессор на аппаратном уровне блокируют попытки запустить прерывание, если его обработчик начал, но не завершил работу.

Таким образом, флаг I разрешает или запрещает все прерывания системы. Если требуется выборочно разрешить некоторое подмножество прерываний,

используются программно-доступные флаги разрешения прерываний непосредственно на внешних устройствах.

Как правило, каждое внешнее устройство, которое может вызвать прерывание, содержит в составе своих регистров разряд флага разрешения прерывания (см. формат регистров CR и CTR на рис. 8.9, 8.13), по умолчанию установленный в 0. Если оставить этот флаг в нуле, то внешнему устройству запрещается формировать запрос контроллеру прерываний.

Иногда бывает удобно (например, в режиме отладки) иметь возможность вызвать обработчик прерывания непосредственно из программы. Если использовать для этих целей команду CALL, которая помещает в стек только адрес возврата, то команда IRET, размещенная последней в обработчике, может исказить значения флагов (все они будут сброшены в 0, т. к. команда CALL формирует только три младшие разряда ячейки верхушки стека, оставляя остальные разряды в 000).

Поэтому в системах команд многих ЭВМ, в т. ч. и нашей модели, имеются команды вызова прерываний — INT *n* (в нашей модели $n \in \{0, 1, ..., 9\}$), где *n* — вектор прерывания. Процессор, выполняя команду INT *n*, производит те же действия, что и при обработке прерывания с вектором *n*.

Характерно, что с помощью команды INT *n* можно вызвать обработчик прерывания даже в том случае, когда флаг разрешения прерывания I сброшен.

8.8. Программная модель кэш-памяти

К описанной в *разд. 8.1* программной модели учебной ЭВМ может быть подключена программная модель кэш-памяти, структура которой в общем виде отображена на рис. 5.2. Конкретная реализация кэш-памяти в описываемой программной модели показана на рис. 8.15.

Кэш-память содержит N ячеек (в модели N может выбираться из множества {4, 8, 16, 32}), каждая из которых включает трехразрядное поле тега (адреса ОЗУ), шестиразрядное поле данных и три однобитовых признака (флага):

- Z — признак занятости ячейки;
- U — признак использования;
- W — признак записи в ячейку.

Таким образом, каждая ячейка кэш-памяти может дублировать одну любую ячейку ОЗУ, причем отмечается ее занятость (в начале работы модели все ячейки кэш-памяти свободны, $\forall Z_i = 0$), факт записи информации в ячейку во время пребывания ее в кэш-памяти, а также использование ячейки (т. е. любое обращение к ней).

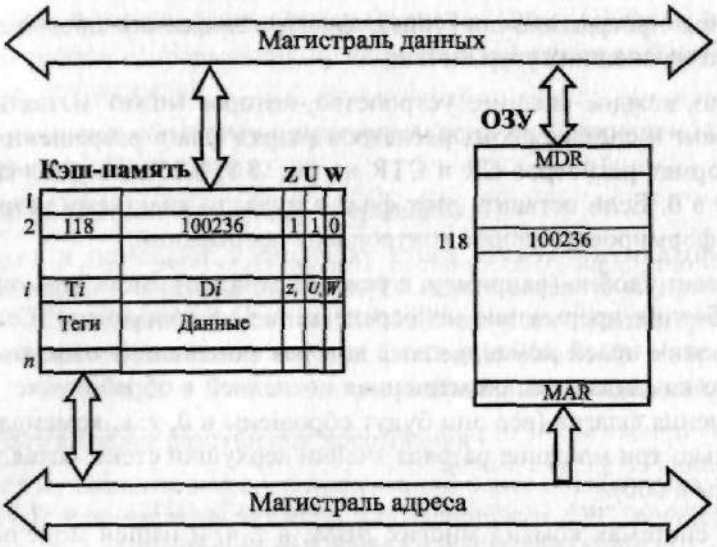


Рис. 8.15. Структура модели кэш-памяти

Текущее состояние кэш-памяти отображается на экране в отдельном окне в форме таблицы, причем количество строк соответствует выбранному числу ячеек кэш. Столбцы таблицы определяют содержимое полей ячеек, например, так, как показано в табл. 8.3.

Таблица 8.3. Пример текущего состояния кэш-памяти

	Теги	Данные	Z	U	W
1	012	220152	1	0	0
2	013	211003	1	1	0
3	050	000025	1	1	1
4	000	000000	0	0	0

Для настройки параметров кэш-памяти можно воспользоваться диалоговым окном **Кэш-память**, вызываемым командой **Вид | Кэш-память**, а затем нажать первую кнопку на панели инструментов открытого окна. После этих действий появится диалоговое окно **Параметры кэш-памяти**, позволяющее выбрать *размер* кэш-памяти, *способ записи* в нее информации и *алгоритм замещения* ячеек.

Напомним, что при сквозной записи при кэш-попадании в процессорных циклах записи осуществляется запись как в ячейку кэш-памяти, так и в ячейку ОЗУ, а при обратной записи — только в ячейку кэш-памяти, причем эта ячейка отмечается битом записи ($W_i := 1$). При очистке ячеек, отмеченных

битом записи, необходимо переписать измененное значение ноля данных в соответствующую ячейку ОЗУ.

При кэш-промахе следует поместить в кэш-память адресуемую процессором ячейку. При наличии свободных ячеек кэш-памяти требуемое слово помещается в одну из них (в порядке очереди). При отсутствии свободных ячеек следует отыскать ячейку кэш-памяти, содержимое которой можно удалить, записав на его место требуемые данные (команду). Поиск такой ячейки осуществляется с использованием *алгоритма замещения строк*.

В модели реализованы три различных алгоритма замещения строк:

- *случайное замещение*, при реализации которого номер ячейки кэш-памяти выбирается случайным образом;
- *очередь*, при которой выбор замещаемой ячейки определяется временем пребывания ее в кэш-памяти;
- *бит использования*, случайный выбор осуществляется только из тех ячеек, которые имеют нулевое значение флага использования.

Напомним, что бит использования устанавливается в 1 при любом обращении к ячейке, однако, как только все биты U_i установятся в 1, все они тут же сбрасываются в 0, так что в кэш всегда ячейки разбиты на два непересекающихся подмножества по значению бита U — те, обращение к которым состоялось относительно недавно (*после* последнего сброса вектора U) имеют значение U = 1, иные — со значением U = 0 являются "кандидатами на удаление" при использовании алгоритма замещения *"бит использования"*.

Если в параметрах кэш-памяти установлен флаг *"с учетом бита записи"*, то все три алгоритма замещения осуществляют поиск "кандидата на удаление" прежде всего среди тех ячеек, признак записи которых не установлен, а при отсутствии таких ячеек (что крайне маловероятно) — среди всех ячеек кэш-памяти. При снятом флаге *"с учетом бита записи"* поиск осуществляется по всем ячейкам кэш-памяти без учета значения W.

Оценка эффективности работы системы с кэш-памятью определяется числом кэш-попаданий по отношению к общему числу обращений к памяти. Учитывая разницу в алгоритмах записи в режимах сквозной и обратной записи, эффективность использования кэш-памяти вычисляется по следующим выражениям (соответственно для сквозной и обратной записи):

$$K = \frac{S_к - S_{к_и}}{S_о}, \tag{8.2}$$

$$K = \frac{S_к - S^i_{к_и}}{S_о}, \tag{8.3}$$

где:

- K — коэффициент эффективности работы кэш-памяти;
- S_0 — общее число обращений к памяти;
- S_K — число кэш-попаданий;
- S_{K_w} — число сквозных записей при кэш-попадании (в режиме сквозной записи);
- $S^i_{K_w}$ — число обратных записей (в режиме обратной записи).

8.9. Вспомогательные таблицы

В данном разделе представлены вспомогательные таблицы (табл. 8.4—8.8) для работы с моделью учебной ЭВМ.

Таблица 8.4. Таблица команд учебной ЭВМ

Мл. \ Ст.	0	1	2	3	4
0	NOP	JMP		MOV	
1	IN	JZ	RD	RD	RDI
2	OUT	JNZ	WR	WR	
3	IRET	JS	ADD	ADD	ADI
4	WRRB	JNS	SUB	SUB	SBI
5	WRSP	JO	MUL	MUL	MULI
6	PUSH	JNO	DIV	DIV	DIVI
7	POP	JRNZ		IN	
8	RET	INT	EI	OUT	
9	HLT	CALL	DI		

Таблица 8.5. Типы адресации, их коды и обозначение

Обозначение	Код	Тип адресации	Пример команды
	0	Прямая (регистровая)	ADD 23 (ADD R3)
#	1	Непосредственная	ADD #33
@	2	Косвенная	ADD @33

Таблица 8.5 *(окончание)*

Обозначение	Код	Тип адресации	Пример команды
[ ]	3	Относительная	ADD [33]
@R	4	Косвенно-регистровая	ADD @R3
@R+	5	Индексная с постинкрементом	ADD @R3+
-@R	6	Индексная с преддекрементом	ADD -@R3

В табл. 8.6 приняты следующие обозначения:

- DD — данные, формируемые командой в качестве (второго) операнда: прямо или косвенно адресуемая ячейка памяти или трехразрядный непосредственный операнд;
- R* — содержимое регистра или косвенно адресуемая через регистр ячейка памяти;
- ADR* — два младших разряда ADR поля регистра CR;
- V — адрес памяти, соответствующий вектору прерывания;
- M(*) — ячейка памяти, прямо или косвенно адресуемая в команде;
- I — пятиразрядный непосредственный операнд со знаком.

Таблица 8.6. Система команд учебной ЭВМ

КОП	Мнемокод	Название	Действие
00	NOP	Пустая операция	Нет
01	IN	Ввод	Acc ← IR
02	OUT	Вывод	OR ← Acc
03	IRET	Возврат из прерывания	FLAGS.PC ← M(SP); INC(SP)
04	WRRB	Загрузка RB	RB ← CR[ADR]
05	WRSP	Загрузка SP	SP ← CR[ADR]
06	PUSH	Поместить в стек	DEC(SP); M(SP) ← R
07	POP	Извлечь из стека	R → M(SP); INC(SP)
08	RET	Возврат	PC → M(SP); INC(SP)
09	HLT	Стоп	Конец командных циклов
10	JMP	Безусловный переход	PC ← CR[ADR]
11	JZ	Переход, если 0	if Acc = 0 then PC ← CR[ADR]

Таблица 8.6 *(продолжение)*

КОП	Мнемокод	Название	Действие
12	JNZ	Переход, если не 0	if Acc ≠ 0 then PC ← CR[ADR]
13	JS	Переход, если отрицательно	if Acc < 0 then PC ← CR[ADR]
14	JNS	Переход, если положительно	if Acc ≥ 0 then PC ← CR[ADR]
15	JO	Переход, если переполнение	if |Acc| > 99999 then PC ← CR[ADR]
16	JNO	Переход, если нет переполнения	if |Acc| ≤ 99999 then PC ← CR[ADR]
17	JRNZ	Цикл	DEC(R); if R > 0 then PC ← CR[ADR]
18	INT	Программное прерывание	DEC(SP); M(SP) ← FLAGS.PC; PC ← M(V)
19	CALL	Вызов подпрограммы	DEC(SP); M(SP) ← PC; PC ← CR(ADR)
20	Нет		
21	RD	Чтение	Acc ← DD
22	WR	Запись	M(*) ← Acc
23	ADD	Сложение	Acc ← Acc + DD
24	SUB	Вычитание	Acc ← Acc – DD
25	MUL	Умножение	Acc ← Acc × DD
26	DIV	Деление	Acc ← Acc/DD
27	Нет		
28	EI	Разрешить прерывание	IF ← 1
29	DI	Запретить прерывание	IF ← 0
30	MOV	Пересылка	R1 ← R2
31	RD	Чтение	Acc ← R*
32	WR	Запись	R* ← Acc
33	ADD	Сложение	Acc ← Acc + R*
34	SUB	Вычитание	Acc ← Acc – R*
35	MUL	Умножение	Acc ← Acc × R*
36	DIV	Деление	Acc ← Acc/R*
37	IN	Ввод	Acc ← ВУ(CR[ADR*])

Таблица 8.6 *(окончание)*

КОП	Мнемокод	Название	Действие
38	OUT	Вывод	ВУ(CR[ADR*]) ← Acc
39	Нет		
40	Нет		
41	RDI	Чтение	Acc ← I
42	Нет		
43	ADI	Сложение	Acc ← Acc + I
44	SBI	Вычитание	Acc ← Acc – I
45	MULI	Умножение	Acc ← Acc × I
46	DIVI	Деление	Acc ← Acc/I

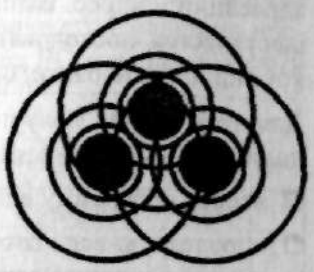
Таблица 8.7. Таблица кодов ASCII (фрагмент)

	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
0				0	@	P	‘	p					А	Р	а	р
1			!	1	A	Q	a	q					Б	С	б	с
2			"	2	B	R	b	r					В	Т	в	т
3			#	3	C	S	c	s					Г	У	г	у
4			$	4	D	T	d	t					Д	Ф	д	ф
5			%	5	E	U	e	u					Е	Х	е	х
6			&	6	F	V	f	v					Ж	Ц	ж	ц
7			‘	7	G	W	g	w					З	Ч	з	ч
8			(	8	H	X	h	x					И	Ш	и	ш
9			)	9	I	Y	i	y					Й	Щ	й	щ
A			*	:	J	Z	j	z					К	Ъ	к	ъ
B			+	;	K	[	k	{					Л	Ы	л	ы
C			,	<	L		l	|					М	Ь	м	ь
D			-	=	M	]	m	}					Н	Э	н	э
E			.	>	N		n						Щ	Ю	щ	ю
F			/	?	O	_	o						П	Я	п	я

Таблица 8.8. Перевод HEX-кодов в десятичные числа

	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
0	0	16	32	48	64	80	96	112	128	144	160	176	192	208	224	240
1	1	17	33	49	65	81	97	113	129	145	161	177	193	209	225	241
2	2	18	34	50	66	82	98	114	130	146	162	178	194	210	226	242
3	3	19	35	51	67	83	99	115	131	147	163	179	195	211	227	243
4	4	20	36	52	68	84	100	116	132	148	164	180	196	212	228	244
5	5	21	37	53	69	85	101	117	133	149	165	181	197	213	229	245
6	6	22	38	54	70	86	102	118	134	150	166	182	198	214	230	246
7	7	23	39	55	71	87	103	119	135	151	167	183	199	215	231	247
8	8	24	40	56	72	88	104	120	136	152	168	184	200	216	232	248
9	9	25	41	57	73	89	105	121	137	153	169	185	201	217	233	249
A	10	26	42	58	74	90	106	122	138	154	170	186	202	218	234	250
B	11	27	43	59	75	91	107	123	139	155	171	187	203	219	235	251
C	12	28	44	60	76	92	108	124	140	156	172	188	204	220	236	252
D	13	29	45	61	77	93	109	125	141	157	173	189	205	221	237	253
E	14	30	46	62	78	94	110	126	142	158	174	190	206	222	238	254
F	15	31	47	63	79	95	111	127	143	159	175	191	207	223	239	255

ГЛАВА 9



Лабораторные работы

Цикл лабораторных работ рассчитан на выполнение студентами в рамках курса "Архитектура ЭВМ" и других, подобных по содержанию.

Цикл включает работы различного уровня. Лабораторные работы № 1—4 ориентированны на первичное знакомство с архитектурой процессора, системой команд, способами адресации и основными приемами программирования на машинно-ориентированном языке. Лабораторная работа № 5 иллюстрирует реализацию командного цикла процессора на уровне микроопераций. Лабораторная работа № 6 посвящена способам организации связи процессора с внешними устройствами, а в лабораторных работах № 7 и 8 рассматривается организация кэш-памяти и эффективность различных алгоритмов замещения.

Все работы выполняются на программной модели учебной ЭВМ и взаимодействующих с ней в программных моделях ВУ и кэш-памяти, описанных в *главе 8*.

Описание работы включает постановку задачи, пример выполнения, набор вариантов индивидуальных заданий, порядок выполнения работы, требования к содержанию отчета и контрольные вопросы.

9.1. Лабораторная работа № 1. Архитектура ЭВМ и система команд

9.1.1. Общие положения

Для решения с помощью ЭВМ некоторой задачи должна быть разработана программа. Программа на языке ЭВМ представляет собой последовательность команд. Код каждой команды определяет выполняемую операцию, тип

адресации и адрес. Выполнение программы, записанной в памяти ЭВМ, осуществляется последовательно по командам в порядке возрастания адресов команд или в порядке, определяемом командами передачи управления.

Для того чтобы получить результат выполнения программы, пользователь должен:

- ввести программу в память ЭВМ;
- определить, если это необходимо, содержимое ячеек ОЗУ и РОН, содержащих исходные данные, а также регистров IR и BR;
- установить в PC стартовый адрес программы;
- перевести модель в режим **Работа**.

Каждое из этих действий выполняется посредством интерфейса модели, описанного в *главе 8*. Ввод программы может осуществляться как в машинных кодах непосредственно в память модели, так и в мнемокодах в окно **Текст программы** с последующим ассемблированием.

Цель настоящей лабораторной работы — знакомство с интерфейсом модели ЭВМ, методами ввода и отладки программы, действиями основных классов команд и способов адресации. Для этого необходимо ввести в память ЭВМ и выполнить в режиме **Шаг** некоторую последовательность команд (определенную вариантом задания) и зафиксировать все изменения на уровне программно-доступных объектов ЭВМ, происходящие при выполнении этих команд.

Команды в память учебной ЭВМ вводятся в виде шестиразрядных десятичных чисел (см. форматы команд на рис. 8.3, коды команд и способов адресации в табл. 8.2—8.4).

В настоящей лабораторной работе будем программировать ЭВМ в машинных кодах.

9.1.2. Пример 1

Дана последовательность мнемокодов, которую необходимо преобразовать в машинные коды, занести в ОЗУ ЭВМ, выполнить в режиме **Шаг** и зафиксировать изменение состояний программно-доступных объектов ЭВМ (табл. 9.1).

Таблица 9.1. Команды и коды

Последовательность	Значения				
Команды	RD#20	WR30	ADD #5	WR@30	JNZ 002
Коды	21 1 020	22 0 030	23 1 005	22 2 030	12 0 002

Введем полученные коды последовательно в ячейки ОЗУ, начиная с адреса 000. Выполняя команды в режиме **Шаг**, будем фиксировать изменения программно-доступных объектов (в данном случае это Acc, PC и ячейки ОЗУ 020 и 030) в табл. 9.2.

Таблица 9.2. Содержимое регистров

PC	Acc	M(30)	M(20)	PC	Acc	M(30)	M(20)
000	000000	000000	000000	004			000025
001	000020			002			
002		000020		003	000030		
003	000025			004			000030

9.1.3. Задание 1

1. Ознакомиться с архитектурой ЭВМ *(см. часть I)*.
2. Записать в ОЗУ "программу", состоящую из пяти команд — варианты задания выбрать из табл. 9.3. Команды разместить в последовательных ячейках памяти.
3. При необходимости установить начальное значение в устройство ввода IR.
4. Определить те программно-доступные объекты ЭВМ, которые будут изменяться при выполнении этих команд.
5. Выполнить в режиме **Шаг** введенную последовательность команд, фиксируя изменения значений объектов, определенных в п. 4, в таблице (см. форму табл. 9.2).
6. Если в программе образуется цикл, необходимо просмотреть не более двух повторений каждой команды, входящей в тело цикла.

Таблица 9.3. Варианты задания 1

№	IR	Команда 1	Команда 2	Команда 3	Команда 4	Команда 5
1	000007	IN	MUL #2	WR10	WR @10	JNS 001
2	X	RD #17	SUB #9	WR16	WR @16	JNS 001
3	100029	IN	ADD #16	WR8	WR@8	JS 001
4	X	RD #2	MUL #6	WR 11	WR @11	JNZ 00
5	000016	IN	WR8	DIV #4	WR @8	JMP 002
6	X	RD #4	WR 11	RD @11	ADD #330	JS 000

Таблица 9.3 *(окончание)*

№	IR	Команда 1	Команда 2	Команда 3	Команда 4	Команда 5
7	000000	IN	WR9	RD @9	SUB#1	JS 001
8	X	RD 4	SUB #8	WR8	WR @8	JNZ 001
9	100005	IN	ADD #12	WR 10	WR @10	JS 004
10	X	RD 4	ADD #15	WR 13	WR @13	JMP 001
11	000315	IN	SUB #308	WR11	WR @11	JMP 001
12	X	RD #988	ADD #19	WR9	WR @9	JNZ 001
13	000017	IN	WR11	ADD 11	WR @11	JMP 002
14	X	RD #5	MUL #9	WR10	WR @10	JNZ 001

9.1.4. Содержание отчета

1. Формулировка варианта задания.
2. Машинные коды команд, соответствующих варианту задания.
3. Результаты выполнения последовательности команд в форме табл. 9.2.

9.1.5. Контрольные вопросы

1. Из каких основных частей состоит ЭВМ и какие из них представлены в модели?
2. Что такое система команд ЭВМ?
3. Какие классы команд представлены в модели?
4. Какие действия выполняют команды передачи управления?
5. Какие способы адресации использованы в модели ЭВМ? В чем отличие между ними?
6. Какие ограничения накладываются на способ представления данных в модели ЭВМ?
7. Какие режимы работы предусмотрены в модели и в чем отличие между ними?
8. Как записать программу в машинных кодах в память модели ЭВМ?
9. Как просмотреть содержимое регистров процессора и изменить содержимое некоторых регистров?
10. Как просмотреть и, при необходимости, отредактировать содержимое ячейки памяти?

11. Как запустить выполнение программы в режиме приостановки работы после выполнения каждой команды?
12. Какие способы адресации операндов применяются в командах ЭВМ?
13. Какие команды относятся к классу передачи управления?

9.2. Лабораторная работа № 2. Программирование разветвляющегося процесса

Для реализации алгоритмов, пути в которых зависят от исходных данных, используют команды условной передачи управления.

9.2.1. Пример 2

В качестве примера (несколько упрощенного по сравнению с заданиями лабораторной работы № 2) рассмотрим программу вычисления функции

$$y = \begin{cases} (x-11)^2 - 125, & \text{при } x \geq 16, \\ \dfrac{x^2 + 72x - 6400}{-168}, & \text{при } x < 16, \end{cases}$$

причем x вводится с устройства ввода IR, результат y выводится на OR.

Граф-схема алгоритма решения задачи показана на рис. 9.1.

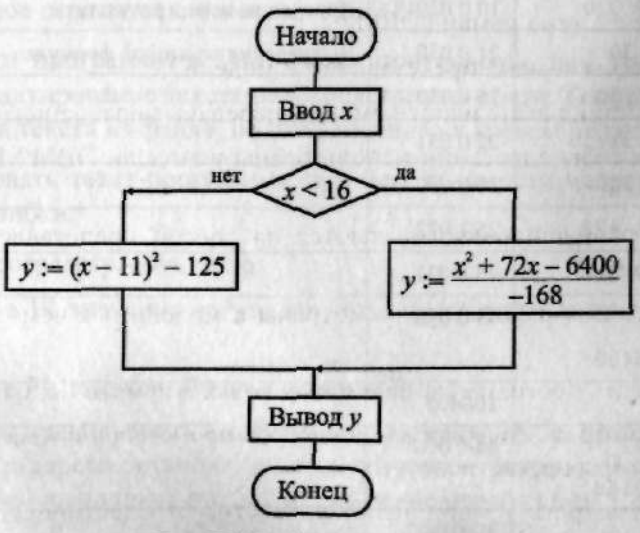


Рис. 9.1. Граф-схема алгоритма

В данной лабораторной работе используются двухсловные команды с непосредственной адресацией, позволяющие оперировать отрицательными числами и числами по модулю, превышающие 999, в качестве непосредственного операнда.

Оценив размер программы примерно в 20—25 команд, отведем для области данных ячейки ОЗУ, начиная с адреса 030. Составленная программа с комментариями представлена в виде табл. 9.4.

Таблица 9.4. Пример программы

Адрес	Команда		Примечание
	Мнемокод	Код	
000	IN	01 0 000	Ввод x
001	WR 30	22 0 030	Размещение x в ОЗУ(ОЗО)
002	SUB #16	24 1 016	Сравнение с границей — $(x-16)$
003	JS 010	13 0 010	Переход по отрицательной разности
004	RD 30	21 0 030	Вычисления по первой формуле
005	SUB #11	24 1 011	
006	WR 31	22 0 031	
007	MUL 31	25 0 031	
008	SUB #125	24 1 125	
009	JMP 020	10 0 020	Переход на вывод результата
010	RD 30	21 0 030	Вычисления по второй формуле
011	MUL 30	25 0 030	
012	WR 31	22 0 031	
013	RD 30	21 0 030	
014	MUL #72	25 1 072	
015	ADD 31	23 0 031	
016	ADI 106400	43 0 000	
017		106400	
018	DIVI 100168	46 0 000	
019		100168	

Таблица 9.4 *(окончание)*

Адрес	Команда		Примечание
	Мнемокод	Код	
020	OUT	02 0 000	Вывод результата
021	HLT	09 0 000	Стоп

9.2.2. Задание 2

1. Разработать программу вычисления и вывода значения функции:

$$y = \begin{cases} F_i(x), & \text{при } x \geq a, \\ F_j(x), & \text{при } x < a, \end{cases}$$

для вводимого из IR значения аргумента x. Функции и допустимые пределы изменения аргумента приведены в табл. 9.5, варианты заданий — в табл. 9.6.

2. Исходя из допустимых пределов изменения аргумента функций (табл. 9.5) и значения параметра a для своего варианта задания (табл. 9.6) выделить на числовой оси Ox области, в которых функция y вычисляется по представленной в п. 1 формуле, и недопустимые значения аргумента. На недопустимых значениях аргумента программа должна выдавать на OR максимальное отрицательное число: 199 999.

3. Ввести текст программы в окно **Текст программы**, при этом возможен набор и редактирование текста непосредственно в окне **Текст программы** или загрузка текста из файла, подготовленного в другом редакторе.

4. Ассемблировать текст программы, при необходимости исправить синтаксические ошибки.

5. Отладить программу. Для этого:

   а) записать в IR значение аргумента $x > a$ (в области допустимых значений);

   б) записать в PC стартовый адрес программы;

   в) проверить правильность выполнения программы (т. е. правильность результата и адреса останова) в автоматическом режиме. В случае наличия ошибки выполнить пп. 5, *г* и 5, *д*; иначе перейти к п. 5, *е*;

   г) записать в PC стартовый адрес программы;

д) наблюдая выполнение программы в режиме **Шаг**, найти команду, являющуюся причиной ошибки; исправить ее; выполнить пп. 5, *а* — 5, *в*;

е) записать в IR значение аргумента $x < a$ (в области допустимых значений); выполнить пп. 5, *б* и 5, *в*;

ж) записать в IR недопустимое значение аргумента x и выполнить пп. 5, *б* и 5, *в*.

6. Для выбранного допустимого значения аргумента x наблюдать выполнение отлаженной программы в режиме **Шаг** и записать в форме табл. 9.2 содержимое регистров ЭВМ перед выполнением каждой команды.

Таблица 9.5. Функции

k	$F_k(x)$	k	$F_k(x)$
1	$\frac{x+17}{1-x}$; $2 \le x \le 12$	5	$\frac{(x+2)^2}{15}$; $50 \le x \le 75$
2	$\frac{(x+3)^2}{x}$; $1 \le x \le 50$	6	$\frac{2x^2+7}{x}$; $1 \le x \le 30$
3	$\frac{1000}{x+10}$; $-50 \le x \le -15$	7	$\frac{x^2+2x}{10}$; $-50 \le x \le 50$
4	$(x+3)^3$; $-20 \le x \le 20$	8	$\frac{8100}{x^2}$; $1 \le x \le 90$

Таблица 9.6. Варианты задания 2

Номер варианта	i	j	a	Номер варианта	i	j	a
1	2	1	12	8	8	6	30
2	4	3	–20	9	2	6	25
3	8	4	15	10	5	7	50
4	6	1	12	11	2	4	18
5	5	2	50	12	8	1	12
6	7	3	15	13	7	6	25
7	6	2	11	14	1	4	5

9.2.3. Содержание отчета

Отчет о лабораторной работе должен содержать следующие разделы:

1. Формулировка варианта задания.
2. Граф-схема алгоритма решения задачи.
3. Размещение данных в ОЗУ.
4. Программа в форме табл. 9.4.
5. Последовательность состояний регистров ЭВМ при выполнении программы в режиме **Шаг** для одного значения аргумента.
6. Результаты выполнения программы для нескольких значений аргумента, выбранных самостоятельно.

9.2.4. Контрольные вопросы

1. Как работает механизм косвенной адресации?
2. Какая ячейка будет адресована в команде с косвенной адресацией через ячейку 043, если содержимое этой ячейки равно 102 347?
3. Как работают команды передачи управления?
4. Что входит в понятие "отладка программы"?
5. Какие способы отладки программы можно реализовать в модели?

9.3. Лабораторная работа № 3. Программирование цикла с переадресацией

При решении задач, связанных с обработкой массивов, возникает необходимость изменения исполнительного адреса при повторном выполнении некоторых команд. Эта задача может быть решена путем использования косвенной адресации.

9.3.1. Пример 3

Разработать программу вычисления суммы элементов массива чисел C_1, $C_2, ..., C_n$. Исходными данными в этой задаче являются: n — количество суммируемых чисел и $C_1, C_2, ..., C_n$ — массив суммируемых чисел. Заметим, что должно выполняться условие $n>1$, т. к. алгоритм предусматривает, по крайней мере, одно суммирование. Кроме того, предполагается, что суммируемые числа записаны в ОЗУ подряд, т. е. в ячейки памяти с последовательными адресами. Результатом является сумма S.

Составим программу для вычисления суммы со следующими конкретными параметрами: число элементов массива — 10, элементы массива расположены в ячейках ОЗУ по адресам 040, 041, 042, ..., 049. Используемые для решения задачи промежуточные переменные имеют следующий смысл: A_i — адрес числа C_i, $i \in \{1, 2, ..., 10\}$; ОЗУ(A_i) — число по адресу A_i, S — текущая сумма; k — счетчик цикла, определяющий число повторений тела цикла.

Распределение памяти таково. Программу разместим в ячейках ОЗУ, начиная с адреса 000, примерная оценка объема программы — 20 команд; промежуточные переменные: A_i — в ячейке ОЗУ с адресом 030, k — по адресу 031, S — по адресу 032. ГСА программы показана на рис. 9.2, текст программы с комментариями приведен в табл. 9.7.

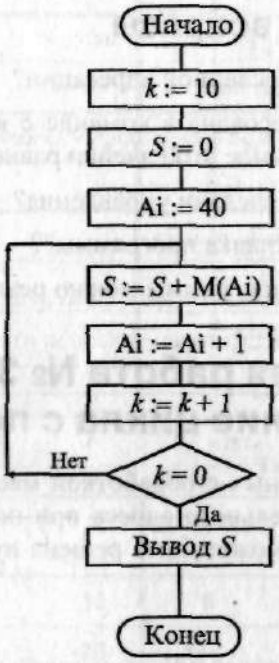


Рис. 9.2. *Граф-схема алгоритма для примера 3*

Таблица 9.7. Текст программы примера 3

Адрес	*Команда*	*Примечание*
000	RD #40	Загрузка начального адреса массива 040
001	WR 30	в ячейку 030

Таблица 9.7 *(окончание)*

Адрес	Команда	Примечание
002	RD #10	Загрузка параметра цикла $k = 10$ в ячейку 031
003	WR 31	
004	RD #0	Загрузка начального значения суммы $S = 0$
005	WR 32	в ячейку 032
006	M1: RD 32	Добавление
007	ADD @30	к текущей сумме
008	WR 32	очередного элемента массива
009	RD30	Модификация текущего
010	ADD #1	адреса массива
011	WR 30	(переход к следующему адресу)
012	RD 31	Уменьшение счетчика
013	SUB #1	(параметра цикла)
014	WR 31	на 1
015	JNZ M1	Проверка параметра цикла и переход при $k \neq 0$
016	RD 32	Вывод
017	OUT	результата
018	HLT	Стоп

9.3.2. Задание 3

1. Написать программу определения заданной характеристики последовательности чисел C_1, C_2, ..., C_n. Варианты заданий приведены в табл. 9.8.
2. Записать программу в мнемокодах, введя ее в поле окна **Текст программы**.
3. Сохранить набранную программу в виде текстового файла и произвести ассемблирование мнемокодов.
4. Загрузить в ОЗУ необходимые константы и исходные данные.
5. Отладить программу.

Таблица 9.8. Варианты задания 3

Номер варианта	Характеристика последовательности чисел $C_1, C_2, ..., C_n$
1	Количество четных чисел
2	Номер минимального числа
3	Произведение всех чисел
4	Номер первого отрицательного числа
5	Количество чисел, равных C_1
6	Количество отрицательных чисел
7	Максимальное отрицательное число
8	Номер первого положительного числа
9	Минимальное положительное число
10	Номер максимального числа
11	Количество нечетных чисел
12	Количество чисел, меньших C_1
13	Разность сумм четных и нечетных элементов массивов
14	Отношение сумм четных и нечетных элементов массивов

Примечание. Под четными (нечетными) элементами массивов понимаются элементы массивов, имеющие четные (нечетные) индексы. Четные числа — элементы массивов, делящиеся без остатка на 2.

9.3.3. Содержание отчета

1. Формулировка варианта задания.
2. Граф-схема алгоритма решения задачи.
3. Распределение памяти (размещение в ОЗУ переменных, программы и необходимых констант).
4. Программа.
5. Значения исходных данных и результата выполнения программы.

9.3.4. Контрольные вопросы

1. Как организовать цикл в программе?
2. Что такое параметр цикла?

3. Как поведет себя программа, приведенная в табл. 9.7, если в ней будет отсутствовать команда WR 31 по адресу 014?
4. Как поведет себя программа, приведенная в табл. 9.7, если метка M1 будет поставлена по адресу 005? 007?

9.4. Лабораторная работа № 4. Подпрограммы и стек

В программировании часто встречаются ситуации, когда одинаковые действия необходимо выполнять многократно в разных частях программы (например, вычисление функции $\sin x$). При этом с целью экономии памяти не следует многократно повторять одну и ту же последовательность команд — достаточно один раз написать так называемую *подпрограмму* (в терминах языков высокого уровня — процедуру) и обеспечить правильный вызов этой подпрограммы и возврат в точку вызова по завершению подпрограммы.

Для *вызова* подпрограммы необходимо указать ее начальный адрес в памяти и передать (если необходимо) параметры — те исходные данные, с которыми будут выполняться предусмотренные в подпрограмме действия. Адрес подпрограммы указывается в команде вызова CALL, а параметры могут передаваться через определенные ячейки памяти, регистры или стек.

Возврат в точку вызова обеспечивается сохранением адреса текущей команды (содержимого регистра PC) при вызове и использованием в конце подпрограммы команды возврата RET, которая возвращает сохраненное значение адреса возврата в PC.

Для реализации механизма вложенных подпрограмм (возможность вызова подпрограммы из другой подпрограммы и т. д.) адреса возврата целесообразно сохранять в стеке. *Стек* ("магазин") — особым образом организованная безадресная память, доступ к которой осуществляется через единственную ячейку, называемую *верхушкой стека*. При записи слово помещается в верхушку стека, предварительно все находящиеся в нем слова смещаются вниз на одну позицию; при чтении извлекается содержимое верхушки стека (оно при этом из стека исчезает), а все оставшиеся слова смещаются вверх на одну позицию. Такой механизм напоминает действие магазина стрелкового оружия (отсюда и второе название). В программировании называют такую дисциплину обслуживания LIFO (Last In First Out, последним пришел — первым вышел) в отличие от дисциплины типа *очередь* — FIFO (First In First Out, первым пришел — первым вышел).

В обычных ОЗУ нет возможности перемещать слова между ячейками, поэтому при организации стека перемещается не массив слов относительно непод-

вижной верхушки, а верхушка относительно неподвижного массива. Под стек отводится некоторая область ОЗУ, причем адрес верхушки хранится в специальном регистре процессора — указателе стека SP.

В стек можно поместить содержимое регистра общего назначения по команде PUSH или извлечь содержимое верхушки в регистр общего назначения по команде POP. Кроме того, по команде вызова подпрограммы CALL значение программного счетчика PC (адрес следующей команды) помещается в верхушку стека, а по команде RET содержимое верхушки стека извлекается в PC. При каждом обращении в стек указатель SP автоматически модифицируется.

В большинстве ЭВМ стек "растет" в сторону меньших адресов, поэтому перед каждой записью содержимое SP уменьшается на 1, а после каждого извлечения содержимое SP увеличивается на 1. Таким образом, SP всегда указывает на верхушку стека.

Цель настоящей лабораторной работы — изучение организации программ с использованием подпрограмм. Кроме того, в процессе организации циклов мы будем использовать новые возможности системы команд модели ЭВМ, которые позволяют работать с новым классом памяти — сверхоперативной (регистры общего назначения — РОН). В реальных ЭВМ доступ в РОН занимает значительно меньшее время, чем в ОЗУ; кроме того, команды обращения с регистрами короче команд обращения к памяти. Поэтому в РОН размещаются наиболее часто используемые в программе данные, промежуточные результаты, счетчики циклов, косвенные адреса и т. п.

В системе команд учебной ЭВМ для работы с РОН используются специальные команды, мнемоники которых совпадают с мнемониками соответствующих команд для работы с ОЗУ, но в адресной части содержат символы регистров R0—R9.

Кроме обычных способов адресации (прямой и косвенной) в регистровых командах используются два новых — постинкрементная и преддекрементная (см. табл. 8.5). Кроме того, к регистровым относится команда организации цикла JRNZ R,M. По этой команде содержимое указанного в команде регистра уменьшается на 1, и если в результате вычитания содержимого регистра не равно 0, то управление передается на метку M. Эту команду следует ставить в конце тела цикла, метку M — в первой команде тела цикла, а в регистр R помещать число повторений цикла.

9.4.1. Пример 4

Даны три массива чисел. Требуется вычислить среднее арифметическое их максимальных элементов. Каждый массив задается двумя параметрами: адресом первого элемента и длиной.

Очевидно, в программе трижды необходимо выполнить поиск максимального элемента массива, поэтому следует написать соответствующую подпрограмму.

Параметры в подпрограмму будем передавать через регистры: R1 — начальный адрес массива, R2 — длина массива.

Рассмотрим конкретную реализацию этой задачи. Пусть первый массив начинается с адреса 085 и имеет длину 14 элементов, второй — 100 и 4, третий — 110 и 9. Программа будет состоять из основной части и подпрограммы. Основная программа задает параметры подпрограмме, вызывает ее и сохраняет результаты работы подпрограммы в рабочих ячейках. Затем осуществляет вычисление среднего арифметического и выводит результат на устройство вывода. В качестве рабочих ячеек используются регистры общего назначения R6 и R7 — для хранения максимальных элементов массивов. Подпрограмма получает параметры через регистры R1 (начальный адрес массива) и R2 (длина массива). Эти регистры используются подпрограммой в качестве регистра текущего адреса и счетчика цикла соответственно. Кроме того, R3 используется для хранения текущего максимума, а R4 — для временного хранения текущего элемента. Подпрограмма возвращает результат через аккумулятор. В табл. 9.9 приведен текст основной программы и подпрограммы. Обратите внимание, цикл в подпрограмме организован с помощью команды JRNZ, а модификация текущего адреса — средствами постинкрементной адресации.

Таблица 9.9. Программа примера 4

Команда	Примечания
Основная программа	
RD #85	Загрузка
WR R1	параметров
RD #14	первого
WR R2	массива
CALL M	Вызов подпрограммы
WR R6	Сохранение результата
RD #100	Загрузка
WR R1	параметров
RD #4	второго
WR R2	массива

Таблица 9.9 *(окончание)*

Команда	Примечания
CALL M	Вызов подпрограммы
WR R7	Сохранение результата
RD #110	Загрузка
WR R1	параметров
RD #9	третьего
WR R2	массива
CALL M	Вызов подпрограммы
ADD R7	Вычисление
ADD R6	среднего
DIV #3	арифметического
OUT	Вывод результата
Подпрограмма MAX	
HLT	Стоп
M: RD @R1	Загрузка
WR R3	первого элемента в R3
L2: RD @R1+	Чтение элемента и модификация адреса
WR R4	Сравнение
SUB R3	и замена,
JS L1	если R3 < R4
MOV R3,R4	
L1: JRNZ R2,L2	Цикл
RD R3	Чтение результата в Acc
RET	Возврат

9.4.2. Задание 4

Составить и отладить программу учебной ЭВМ для решения следующей задачи. Три массива в памяти заданы начальными адресами и длинами. Вычислить и вывести на устройство вывода среднее арифметическое параметров этих массивов. Параметры определяются заданием к предыдущей лабораторной работе (см. табл. 9.8), причем соответствие между номерами вариантов заданий 3 и 4 устанавливается по табл. 9.10.

Таблица 9.10. Соответствие между номерами заданий

Номер варианта задания 4	1	2	3	4	5	6	7	8	9	10	11	12	13	14
Номер строки в табл. 9.9	5	7	13	11	9	12	1	10	14	3	6	8	2	4

9.4.3. Содержание отчета

1. Формулировка варианта задания.
2. Граф-схема алгоритма основной программы.
3. Граф-схема алгоритма подпрограммы.
4. Распределение памяти (размещение в ОЗУ переменных, программы и необходимых констант).
5. Тексты программы и подпрограммы.
6. Значения исходных данных и результата выполнения программы.

9.4.4. Контрольные вопросы

1. Как работает команда `MOV R3, R7`?
2. *Какие действия* выполняет процессор при реализации команды `CALL`?
3. Как поведет себя программа примера 4, если в ней вместо команд `CALL M` использовать команды `JMP M`?
4. После начальной установки процессора (сигнал **Сброс**) указатель стека SP устанавливается в 000. По какому адресу будет производиться запись в стек первый раз, если не загружать SP командой `WRSP`?
5. Как, *используя механизмы постинкрементной и преддекрементной* адресации, организовать дополнительный стек в произвольной области памяти, не связанный с SP?

9.5. Лабораторная работа № 5. Командный цикл процессора

Реализация программы в ЭВМ сводится к последовательному выполнению команд. Каждая команда, в свою очередь, выполняется как последовательность микрокоманд, реализующих элементарные действия над операционными элементами процессора.

В программной модели учебной ЭВМ предусмотрен **Режим микрокоманд**, в котором действие командного цикла реализуется и отображается на уровне микрокоманд. Список микрокоманд текущей команды выводится в специальном окне **Микрокомандный уровень** (см. рис. 8.8).

9.5.1. Задание 5.1

Выполнить снова последовательность команд по варианту задания 1 (см. табл. 9.3), но в режиме **Шаг**. Зарегистрировать изменения состояния процессора и памяти в форме табл. 9.11, в которой приведены состояния ЭВМ при выполнении примера 1 (фрагмент).

9.5.2. Задание 5.2

Записать последовательность микрокоманд для следующих команд модели учебной ЭВМ:

- ADD R3
- ADD @R3
- ADD @R3+
- ADD -@R3
- JRNZ R3,M
- MOV R4,R2
- JMP M
- CALL M
- RET: PUSH R3
- POP R5

9.5.3. Контрольные вопросы

1. Какие микрокоманды связаны с изменением состояния аккумулятора?
2. Какие действия выполняются в модели по микрокоманде MRd? RWr?
3. Попробуйте составить микропрограмму (последовательность микрокоманд, реализующих команду) для несуществующей команды "умножение модулей чисел".
4. Что изменится в работе процессора, если в каждой микропрограмме микрокоманду увеличения программного счетчика PC := PC + 1 переместить в самый конец микропрограммы?

Таблица 9.11. Состояние модели в режиме моделирования на уровне микрокоманд

Адрес (PC)	Мнемокод	Микрокоманда	ОЗУ		CR			АУ		Ячейки	
			MAR	MDR	COP	TA	ADR	Acc	DR	020	030
000	RD #20	MAR := PC	000	000000	00	0	000	000000	000000	000000	000000
		MRd	000								
		CR := MDR		211020							
		PC := PC + 1			21	1	020				
001		Acc := 000.ADR									
	WR 30	MAR := PC						000020			
		MRd	001								
		CR := MDR		220030							
		PC := PC + 1			22	0	030				
002		MAR := ADR									
		MDR := Acc	030								
		MWr		000020							
	ADD #5	MAR := PC									000020
		MRd	002								
		CR := MDR		231005							
		PC := PC + 1			23	1	005				
003		DR := 000.ADR									
		F_{AY} := ALI							000005		
	WR @30	MAR := PC						000025			

9.6. Лабораторная работа № 6. Программирование внешних устройств

Целью этой лабораторной работы является изучение способов организации взаимодействия процессора и внешних устройств (ВУ) в составе ЭВМ.

Выше отмечалось, что связь процессора и ВУ может осуществляться в синхронном или асинхронном режиме. *Синхронный режим* используется для ВУ, всегда готовых к обмену. В нашей модели такими ВУ являются дисплей и тоногенератор — процессор может обращаться к этим ВУ, не анализируя их состояние (правда дисплей блокирует прием данных после ввода 128 символов, формируя флаг ошибки).

Асинхронный обмен предполагает анализ процессором состояния ВУ, которое определяет готовность ВУ выдать или принять данные или факт осуществления некоторого события, контролируемого системой. К таким устройствам в нашей модели можно отнести клавиатуру и блок таймеров.

Анализ состояния ВУ может осуществляться процессором двумя способами:

- в программно-управляемом режиме;
- в режиме прерывания.

В первом случае предполагается программное обращение процессора к регистру состояния ВУ с последующим анализом значения соответствующего разряда слова состояния. Такое обращение следует предусмотреть в программе с некоторой периодичностью, независимо от фактического наступления контролируемого события (например, нажатие клавиши).

Во втором случае при возникновении контролируемого события ВУ формирует процессору запрос на прерывание программы, по которому процессор и осуществляет связь с ВУ.

9.6.1. Задание 6

Свой вариант задания (табл. 9.12) требуется выполнить двумя способами — сначала в режиме программного контроля, далее модифицировать программу таким образом, чтобы события обрабатывались в режиме прерывания программы. Поскольку "фоновая" (основная) задача для этого случая в заданиях отсутствует, роль ее может сыграть "пустой цикл":

```
M: NOP
   NOP
   JMP M
```

Таблица 9.12. Варианты задания 6

№ варианта	Задание	Используемые ВУ	Пояснения
1	Ввод пятиразрядных чисел в ячейки ОЗУ	Клавиатура	Программа должна обеспечивать ввод последовательности ASCII-кодов десятичных цифр (не длиннее пяти), перекодировку в "8421", упаковку в десятичное число (первый введенный символ — старшая цифра) и размещение в ячейке ОЗУ. ASCII-коды не-цифр игнорировать
2	Программа ввода символов с клавиатуры с выводом на дисплей	Клавиатура, дисплей, таймер	Очистка буфера клавиатуры после ввода 50 символов или каждые 10 с
3	Вывод на дисплей трех текстов, хранящихся в памяти, с задержкой	Дисплей, таймер	Первый текст выводится сразу при запуске программы, второй — через 15 с, третий — через 20 с после второго
4	Вывод на дисплей одного из трех текстовых сообщений, в зависимости от нажатой клавиши	Клавиатура, дисплей	<1> — вывод на дисплей первого текстового сообщения, <2> — второго, <3> — третьего, остальные символы — нет реакции
5	Выбирать из потока ASCII-кодов только цифры и выводить их на дисплей	Клавиатура, дисплей, тоногенератор	Вывод каждой цифры сопровождается коротким звуковым сигналом
6	Выводить на дисплей каждый введенный с клавиатуры символ, причем цифру выводить "в трех экземплярах"	Клавиатура, дисплей, тоногенератор	Вывод каждой цифры сопровождается троекратным звуковым сигналом
7	Селективный ввод символов с клавиатуры	Клавиатура, дисплей	Все русские буквы, встречающиеся в строке ввода — в верхнюю часть экрана дисплея (строки 1—4), все цифры — в нижнюю часть экрана (строки 5—8), остальные символы не выводить
8	Вывод содержимого заданного участка памяти на дисплей посимвольно с заданным промежутком времени между выводами символов	Дисплей, таймер	Остаток от деления на 256 трех младших разрядов ячейки памяти рассматривается как ASCII-код символа. Начальный адрес памяти, длина массива вывода и промежуток времени — параметры подпрограммы
9	Программа ввода символов с клавиатуры с выводом на дисплей	Клавиатура, дисплей	Очистка буфера клавиатуры после ввода 35 символов

Таблица 9.12 *(окончание)*

№ варианта	Задание	Используемые ВУ	Пояснения
10	Выводить на дисплей каждый введенный с клавиатуры символ, причем заглавную русскую букву выводить "в двух экземплярах"	Клавиатура, дисплей, таймер	Очистка буфера клавиатуры после ввода 48 символов, очистка экрана каждые 15 с
11	Вывод на дисплей содержимого группы ячеек памяти в числовой форме (адрес и длина группы — параметры подпрограммы)	Дисплей, таймер	Содержимое ячейки распаковывается (с учетом знака), каждая цифра преобразуется в соответствующий ASCII-код и выдается на дисплей. При переходе к выводу содержимого очередной ячейки формируется задержка 10 с
12	Определить промежуток времени между двумя последовательными нажатиями клавиш	Клавиатура, таймер	Результат выдается на OR. (Учитывая инерционность модели, нажатия не следует производить слишком быстро.)

9.6.2. Задания повышенной сложности

1. Разработать программу-тест на скорость ввода символов с клавиатуры. По звуковому сигналу включается клавиатура и таймер на T секунд. Можно начинать ввод символов, причем каждый символ отображается на дисплее, ведется подсчет количества введенных символов (после каждых 50 дается команда на очистку буфера клавиатуры, после 128 — очищается дисплей). Переполнение таймера выключает клавиатуру и включает сигнал завершения ввода (можно тон этого сигнала сопоставить с количеством введенных символов). Параметр T вводится из IR. Результат S — средняя скорость ввода (символ/с) выдается на OR. Учитывая, что модель учебной ЭВМ оперирует только целыми числами, можно выдавать результат в формате $S \times 60$ символов/мин.
2. Разработать программу-тест на степень запоминания текста. Три различных вариантах текста выводятся последовательно на дисплей на T_1 секунд с промежутками T_2 секунд. Далее эти тексты (то, что запомнилось) вводятся с клавиатуры (в режиме ввода строки) и программно сравниваются с исходными текстами. Выдается количество (процент) ошибок.
3. Разработать программу-калькулятор. Осуществлять ввод из буфера клавиатуры последовательности цифр, упаковку (см. задание 1 в табл. 9.12).

Разделители — знаки бинарных арифметических операций и =. Результат переводится в ASCII-коды и выводится на дисплей.

9.6.3. Порядок выполнения работы

1. Запустить программную модель учебной ЭВМ и подключить к ней определенные в задании внешние устройства (меню **Внешние устройства | Менеджер ВУ**).
2. Написать и отладить программу, предусмотренную заданием, с использованием программного анализа флагов готовности ВУ. Продемонстрировать работающую программу преподавателю.
3. Изменить отлаженную в п. 2 программу таким образом, чтобы процессор реагировал на готовность ВУ с помощью подсистемы прерывания. Продемонстрировать работу измененной программы преподавателю.

9.6.4. Содержание отчета

1. Текст программы с программным анализом флагов готовности ВУ.
2. Текст программы с обработчиком прерывания.

9.6.5. Контрольные вопросы

1. При каких условиях устанавливается и сбрасывается флаг готовности клавиатуры Rd?
2. Возможно ли в блоке таймеров организовать работу всех трех таймеров с разной тактовой частотой?
3. Как при получении запроса на прерывание от блока таймеров определить номер таймера, достигшего состояния 99 999 (00 000)?
4. Какой текст окажется на экране дисплея, если после нажатия в окне обозревателя дисплея кнопки **Очистить** и загрузки по адресу CR (11) константы #10 вывести по адресу DR (10) последовательно пять ASCII-кодов русских букв А, Б, В, Г, Д?
5. В какой области памяти модели ЭВМ могут располагаться программы — обработчики прерываний?
6. Какие изменения в работе отлаженной вами второй программы произойдут, если завершить обработчик прерываний командой `RET`, а не `IRET`?

9.7. Лабораторная работа № 7. Принципы работы кэш-памяти

В *разд. 8.8* данной книги описаны некоторые алгоритмы замещения строк кэш-памяти. Цель настоящей лабораторной работы — проверить работу различных алгоритмов замещения при различных режимах записи.

9.7.1. Задание 7

В качестве задания предлагается некоторая короткая "программа" (табл. 9.14), которую необходимо выполнить с подключенной кэш-памятью (размером 4 и 8 ячеек) в шаговом режиме для следующих двух вариантов алгоритмов замещения (табл. 9.13).

Таблица 9.13. Пояснения к вариантам задания 7

Номера вариантов	Режим записи	Алгоритм замещения
1, 7, 11	Сквозная	СЗ, без учета бита записи
	Обратная	О, с учетом бита записи
2, 5, 9	Сквозная	БИ, без учета бита записи
	Обратная	О, с учетом бита записи
3, 6, 12	Сквозная	О, без учета бита записи
	Обратная	СЗ, с учетом бита записи
4, 8, 10	Сквозная	БИ, без учета бита записи
	Обратная	БИ, с учетом бита записи

Таблица 9.14. Варианты задания 7

№ варианта	Номера команд программы						
	1	2	3	4	5	6	7
1	RD #12	WR 10	WR @10	ADD 12	WR R0	SUB 10	PUSH R0
2	RD #65	WRR2	MOV R4,R2	WR 14	PUSH R2	POP R3	CALL 002
3	RD #16	SUB #5	WR 9	WR @9	WR R3	PUSH R3	POP R4
4	RD #99	WR R6	MOV R7,R6	ADD R7	PUSH R7	CALL 006	POP R8
5	RD #11	WR R2	WR -@R2	PUSH R2	CALL 005	POP R3	RET
6	RD #19	SUB #10	WR9	ADD #3	WR ©9	CALL 006	POPR4

Таблица 9.14 *(окончание)*

№ варианта	Номера команд программы						
	1	2	3	4	5	6	7
7	RD #6	CALL 006	WR11	WRR2	PUSH R2	RET	JMP 002
8	RD#8	WRR2	WR @R2+	PUSH R2	POP R3	WR -@R3	CALL 003
9	RD #13	WR14	WR@14	WR@13	ADD 13	CALL 006	RET
10	RD #42	SUB #54	WR16	WR@16	WRR1	ADD @R1+	PUSH R1
11	RD #10	WRR5	ADD R5	WRR6	CALL 005	PUSH R6	RET
12	JMP 006	RD #76	WR 14	WRR2	PUSH R2	RET	CALL 001

Не следует рассматривать заданную последовательность команд как фрагмент программы[1]. Некоторые конструкции, например, последовательность команд PUSH R6, RET в общем случае не возвращает программу в точку вызова подпрограммы. Такие группы команд введены в задание для того, чтобы обратить внимание студентов на особенности функционирования стека.

9.7.2. Порядок выполнения работы

1. Ввести в модель учебной ЭВМ текст своего варианта программы (см. табл. 9.14), ассемблировать его и сохранить на диске в виде txt-файла.
2. Установить параметры кэш-памяти размером 4 ячейки, выбрать режим записи и алгоритм замещения в соответствии с первой строкой своего варианта из табл. 9.13.
3. В шаговом режиме выполнить программу, фиксируя после каждого шага состояние кэш-памяти.
4. Для одной из команд записи (WR) перейти в режим **Такт** и отметить, в каких микрокомандах происходит изменение кэш-памяти.
5. Для кэш-памяти размером 8 ячеек установить параметры в соответствии со второй строкой своего варианта из табл. 9.13 и выполнить программу в шаговом режиме еще раз, фиксируя последовательность номеров замещаемых ячеек кэш-памяти.

[1] Напомним, что программа определяется как последовательность команд, выполнение которых позволит получить некий *результат*.

9.7.3. Содержание отчета

1. Вариант задания — текст программы и режимы кэш-памяти.
2. Последовательность состояний кэш-памяти размером 4 ячейки при однократном выполнении программы (команды 1—7).
3. Последовательность микрокоманд при выполнении команды WR с отметкой тех микрокоманд, в которых возможна модификация кэш-памяти.
4. Для варианта кэш-памяти размером 8 ячеек — последовательность номеров замещаемых ячеек кэш-памяти для второго варианта параметров кэш-памяти при двукратном выполнении программы (команды 1—7).

9.7.4. Контрольные вопросы

1. В чем смысл включения кэш-памяти в состав ЭВМ?
2. Как работает кэш-память в режиме обратной записи? Сквозной записи?
3. Как зависит эффективность работы ЭВМ от размера кэш-памяти?
4. В какую ячейку кэш-памяти будет помещаться очередное слово, если свободные ячейки отсутствуют?
5. Какие алгоритмы замещения ячеек кэш-памяти вам известны?

9.8. Лабораторная работа № 8. Алгоритмы замещения строк кэш-памяти

Цель работы — изучение влияния параметров кэш-памяти и выбранного алгоритма замещения на эффективность работы системы. Эффективность в данном случае оценивается числом кэш-попаданий по отношению к общему числу обращений к памяти. Учитывая разницу в алгоритмах в режимах *сквозной* и *обратной записи*, эффективность использования кэш-памяти вычисляется выражениям (8.2) и (8.3) соответственно для сквозной и обратной записи.

Очевидно, эффективность работы системы с кэш-памятью будет зависеть не только от параметров кэш-памяти и выбранного алгоритма замещения, но и от класса решаемой задачи. Так, линейные программы должны хорошо работать с алгоритмами замещения типа *очередь*, а программы с большим числом условных переходов, зависящих от случайных входных данных, могут давать неплохие результаты с алгоритмами *случайного замещения*. Можно предположить, что программы, имеющие большое число повторяющихся участков (часто вызываемых подпрограмм и/или циклов) при прочих равных условиях обеспечат более высокую эффективность применения кэш-памяти,

чем линейные программы. И, разумеется, на эффективность напрямую должен влиять размер кэш-памяти.

Для проверки высказанных выше предположений выполняется настоящая лабораторная работа.

9.8.1. Задание 8

В данной лабораторной работе все варианты задания одинаковы: исследовать эффективность работы кэш-памяти при выполнении двух разнотипных программ, написанных и отлаженных вами при выполнении лабораторных работ № 2 и 4.

9.8.2. Порядок выполнения работы

1. Загрузить в модель учебной ЭВМ отлаженную программу из лабораторной работы № 2.
2. В меню **Работа** установить режим **Кэш-память**.
3. В меню **Вид** выбрать команду **Кэш-память**, открыв тем самым окно **Кэш-память**, в нем нажать первую слева кнопку на панели инструментов, открыв диалоговое окно **Параметры кэш-памяти**, и установить следующие параметры кэш-памяти: размер — 4, режим записи — сквозная, алгоритм замещения — случайное, без учета бита записи (W).
4. Запустить программу в автоматическом режиме; по окончании работы просмотреть результаты работы кэш-памяти в окне **Кэш-память**, вычислить значение коэффициента эффективности K и записать в ячейку табл. 9.15, помеченную звездочкой.
5. Выключить кэш-память модели (**Работа | Кэш-память**) и изменить один из ее параметров — установить флаг с учетом бита записи (в окне **Параметры кэш-памяти**).
6. Повторить п. 4, поместив значение полученного коэффициента эффективности в следующую справа ячейку табл. 9.15.
7. Последовательно меняя параметры кэш-памяти, повторить пп. 3—5, заполняя все ячейки табл. 9.15.

> ***Совет***
>
> При очередном запуске программы не забывайте устанавливать процессор модели в начальное состояние, нажимая кнопку **R** в окне **Процессор**!

8. Повторить все действия, описанные в пп. 1—7 для программы из лабораторной работы № 4, заполняя вторую таблицу по форме табл. 9.15.

9.8.3. Содержание отчета

1. Две таблицы по форме табл. 9.15 с результатами моделирования программ из лабораторных работ № 2 и 4 при разных режимах работы кэш-памяти.
2. Выводы, объясняющие полученные результаты.

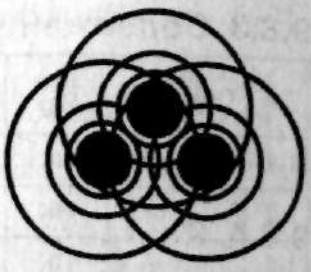
9.8.4. Контрольные вопросы

1. Как работает алгоритм замещения *очередь* при установленном флажке **С учетом бита записи** в диалоговом окне **Параметры кэш-памяти**?
2. Какой алгоритм замещения будет наиболее эффективным в случае применения кэш-памяти большого объема (в кэш-память целиком помещается программа)?
3. Как скажется на эффективности алгоритмов замещения учет значения бита записи W при работе кэш-памяти в режиме обратной записи? Сквозной записи?
4. *Для каких целей в структуру ячейки кэш-памяти включен бит использования. Как устанавливается и сбрасывается этот бит?*

Таблица 9.15. Результаты эксперимента

Способ	*Сквозная запись*					
Алгоритм	**Случайное замещение**		**Очередь**		**Бит U**	
Размер	**без W**	**с W**	**без W**	**с W**	**без W**	**с W**
4	*					
8						
16						
32						
Способ	*Обратная запись*					
Алгоритм	**Случайное замещение**		**Очередь**		**Бит U**	
Размер	**без W**	**с W**	**без W**	**с W**	**без W**	**с W**
4						
8						
16						
32						

ГЛАВА 10



Курсовая работа

10.1. Цель и содержание работы

Целью курсовой работы является:

- обобщение, закрепление и углубление знаний по дисциплинам, связанным с проектированием средств ВТ;
- формирование навыков разработки и оформления текстовой и графической технической документации;
- развитие навыков устных сообщений по содержанию работы.

Содержанием курсовой работы является *разработка арифметико-логического устройства* (АЛУ), реализующего заданный набор операций с учетом ограничений на код выполнения операций и способ построения управляющего автомата.

10.2. Задания

Задания на курсовую работу включают в себя некоторый набор исходных данных и ограничений для проектирования АЛУ. Все варианты задания сведены в табл. 10.1. Строка таблицы представляет один вариант задания, причем номер варианта определяется номером группы (1—2) и порядковым номером студента по списку группы (1—25).

Разрабатываемое АЛУ должно выполнять одну арифметическую и одну поразрядную бинарную логическую операцию, причем на способ выполнения арифметической операции заданием накладываются некоторые ограничения. Варианты операций обозначаются в табл. 10.1 следующим образом:

- ± — алгебраическое сложение/вычитание;
- × — умножение обыкновенное;

Таблица 10.1. Варианты курсовых заданий

№	Операции	Код ВО	Флаги	Тип УА	№	Операции	Код ВО	Флаги	Тип УА
1-1	±, &	ПК	OV, Z	2	2-1	×2, ⊕	ПК	OV, P	4
1-2	×, ∨	ПК	OV, P	3	2-2	×, ⊕	ПК	OV, C	1
1-3	÷1, ⊕	ПК	OV, Z	4	2-3	±, &	ОК	OV, Z	5
1-4	×2, ≡	ПК	OV, C	5	2-4	÷2, ≡	ПК	OV, P	6
1-5	÷2, &	ПК	OV, Z	6	2-5	±, &	ПК	OV, Z	4
1-6	×, ∨	ПК	OV, P	1	2-6	÷1, ∨	ПК	OV, P	3
1-7	±, ≡	ОК	OV, C	2	2-7	±, &	ДК	OV, Z	2
1-8	×2, ⊕	ПК	OV, P	3	2-8	×2, ≡	ПК	OV, P	5
1-9	÷1, &	ПК	OV, Z	4	2-9	÷2, &	ПК	OV, Z	4
1-10	×2, ∨	ПК	OV, C	5	2-10	×2, ∨	ПК	OV, P	6
1-11	±, ≡	ДК	OV, Z	6	2-11	±, &	ОК	OV, Z	1
1-12	×, ∨	ПК	OV, P	5	2-12	÷1, ∨	ПК	OV, Z	2
1-13	±, ⊕	ОК	OV, C	4	2-13	±, &	ДК	OV, C	3
1-14	÷2, ∨	ПК	OV, P	6	2-14	×2, ⊕	ПК	OV, Z	4
1-15	±, &	ДК	OV, Z	3	2-15	÷1, ≡	ПК	OV, P	3
1-16	×, ∨	ПК	OV, C	2	2-16	÷2, ∨	ПК	OV, Z	2
1-17	±, ≡	ПК	OV, Z	1	2-17	±, &	ОК	OV, C	1
1-18	×2, ⊕	ПК	OV, P	1	2-18	×, ⊕	ПК	OV, C	6
1-19	±, &	ОК	OV, C	2	2-19	÷1, &	ПК	OV, Z	5
1-20	÷2, ∨	ПК	OV, P	3	2-20	×2, ∨	ПК	OV, P	1
1-21	÷1, &	ПК	OV, Z	4	2-21	±, ≡	ОК	OV, Z	2
1-22	×, ≡	ПК	OV, C	5	2-22	×, ⊕	ПК	OV, Z	6
1-23	±, &	ДК	OV, Z	6	2-23	±, &	ДК	OV, P	5
1-24	×2, ∨	ПК	OV, P	3	2-24	×2, ∨	ПК	OV, C	4
1-25	×1, ≡	ПК	OV, C	5	2-25	÷2, ≡	ПК	OV, Z	3

- ×2 — умножение ускоренное (с анализом двух разрядов множителя);
- ÷1 — деление с восстановлением остатка;
- ÷2 — деление без восстановления остатка;
- ∨ — дизъюнкция;
- & — конъюнкция;
- ⊕ — неравнозначность;
- ≡ — эквивалентность.

Для всех вариантов заданий исходные данные (операнды) поступают в формате 16-разрядных двоичных чисел с фиксированной запятой, представленных в прямом коде $[a_0 a_1 ... a_{15}]_d$, $[b_0 b_1 ... b_{15}]_d$, причем нулевой разряд является знаковым и запятая фиксирована после знакового разряда. Таким образом, в арифметических операциях участвуют правильные дроби со своими знаками (в логических операциях, естественно, положение запятой и знак игнорируются, операции выполняются над 16-разрядными двоичными векторами). Соответственно, результат операции должен быть представлен в той же форме: $[c_0 c_1 ... c_{15}]_d$.

В задании вводится ограничение на код выполнения операции (столбец **Код ВО** в табл. 10.1). Если код ВО отличается от прямого — обратный (ОК) или дополнительный (ДК), то при выполнении *арифметической операции* следует перевести операнды в заданный код, выполнить в нем операцию, а результат вновь перевести в прямой код. Логические операции, естественно, выполняются без всякого преобразования.

Результатом выполнения операции в АЛУ должно быть не только значение суммы (произведения, конъюнкции и др.) но и *признаки результата* (флаги). Каждый вариант задания предполагает формирования двух различных флагов (заданных в столбце **Флаги** табл. 10.1) из приведенного ниже множества.

- Z — признак нулевого результата;
- P — признак четности числа единиц в результате;
- C — признак переноса (заема) из старшего разряда;
- OV — признак арифметического переполнения.

В столбце **Тип УА** задан номер *типа управляющего автомата*, который необходимо использовать при проектировании заданного АЛУ. Список типов УА приведен ниже.

- 1 — "жесткая логика", автомат Мура;
- 2 — "жесткая логика", автомат Мили;

- 3 — программируемая логика, единый формат микрокоманды, принудительная адресация;
- 4 — программируемая логика, единый формат микрокоманды, естественная адресация;
- 5 — программируемая логика, различные форматы для операционных микрокоманд и микрокоманд перехода, естественная адресация;
- 6 — программируемая логика, различные форматы для операционных микрокоманд и микрокоманд перехода, принудительная адресация.

В задании не определены ограничения на базис логических, операционных элементов и элементов памяти. Поэтому при разработке структурных и функциональных схем можно использовать любые стандартные логические и операционные элементы.

10.3. Этапы выполнения работы

В *главе 4* настоящего пособия подробно рассматривается процесс проектирования цифрового устройства. При проектировании его удобно представить в виде композиции операционного и управляющего автоматов *(см. разд. 4.2)*. Тогда процесс проектирования устройства сводится к процедурам последовательного проектирования операционного и управляющего автоматов. Здесь можно выделить следующие этапы:

1. Разработка алгоритмов выполняемых операций. На этом этапе следует определить список входных, выходных и внутренних переменных и выбрать коды выполняемых операций. Поскольку все задания предполагают реализацию одной/двух арифметических и одной логической операций, целесообразно представить все алгоритмы в форме объединенной ГСА.
2. Разработка структуры операционного автомата — определение состава элементов и связей между ними. Разработка структуры нестандартных элементов. Результатом работы на этом этапе должна стать структурная (функциональная) схема операционного автомата, а также функциональные схемы всех использованных в ОА нестандартных элементов.
3. Определение списка микроопераций и логических условий. Необходимо сопоставить каждому оператору из ГСА микрокоманду или группу микрокоманд, обеспечивающих реализацию этого оператора на разработанной ранее структуре. На этом этапе возможно расширение набора элементов и/или связей структуры, если без такого расширения не удается реализовать все операторы ГСА. Кроме того, необходимо определить, где будут формироваться значения логических переменных, которые анализируются в логических вершинах ГСА и при необходимости предусмотреть специ-

альные элементы структуры для формирования этих значений. Результат работы на этом этапе — списки микроопераций и логических условий ОА.

4. Разработка микропрограммы выполнения заданных операций на выбранной структуре ОА. В простейшем случае можно сохранить топологию графа алгоритма и просто заменить операторы во всех операторных вершинах на соответствующие микрооперации, а условия, которые анализируются в условных вершинах — на соответствующие логические условия из списка, полученного на предыдущем этапе. Однако при переходе от ГСА к микропрограмме следует всегда стремиться к уменьшению числа (операторных) вершин, что, в свою очередь, приведет к упрощению схемы управляющего автомата. Достигнуть этого можно, например, совмещением двух или более операторных вершин ГСА в одну вершину микропрограммы, если смысл реализуемого алгоритма и разработанная ранее структура операционного автомата позволяют выполнить эти действия одновременно. Разработанная на этом этапе микропрограмма является исходной для проектирования управляющего автомата.

На этом заканчивается процесс разработки операционного автомата.

Этапы разработки управляющего автомата различны в зависимости от его типа. Для разработки микропрограммного автомата с *"жесткой" логикой* следует:

1. Осуществить разметку микропрограммы. Эта процедура устанавливает соответствие между вершинами микропрограммы и состояниями автомата. В *разд. 4.4.1* настоящего пособия описано, как осуществлять разметку микропрограммы для проектирования *автомата Мура* и *автомата Мили*.
2. Построить граф автомата. Граф автомата строят по размеченной микропрограмме, причем вершины графа соответствуют состояниям автомата, а ребра — переходам, на этом этапе можно не показывать на графе функцию переходов.
3. Выбрать тип элемента памяти, закодировать состояния автомата.
4. Составить автоматную таблицу переходов. Пример построения такой таблицы для случая использования в качестве элементов памяти D-триггеров приведен в *разд. 4.4.1*. Аналогичный формат имеет таблица при использовании Т-триггеров. Если в качестве элемента памяти автомата выбран RS-триггер, в каждом разряде необходимо сформировать две функции возбуждения — для R- и S-входов.
5. Определить функции возбуждения для переключения элементов памяти. Автоматная таблица может рассматриваться как таблица истинности, задающая функции возбуждения для входов элементов памяти автомата. Все функции возбуждения в общем случае зависят от значений элементов па-

мяти T_j и значений логических условий X_j. При необходимости можно для каждой из функций построить карту Карно и записать ее минимальное выражение. Иногда проще бывает предварительно дешифрировать состояния автомата и записать функции возбуждения в зависимости от текущего состояния автомата и слова логических условий.

6. Определить функции выходов, формирующие значения микроопераций. Для автомата Мура функция выходов в каждом такте дискретного времени зависит только от текущего состояния автомата и значение выхода определяется содержимым операторной вершины микропрограммы, соответствующей этому состоянию автомата. В автомате Мили выходное слово соответствует содержимому той операторной вершины микропрограммы, через которую осуществляется переход из текущего состояния автомата в следующее. Поэтому функция выходов автомата Мили, как и его функция переходов, зависит от текущего состояния автомата и слова логических условий.
7. Построить функциональную схему УА. Получив выражения для функций возбуждения и выходов, можно построить функциональную схему управляющего автомата с использованием выбранных элементов памяти и стандартного базиса логических и операционных элементов.

Для разработки микропрограммного автомата с программируемой логикой следует:

1. Разбить множество микроопераций на подмножества попарно-несовместимых микроопераций (этот пункт не выполняется, если выбран "вертикальный" или "горизонтальный" способ кодирования поля микроопераций).
2. Определить формат микрокоманды (микрокоманд).
3. Разработать функциональную схему управляющего автомата.
4. Заполнить таблицу программирования ПЗУ микрокоманд.

Проектирование управляющего автомата с программируемой логикой с различными способами адресации микрокоманд и кодирования микроопераций подробно описаны в *разд. 4.4.2*.

10.4. Содержание пояснительной записки

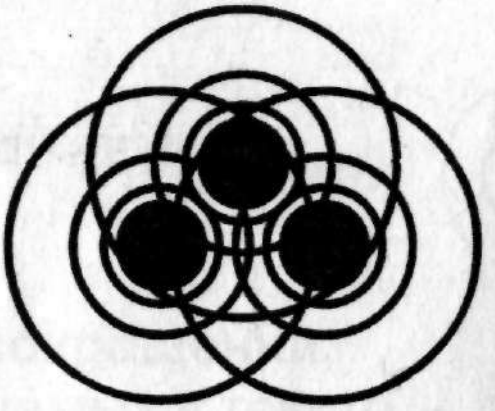
Пояснительная записка к курсовой работе должна включать следующую информацию (для вариантов с управляющими автоматами с "жесткой" логикой):

1. Титульный лист.
2. Задание на проектирование АЛУ.

3. Форматы входных, выходных и внутренних переменных, с которыми оперирует АЛУ.
4. ГСА выполняемых операций и объединенную ГСА.
5. Структурную схему операционного автомата АЛУ.
6. Функциональные схемы "нестандартных" элементов ОА. При необходимости привести процедуры синтеза операционных элементов (например, карты Карно для минимизации булевых функций).
7. Список микроопераций, реализуемых в ОА.
8. Список логических условий, формируемых в ОА.
9. Микропрограмму выполняемых в АЛУ операций в терминах микроопераций и логических условий с разметкой состояний для проектирования управляющего автомата.
10. Граф автомата.
11. Таблицу кодирования внутренних состояний автомата.
12. Описание выбранного элемента памяти (триггера) и его таблица функционирования.
13. Автоматную таблицу переходов.
14. *Выражения для функций возбуждения элементов памяти (при необходимости — процедуру минимизации).*
15. Выражения для функций выходов управляющего автомата.
16. Функциональную схему управляющего автомата.
17. *Заключение.*
18. Библиографический список.

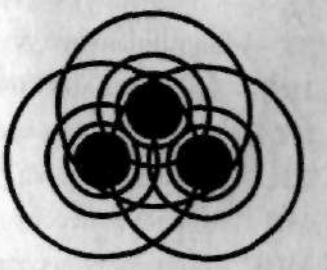
Для вариантов с *управляющими автоматами с программируемой логикой* вместо информации, приведенной в пп. 9—16, необходимо включить:

1. Микропрограмму выполняемых в АЛУ операций в терминах микроопераций и логических условий.
2. Формат или форматы микрокоманд с указанием размеров и назначения полей.
3. Разбиение множества микроопераций ОА на подмножества несовместимых микроопераций.
4. Функциональную схему управляющего автомата.
5. Таблицу программирования ПЗУ микрокоманд.



ПРИЛОЖЕНИЯ

ПРИЛОЖЕНИЕ 1



Список сокращений, используемых в тексте

CISC — Complex Instruction Set Computer, компьютер с полным набором команд

CPL — текущий уровень привилегий

DPL — уровень привилегий дескриптора

EEPROM — энергонезависимая память перепрограммируемая, с электрическим стиранием

GDT — глобальная дескрипторная таблица

IDT — дескрипторная таблица прерываний

IEEE — Institute of Electrical and Electronics Engineers, Институт инженеров по электротехнике и электронике

LDT — локальная дескрипторная таблица.

RISC — Reduced Instruction Set Computer, компьютер с сокращенным набором команд

RPL — запрашиваемый уровень привилегий

TLB — буфер ассоциативной трансляции

TSS — сегмент состояния задачи

АВМ — аналоговая вычислительная машина

АЛС — арифметико-логическая секция

АЛУ — арифметико-логическое устройство

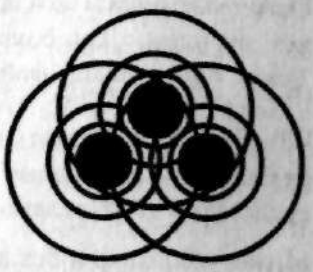
БИС — большая интегральная схема

ВЗУ — внешнее запоминающее устройство

ВУ — внешнее устройство

ГСА — граф-схема алгоритма
ЗУ — запоминающее устройство
ИС — интегральная схема
ИТ — интегральная технология
КЦ — командный цикл
МП — микропроцессор
МПС — микропроцессорная система
МЦ — машинный цикл
ОА — операционный автомат
ОЗУ — *оперативное запоминающее устройство*
ОМЭВМ — однокристальная микроЭВМ
ПВВ — подсистема ввода/вывода
ПДП — прямой доступ в память
ПЗУ — постоянное запоминающее устройство
РОН — регистр общего назначения
СБИС — сверхбольшая интегральная схема
СИС — интегральная схема средней степени интеграции
СОЗУ — сверхоперативное ОЗУ
ТТЛ — транзисторно-транзисторная логика
УА — управляющий автомат
УАЖЛ — управляющий автомат с "жесткой" логикой
УАПЛ — управляющий автомат с программируемой логикой
УВв — устройство ввода
УВВ — устройство ввода/вывода
УВыв — устройство вывода
УОД — устройство обработки данных
УСМ — устройство связи с магистралью
УУС — устройство управления и синхронизации
ЦУУ — центральное устройство управления
ЭЦВМ — электронные цифровые вычислительные машины

ПРИЛОЖЕНИЕ 2



Описание компакт-диска

Компакт-диск содержит исполняемые файлы программной модели учебной ЭВМ, подробно описанной в *разд. 8.1*. Программная модель представлена одним exe-файлом, не требует инсталляции, может размещаться в любом каталоге и запускается обычным образом.

При работе программы формируется текстовый log-файл, в который помещается информация о процессе функционирования модели. Текст этого файла выводится в основное окно модели **Модель учебной ЭВМ**, а при закрытии программы сохраняется по умолчанию в текущем каталоге или записывается по пути, указанному в диалоговом окне, вызываемом через последовательность **Работа | Настройки | Параметры системы**.

На компакт-диске размещены два исполняемых файла:

- CompModel.exe, объемом 496 Кбайт;
- CompModel-2K.exe, объемом 868 Кбайт.

Основным файлом является CompModel.exe, который может работать на любой ПЭВМ с операционной системой Windows XP.

Для работы с предыдущими версиями ОС Windows (Windows 2000 и Windows 98) можно использовать файл CompModel-2K.exe, обладающий несколько меньшими функциональными возможностями. Эта программа реализует ту же модель ЭВМ, что и CompModel, кроме возможности подключения модели кэш-памяти.

При желании можно запустить программу CompModel.exe под Windows 2000 или Windows 98 — функционировать модель будет правильно (в том числе и кэш-память), но возникнут проблемы с сохранением созданного текста из окна **Текст программы**.

Текстовые файлы, созданные в других редакторах, загружаются в это окно без проблем. Для сохранения написанного (отредактированного) в окне **Текст программы** файла можно выделить весь текст (нажать комбинацию клавиш <Ctrl>+<A>), скопировать его в буфер обмена (комбинация клавиш <Ctrl>+<C>) и вставить в окно любого предварительно открытого текстового редактора (комбинация клавиш <Ctrl>+<V>), после чего сохранить файл средствами этого редактора.

Можно обойтись и без внешнего редактора, если *после успешного ассемблирования* сохранить дизассемблированный текст из окна **Программа**. Однако при таком способе сохранения текста программы имена меток (если они существуют в программе) будут заменены на их числовые значения.

Литература

1. Акушский И. Я., Юдицкий Д. И. Машинная арифметика в остаточных классах. — М.: Сов. радио, 1968.
2. Баранов С. И. Синтез микропрограммных автоматов. — Л.: Энергия, 1974.
3. Григорьев В. Л. Микропроцессор i486. Архитектура и программирование. В четырех книгах. — М.: ГРАНАЛ, 1993.
4. Жмакин А. П., Титов В. С. Однокристальные микроЭВМ в системах управления: Учебное пособие. — Курск: Курск. гос. тех. ун-т, 2002.
5. Закревский А. Д. Алгоритмы синтеза дискретных автоматов. — М.: Наука, 1971.
6. Каган Б. М. ЭВМ и системы. — М.: Энергоатомиздат, 1985.
7. Майоров С. А., Новиков Г. А. Принципы организации ЦВМ. — Л.: Машиностроение, 1974.
8. Савельев А. Я. Прикладная теория цифровых автоматов. — М.: Высшая школа, 1987.
9. Соловьев Г. Н. Арифметические устройства ЦВМ. — М.: Энергия, 1978.
10. Стариченко Б. Е. Теоретические основы информатики. — М.: Горячая линия — Телеком, 2003.
11. Таненбаум Э. Архитектура компьютера. 4-е изд. — СПб.: Питер, 2003.
12. Хамахер К., Вранешич З., Заки С. Организация ЭВМ. 5-е изд. — СПб.: Питер, 2003.
13. Хвощ С. Т. и др. Микропроцессоры и микроЭВМ в системах автоматического управления: Справочник. — Л.: Машиностроение, 1987.
14. Юров В. И. Assembler. — СПб.: Питер, 2003.

Предметный указатель

Б

В

Г

Д

З

И

К

Л

М

Н

О

П

Р

С

Т

У

Ф

Ц

Ч

Ш

Э

Жмакин Анатолий Петрович, кандидат технических наук, доцент кафедры программного обеспечения и администрирования информационных систем Курского государственного университета. Автор научных работ и пособий в области проектирования микропроцессорных систем различного назначения.

АРХИТЕКТУРА ЭВМ

В рамках одного издания студент получает возможность изучить теоретические вопросы по основным разделам курса, выполнить лабораторный практикум и курсовое проектирование.

В теоретической части рассмотрены общие вопросы функциональной и структурной организации ЭВМ и архитектура микропроцессоров и микропроцессорных систем. Подробно изложены темы, зачастую отсутствующие в современной литературе, посвященной аппаратным средствам: арифметические основы ЭВМ, организация микропрограммного управления, устройство и взаимодействие разных уровней памяти ЭВМ, проектирование операционных и управляющих автоматов. Лабораторные работы, выполняемые с помощью программной модели учебной ЭВМ, позволят получить наглядное представление о командном цикле процессора и взаимодействию устройств ЭВМ. Благодаря наличию всех компонентов обучения, книга будет полезна как студентам, так и преподавателям ИТ-специальностей.

Компакт-диск содержит программную модель учебной ЭВМ для выполнения лабораторного практикума

ISBN 5-94157-719-2

БХВ-ПЕТЕРБУРГ
194354, ул. Есенина, 5Б
E-mail: mail@bhv.ru
Internet: www.bhv.ru
Тел./факс: (812) 591-6243