# 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 1642г. 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 вычислительны 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.

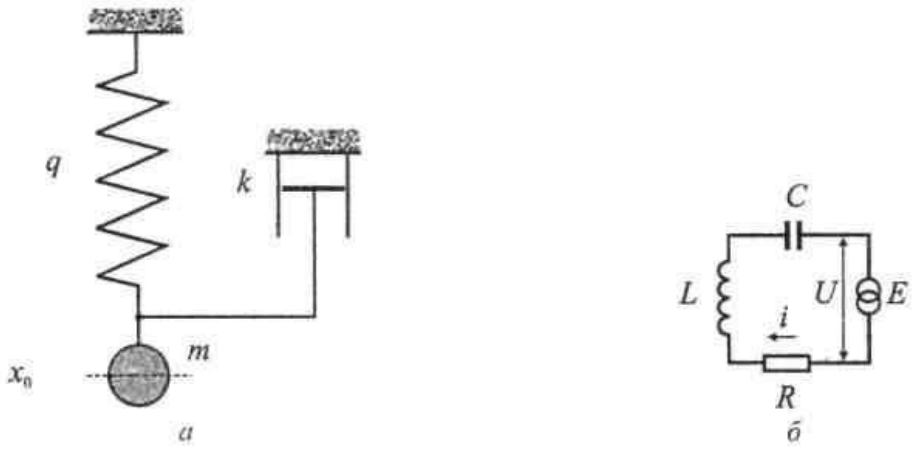



**Рис. 1.1.** Модель: *а* — механическая система; *б* — "аналогичная" ей электрическая цепь

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" (FORNRAN) 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 выделять 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

---

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

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.

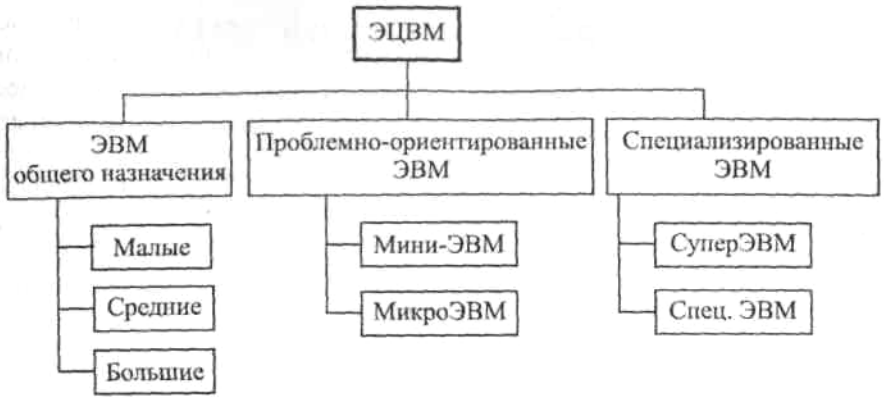



**Рис. 1.2.** Вариант классификации ЭВМ

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".

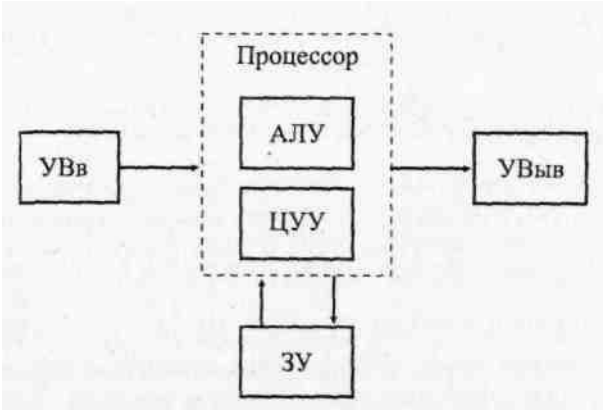



**Рис. 1.3.** Классическая архитектура ЭВМ

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 выполнение. *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 управляющих системах.

Действительно, поскольку и команды программы, и данные кодируются 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 про граммы 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 программ; 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, выполнение 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 слове, 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).*

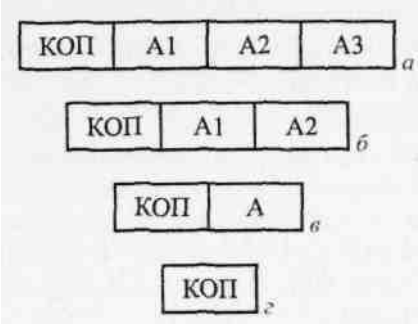



**Рис. 2.1.** Форматы команд: *а* — трехадресный; *б* — двухадресный; *в* — одноадресный; *г* — безадресный

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 (ОАА, десятичная коррекция аккумулятора).

Поэтому в системах команд реальных ЭВМ трехадресные команды встречаются редко. Чаше 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 LIFO — 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 признаке (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 представления 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 $Z_p \rightarrow Z_q$ . 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 $Z_p \rightarrow Z_r \rightarrow Z_q$ 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 $Z_P \rightarrow Z_1 \rightarrow Z_q$**

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.

| 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:** $\pi$ — 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 $Z_P \rightarrow Z_w \rightarrow Z_q$

Очевидно, первая и вторая часть преобразования не связаны друг с другом, что дает основание рассматривать их по отдельности. Алгоритмы перевода $Z_w \rightarrow Z_q$ 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,..., $m$-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 $\gamma_i$, а число с $i$ разрядами с ($i$-1)-го по 0-й— $\delta_i$. Очевидно, $i \in [0, m\text{-}1]$, $\gamma_0 = \delta_{m-1} = Z_q$.

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

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:

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

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:

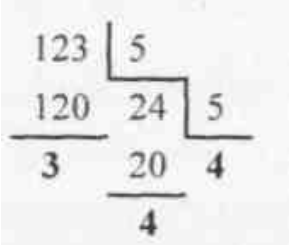
$$b_i = \delta_i \text{ div } q^i \; ; \; \delta_{i-1} = \delta_i \bmod q^i \, . \qquad (3.4)$$

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 пор, 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 $123_{10} \rightarrow Z_5$. 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 ($1\log_5 123 = 2{,}99$, т.е. $m$–1 = 2). Далее:

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

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

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

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

**Пример 3.2**

Выполнить преобразование $443_5 \rightarrow Z_{10}$. Решение:

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, $Z_3 \rightarrow Z_8$ проще осуществить через промежуточное преобразование к десятичной системе $Z_3 \rightarrow Z_{10} \rightarrow Z_8$. 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 $0, Y_p \rightarrow 0, Y_q$ .

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. $0, Y_p \rightarrow 0, Y_{10} \rightarrow 0, Y_q$ .

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

*Алгоритмы перевода* $0, Y_{10} \rightarrow 0, Y_q$ 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:

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

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

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

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

Если вновь позаимствовать в PASCFL 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:

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

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

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

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

$$\begin{array}{lll} 0{,}375 \times 2 = & \mathbf{0,} & 750 \\ 0{,}75 \times 2 = & \mathbf{1,} & 50 \\ 0{,}5 \times 2 = & \mathbf{1,} & 0 \end{array}$$

**Рис. 3.2.** Результат выполнения примера 3.3

Перевод $0{,}Y_p$ ->$0{,}Y_{10}$, 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 $5{,}3(3)_{10} \rightarrow Y_8$.

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

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 $Z_p \rightarrow Z_q$ 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

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

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

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

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

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, σ = log10/log2 = 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.

**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 ( $N$ ) | $2^6 = 64$ | $3^4 = 81$ | $4^3 = 64$ | $6^2 = 36$ | $12^1 = 12$ |

Точное расположение максимума экономичности может быть установлено путем следующих рассуждений. Пусть имеется $n$ знаков для записи чисел, а основание системы числения равно $p$ . Тогда количество разрядов числа $k = n/p$, а общее количество чисел $N$, которые могут быть составлены, равно:

$$N = p^{\frac{n}{p}}. \qquad (3.10)$$

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 $p$ .

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

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* представление 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

$$|A| \le 2^n - 1, \qquad (3.12)$$

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

$$|A| \le 1 - 2^{-n}. \qquad (3.13)$$

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

$$a_0, a_1a_2a_3 \dots a_n. \qquad (3.14)$$

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

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

## 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 $B$ ($c_0 := b_0$).

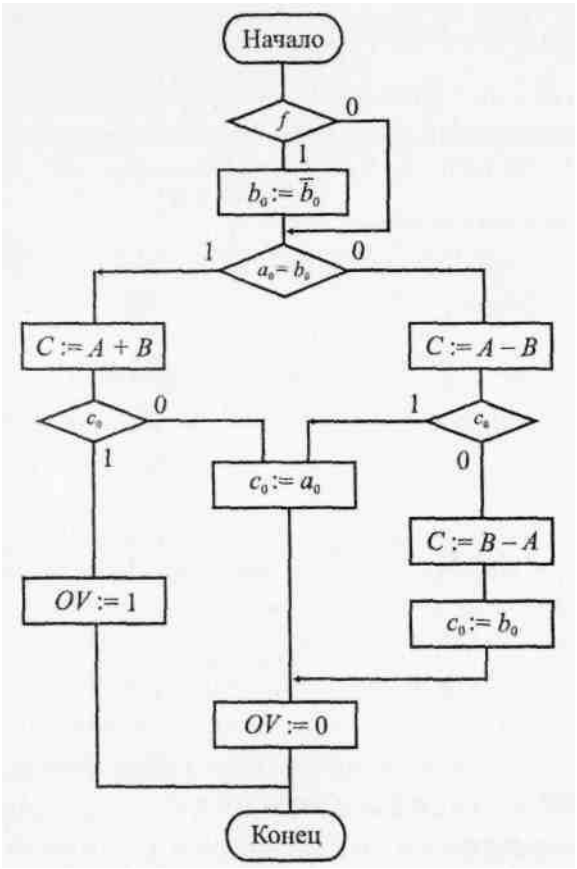



**Рис. 3.3.** Граф алгоритма алгебраического сложения-вычитания

## 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:

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

Действительно, из (3.17) следует, что при $A = –0,a_1a_2a_3 \ldots a_n$ обратный код числа $[A]_i = 2 + A – 2^{-n}$ . Откуда $[A]_i – A = 2 – 2^{-n}$.

$$1,\overline{a}_1\overline{a}_2\ldots\overline{a}_n + 0,a_1a_2\ldots a_n = (1+0),(\overline{a}_1 + a_1)(\overline{a}_2 + a_2)\ldots(\overline{a}_n + a_n) = \\ = 1,11\ldots1 = 2 - 2^{-n},$$

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.

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

$[A>0]_i + [B<0]_i = A + 2 + B – 2^{-n}$. 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.

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

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

□ Случай 4.

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

$[A<0]_i + [B <0]_i = 2 + A – 2^{-n} + 2 + B – 2^{-n}$. 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.

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

**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.

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

**Рис. 3.5.** Результат выполнения примера 3.6

**Пример 3.7**

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

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

**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.

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

**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.

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

**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.

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

**Рис. 3.10.** Результат выполнения примера 3.11

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

$$OV = \bar{a}_0 \bar{b}_0 c_0 \vee a_0 b_0 \bar{c}_0 .$$

**Пример 3.12**

Сложить числа в обратном коде *(A> B , B* > 0 , |*A* + *B*| = 1). Результат — и 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.

$$
\begin{array}{lll}
A = -0{,}0111 & [A]_d = 1{,}0111 & [A]_i = 1{,}1000 \\
B = -0{,}1001 & [B]_d = 1{,}1001 & [B]_i = 1{,}0110 \\
 & & \mathbf{1} \leftarrow 0{,}1110 \\
 & & \quad 1 \\
C = +0{,}1111 \Leftarrow & [C]_d = 0{,}1111 \Leftarrow & [C]_i = 0{,}1111
\end{array}
$$

**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:

$$[-0, a_1 a_2 \ldots a_n]_c = 1, \overline{a}_1 \overline{a}_2 \ldots \overline{a}_n + 2^{-n} = [-0, a_1 a_2 \ldots a_n]_i + 2^{-n}. \qquad (3.21)$$

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

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

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 | ⇒ | $[A]_d = 0{,}0111$ | ⇒ | $[A]_i = 0{,}0111$ | ⇒ | $[A]_c = 0{,}0111$ |
| −0,0111 | ⇒ | $[A]_d = 1{,}0111$ | ⇒ | $[A]_i = 1{,}1000$ | ⇒ | $[A]_c = 1{,}1001$ |
| −0,1000 | ⇒ | $[A]_d = 1{,}1000$ | ⇒ | $[A]_i = 1{,}0111$ | ⇒ | $[A]_c = 1{,}1000$ |
| −0,0101 | ⇒ | $[A]_d = 1{,}0101$ | ⇒ | $[A]_i = 1{,}1010$ | ⇒ | $[A]_c = 1{,}1011$ |

**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.

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

$[A<0]_i + [B<0]_i = 2 + A + 2 + B$ . 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.

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

**Рис. 3.14.** Результат выполнения примера 3.15

**Пример 3.16**

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

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

**Рис. 3.15.** Результат выполнения примера 3.16

**Пример 3.17**

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

$$
\begin{array}{lcll}
A = -0{,}0101 & & [A]_d = 1{,}0101 & [A]_c = 1{,}1011 \\
B = -0{,}0110 & & [B]_d = 1{,}0110 & [B]_c = 1{,}1010 \\
C = -0{,}1011 & \Leftarrow & [C]_d = 1{,}1011 & \Leftarrow \quad [C]_c = \overline{+}1{,}0101
\end{array}
$$

**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.

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

**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.

$$
\begin{array}{lcll}
A = +0{,}1101 & & [A]_d = 0{,}1101 & [A]_c = 0{,}1101 \\
B = +0{,}0101 & & [B]_d = 1{,}0101 & [B]_c = 0{,}0101 \\
C = -0{,}1110 & \Leftarrow & [C]_d = 1{,}1110 & \Leftarrow \quad [C]_c = 1{,}0010
\end{array}
$$

**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.

$$
\begin{array}{lcll}
A = -0{,}1011 & & [A]_d = 1{,}1011 & [A]_c = 1{,}0101 \\
B = -0{,}0011 & & [B]_d = 1{,}1000 & [B]_c = 1{,}1000 \\
C = +0{,}1101 & \Leftarrow & [C]_d = 0{,}1101 & \Leftarrow \quad [C]_c = +0{,}1101
\end{array}
$$

**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,

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

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

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

**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:

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

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:

- $A' = a_1 a_2 \ldots a_n$, $B' = b_1 b_2 \ldots b_n$, $C' = c_1 c_2 \ldots c_n$ —модули чисел;
- $c_{-1}$ — перенос из знакового разряда;
- $\alpha^* = \bar{a}_0 \bar{b}_0 c_0 \vee a_0 b_0 \bar{c}_0 \vee c_0 \bar{c}_1 \bar{c}_2 \ldots \bar{c}_n$ — ситуации переполнения в дополнительном коде.

В алгоритме рис. 3.22 можно отметить один недостаток. При выполнении вычитания ($f$ = 1) необходимо получить дополнение второго операнда: $B'$ := $B'$ + 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 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:

$$B' := \bar{B}';$$
$$C := A + B + f.$$

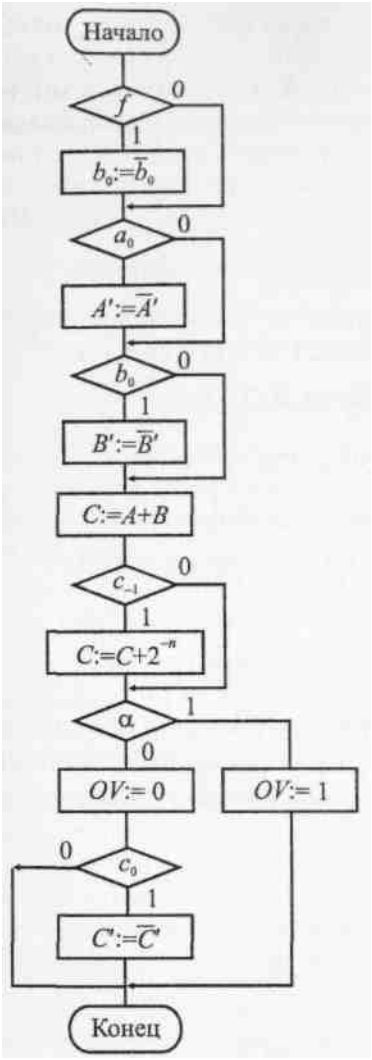



**Рис. 3.21.** Алгоритм алгебраического сложения в обратном коде

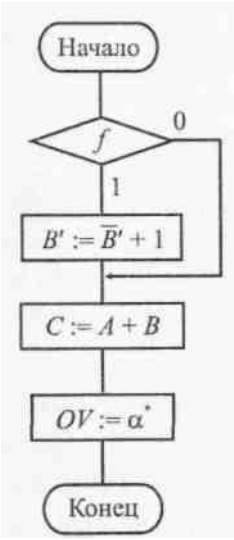



**Рис. 3.22.** Алгоритм алгебраического сложения/вычитания в дополнительном

## 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:

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

а модуль произведения равен произведению модулей сомножителей.

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

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

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.

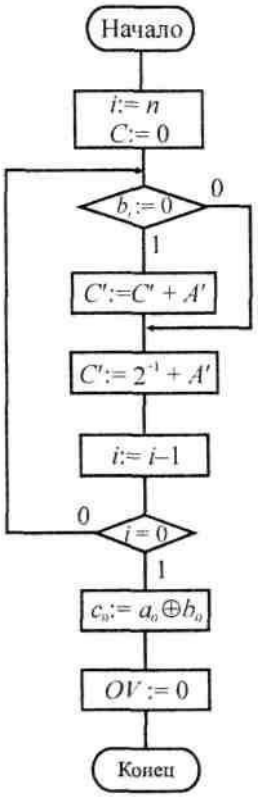
Очевидно, число шагов при использовании приведенного выше метода равно разрядности модуля множителя. Алгоритм умножения чисел, представленных в прямом коде, приведен на рис. 3.23.

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

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




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.

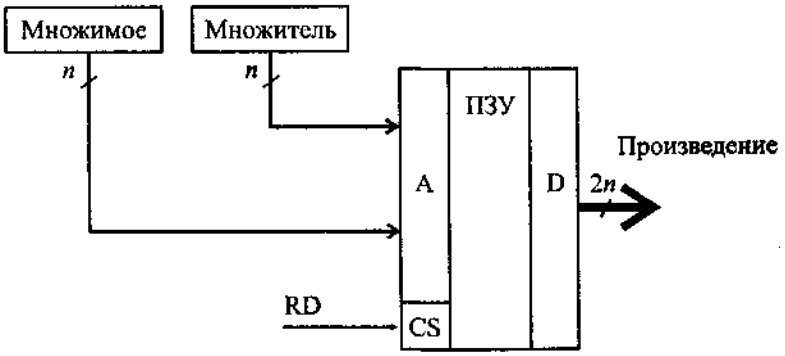
Дополнительные затраты оборудования при реализации логических методов ускорения умножения не зависят от разрядности операндов. Усложняется в основном схема управления АЛУ. В ЭВМ для ускорения умножения часто используются комбинации этих методов.

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.

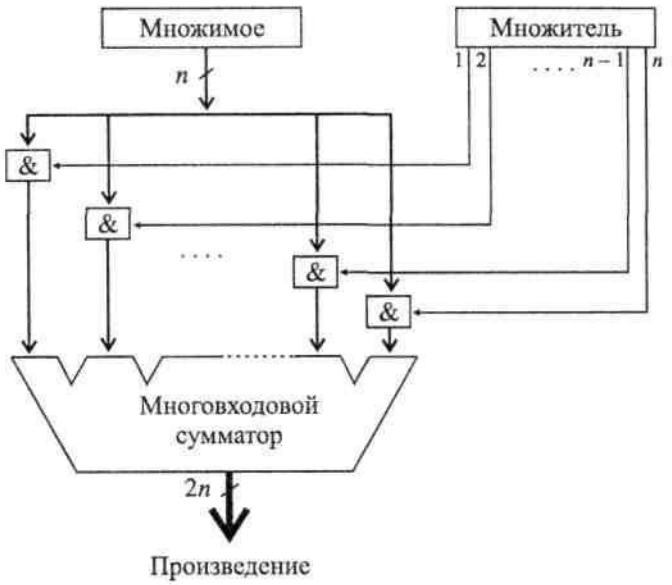


**Рис. 3.24.** Матричное умножение

**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:

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

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 × 01 = 0001 | 10 × 10 = 0100 |
| 01 × 10 = 0010 | 10 × 11 = 0110 |
| 01 × 11 = 0011 | 11 × 11 = 1001 |

**Рис. 3.26.** Вспомогательная таблица умножения

**Пример 3.22**

Выполним умножение 6 × 10 = 60 или в двоичном коде 01.10×10.10 = 00111100. Из таблицы получаем частичные произведения: $A_l \times B_l$ =10×10 = 0100, $A_l \times B_h$ = 10×10 = 0100, $A_h \times B_l$=01×10 = 0010, $A_h \times B_h$ =01×10 = 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 7×11 = 77 или в двоичном коде 01.11×10.11 = 01001101.

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

Теперь сложим частичные произведения, предварительно сдвинув их в

соответствии с (3.28). Результат— на рис. 3.28.

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

**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 и 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

$$|A| < |B|, \qquad (3.29)$$

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. Вычитают 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.

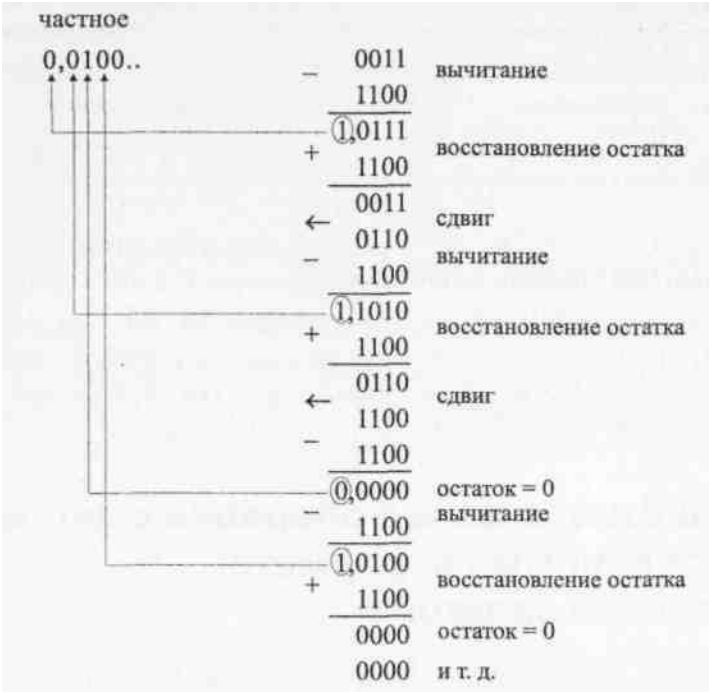



**Рис. 3.29.** Пример деления

## 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 1,5*n* 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, можно не восстанавливать, но на следующем шаге в этом случае нужно вместо 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:

$$N = m \cdot q^p, \qquad (3.30)$$

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.

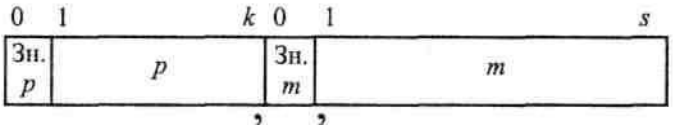



**Рис. 3.30.** Формат числа с плавающей запятой

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.

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

Мантисса числа в стандарте IEEE нормализована и лежит в диапазоне $1 \le T < 2$.

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.

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).

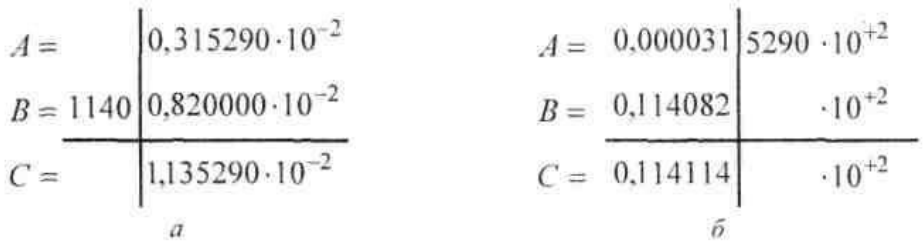



**Рис. 3.31.** Выравнивание порядков

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).

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

**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.*

$$
\begin{array}{ll}
A = 0{,}24512 \cdot 10^{-8} & \\
B = -0{,}24392 \cdot 10^{-8} & \\
\hline
C = 0{,}00120 \cdot 10^{-8} & = 0{,}12000 \cdot 10^{-10}
\end{array}
$$

**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".

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.

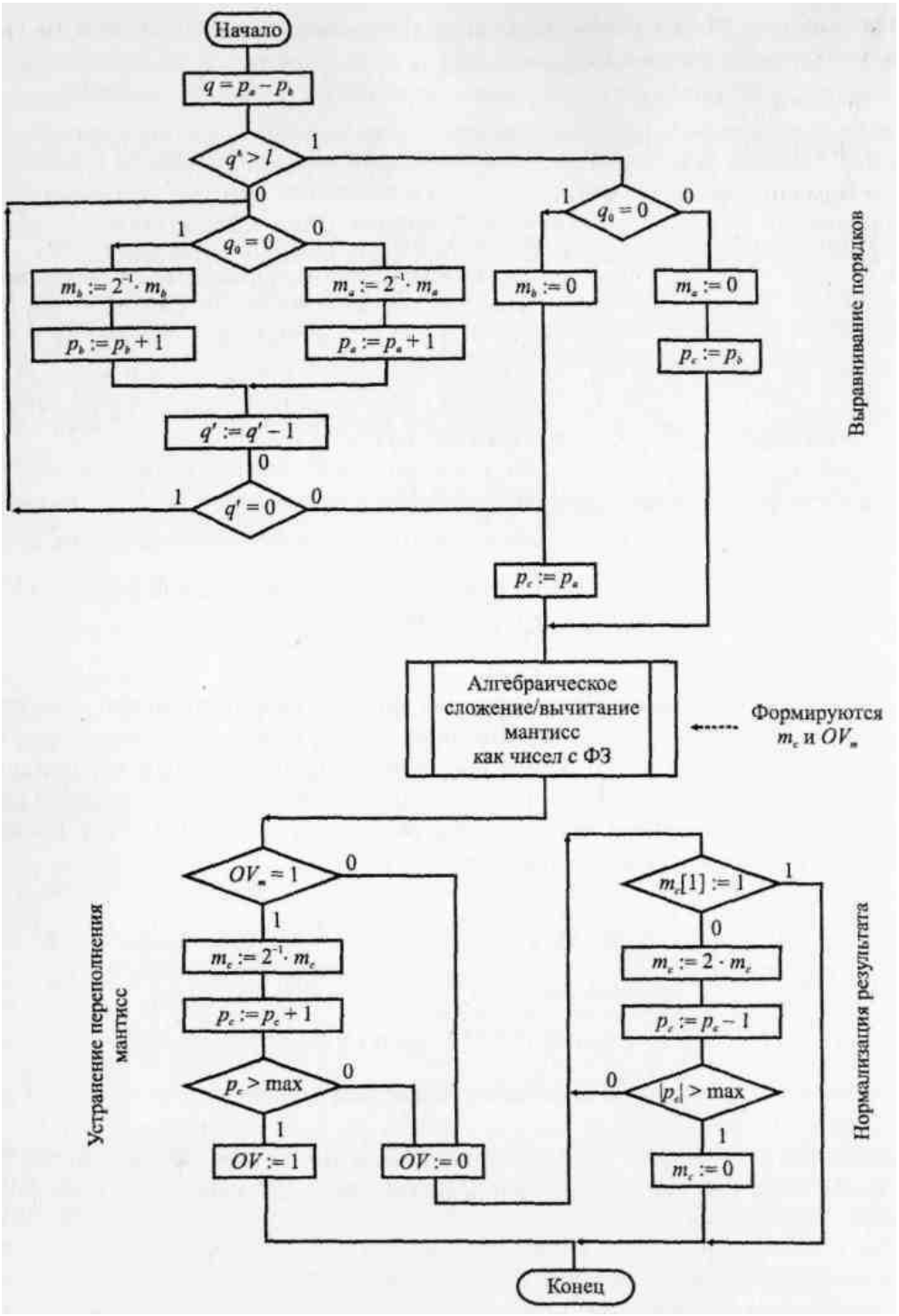



**Рис. 3.34.** Алгоритм операции сложения чисел с плавающей запятой

## 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 пример, цифра 7 может быть представлена как 1100 или как 1011. Для одоления 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 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.

$A = 4754 = 0100\ \ 0111\ 0101\ 0100$
$B = 2917 = 0010\ \ 1001\ 0001\ 0111$
$C = 7671 = 0111\ \leftarrow 0000\ 0110\ 1011$
$7\ \ \ 0\ \ \ 6\ \ \ 11$

**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 ряды 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**

$A = 4754 = 0100 \leftarrow 0111\ 0101\ \ 0100$
$B = 2917 = 0010\ \ 1001\ 0001\ \ 0111$
$C = 7671\ \ 0111\ \ 0000\ 0110 \leftarrow 1011$
$0110\ \ \ \ \ \ 0110$
$= 0111\ 0110\ \ 0111\ 0001$
$7\ \ \ \ 6\ \ \ \ 7\ \ \ \ 1$

**Рис. 3.36.** Результат выполнения примера 3.27

Результат — на рис. 3.36.
Рассмотрим еще один пример арифметического сложения в коде "8421".

Пример 3.28

Результат — на рис. 3.37.

$A = 3852 = 0011\ \ 1000\ 0101\ \ 0010$
$B = 5179 = 0101\ \ 0001\ 0111\ \ 1001$
$C = 9031\ \ 1000\ \ 1001 \leftarrow 1100 \leftarrow 1011$
$0110\ \ 0110$
$1000 \leftarrow 1010\ \ 0011\ \ 0001$
$0110$
$= 1001\ \ 0000\ 0011\ \ 0001$
$9\ \ \ \ 0\ \ \ \ 3\ \ \ \ 1$

**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 сложением 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:

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

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:

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

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

$$A = (\alpha_1, \alpha_2, \ldots, \alpha_n), \qquad (3.35)$$

где $\alpha \equiv A (\mathrm{mod}\ p)$.

Обозначим произведение

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

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.

$P=3\cdot5\cdot7 = 105$ .

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.

1 $\equiv 105 \equiv 210 \equiv ... \equiv (\mathrm{mod}\ 3), (\mathrm{mod}\ 15), (\mathrm{mod}\ 7)$;

2 $\equiv 106 \equiv 211 \equiv ... \equiv (\mathrm{mod}\ 3), (\mathrm{mod}\ 5), (\mathrm{mod}\ 7)$

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

$$\gamma_i = \alpha_i + \beta_i - \frac{\alpha_i + \beta_i}{p_i} \cdot p_i, \qquad (3.37)$$

$$\delta_i = \alpha_i \cdot \beta_i - \frac{\alpha_i \cdot \beta_i}{p_i} \cdot p_i. \qquad (3.38)$$

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

$$A = k_i p_i + \alpha_i,\ B = l_i p_i + \beta_i,\ i = 1, 2, \ldots, n, \qquad (3.39)$$

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

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

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

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

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) |

Операция елена, т. к. в СОК отсутствуют отрицательные числа. Однако в частных случаях, когда $A, B, (A–B) \in [0, P)$, можно записать

$$\lambda_i = \alpha_i - \beta_i - \left[\frac{\alpha_i - \beta_i}{p_i}\right] \cdot p_i , \qquad (3.40)$$

$$\lambda_i = (\alpha_i - \beta_i) \pmod{p_i}, \quad i = 1, 2, \ldots n.$$

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 уменьшаемого добавляется соответствующее основание.

**Пример 3.33**

Рассмотрим вычитание *A–B* для случаев *A>B.* Результат— на рис. 3.41.

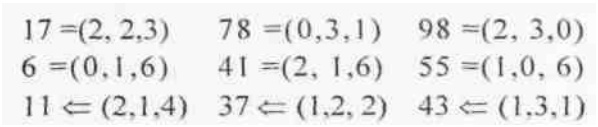



**Рис. 3.41.** Результат выполнения примера 3.33

### 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 $\hbar$ величину

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

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

Примем в качестве нуля число $\hbar$ и будем представлять положительные числ; $N$=|N| в виде $N' = \hbar + |N|$, а отрицательные числа $N = -|N|$ в виде $N' = \hbar - |N|$. Тогда при алгебраическом суммировании получим следующий вид представления положительных и отрицательных чисел:

$$N' = \hbar + N .$$

Это означает, что в принятом представлении мы всегда будем иметь дело с положительными числами, однако числа в искусственной форме $N'$ в интервале $[0, \hbar)$ будут отображать отрицательные числа, а в интервале $[\hbar, P)$ — 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 $F = \{f_1, f_2, \ldots, f_k\}$ 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 *макропрограммой.* 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).

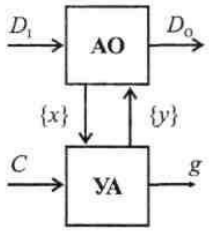



**Рис. 4.1.** Устройство как композиция автоматов

*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 $D_I$, которые в соответствии с алгоритмом операции преобразуются в выходные данные $D_O$ . 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 четыре 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:

- $A = a_0a_1a_2...a_{15}$ —первый операнд (делимое);
- $B = b_0b_1b_2...b_{15}$ —второй операнд (делитель);
- $C=c_0c_1c_2...c_{15}$ — результат операции (частное), в процессе выполнения алгоритма переменная $C$ используется для хранения остатка;
- $D= d_0d_1d_2...d_{15}$ — 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).

$$c_0 = a_0\bar{b}_0 \vee \bar{a}_0b_0. \qquad (4.1)$$

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).

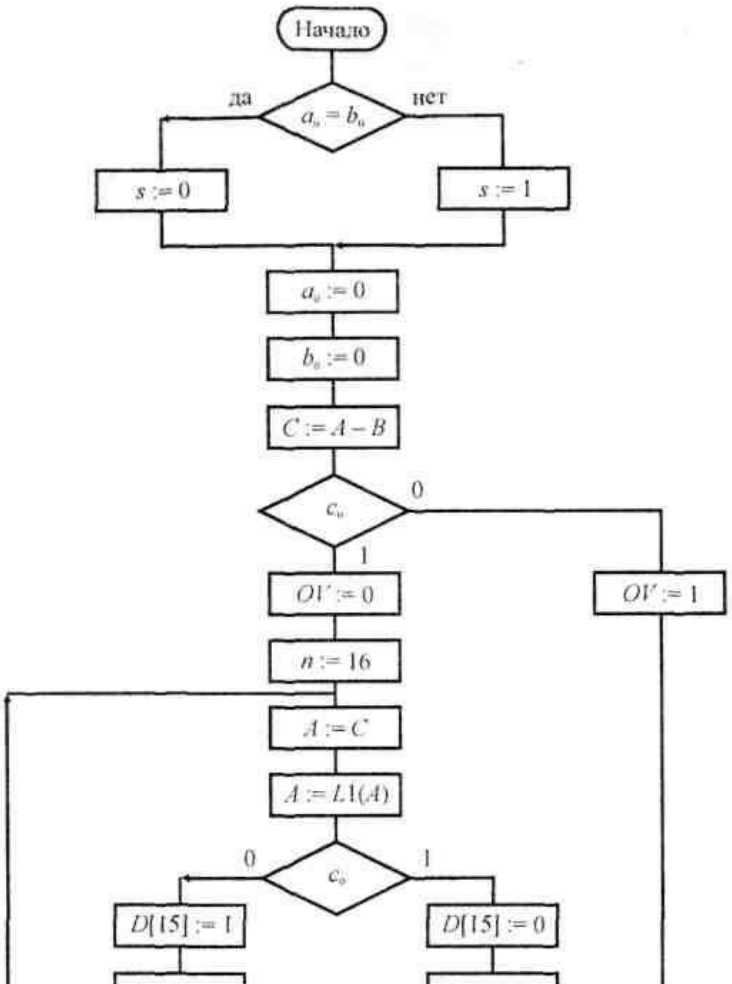


**Рис. 4.3.** Граф-схема алгоритма деления

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.

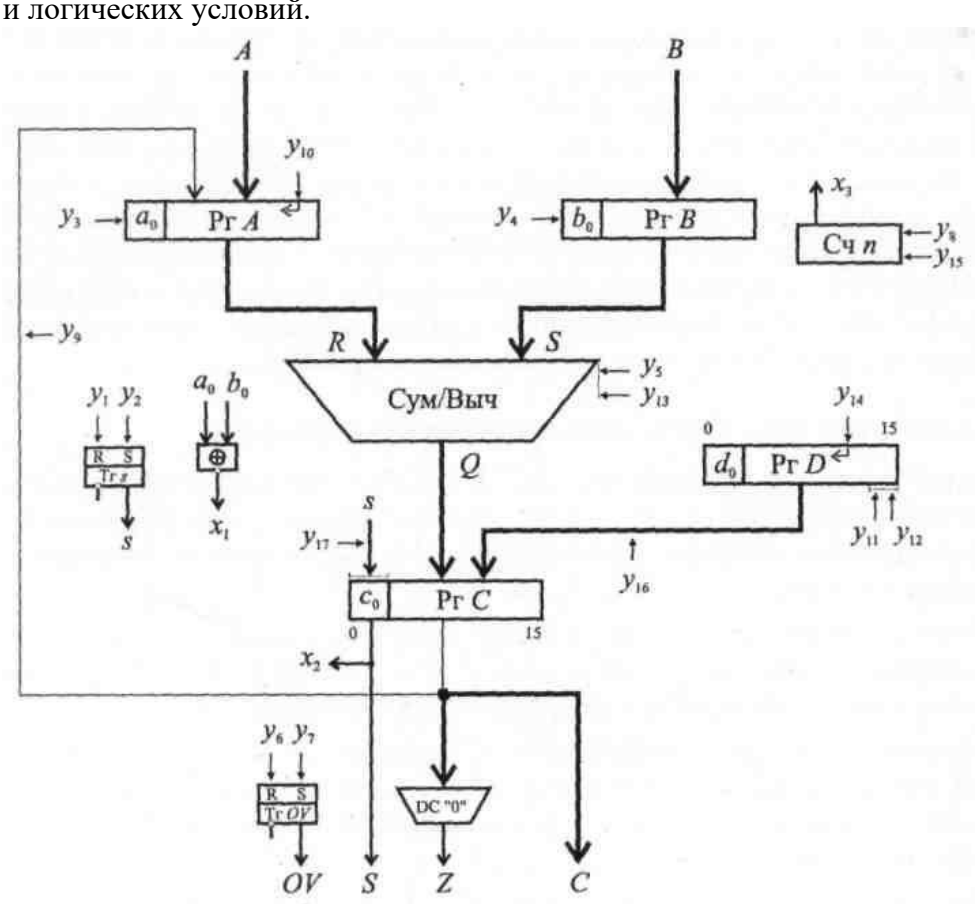



**Рис. 4.4.** Операционный автомат АЛУ

**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 |
|---|---|---|---|---|---|
| $y_4$ | $b_0 := \overline{0}$ | $y_{13}$ | $C := A + B$ | | |
| $y_5$ | $C := R + S$ | $y_{14}$ | $D := LI(D)$ | | |

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 $D := L1(D)$; $n:=n-1$ и некоторых других. В то же время, операторы $A := C$, $A := L1(A)$ нельзя выполнять 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.

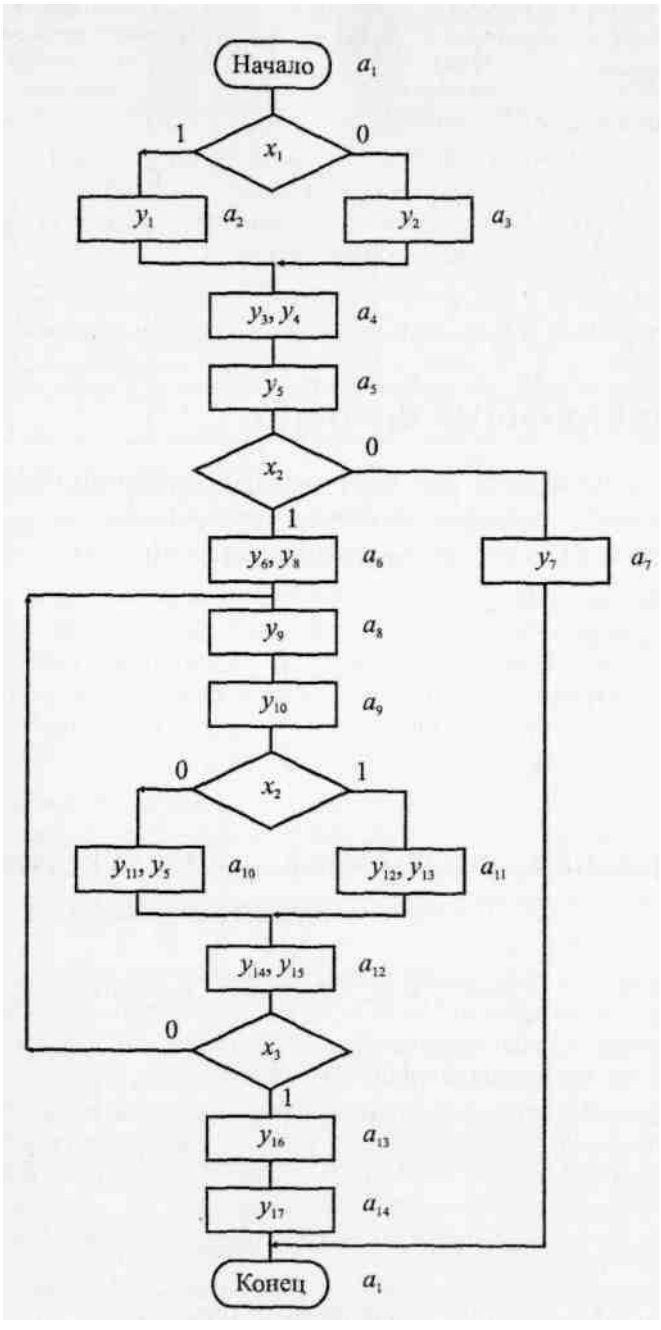

Начало
$a_1$
1
$x_1$
0
$y_1$
$a_2$
$y_2$
$a_3$
$y_3, y_4$
$a_4$
$y_5$
$a_5$
0
$x_2$
1
$y_6, y_8$
$a_6$
$y_7$
$a_7$
$y_9$
$a_8$
$y_{10}$
$a_9$
0
$x_2$
1
$y_{11}, y_5$
$a_{10}$
$y_{12}, y_{13}$
$a_{11}$
$y_{14}, y_{15}$
$a_{12}$
0
$x_3$
1
$y_{16}$
$a_{13}$
$y_{17}$
$a_{14}$
Конец
$a_1$

**Рис. 4.5.** Микропрограмма деления

## 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 микропрограммы 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.

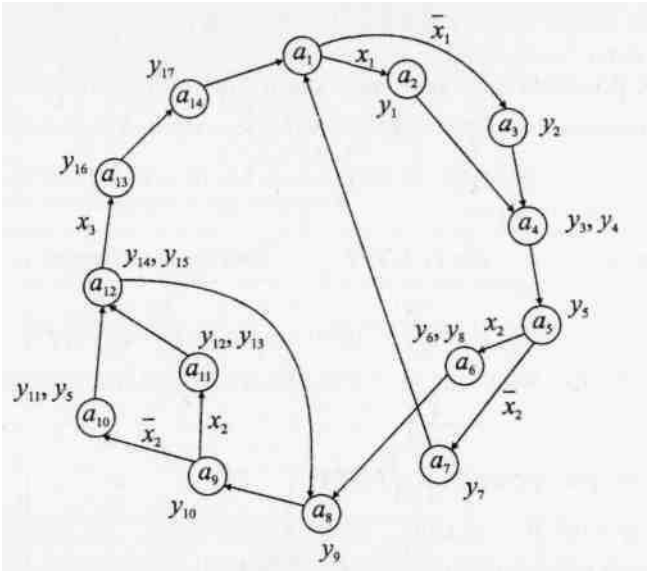



**Рис. 4.6.** Граф микропрограммного автомата Мура

**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 ($24 > 14$ ; $2 < 14$) — 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 Т-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 |

*ов* 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 $\overline{a}_1...\overline{a}_{i-1}a_i\overline{a}_{i+1}...\overline{a}_{14}$. 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]:

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

**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 $y_1$, $y_2$,... $y_{17}$.

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

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 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:

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

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.

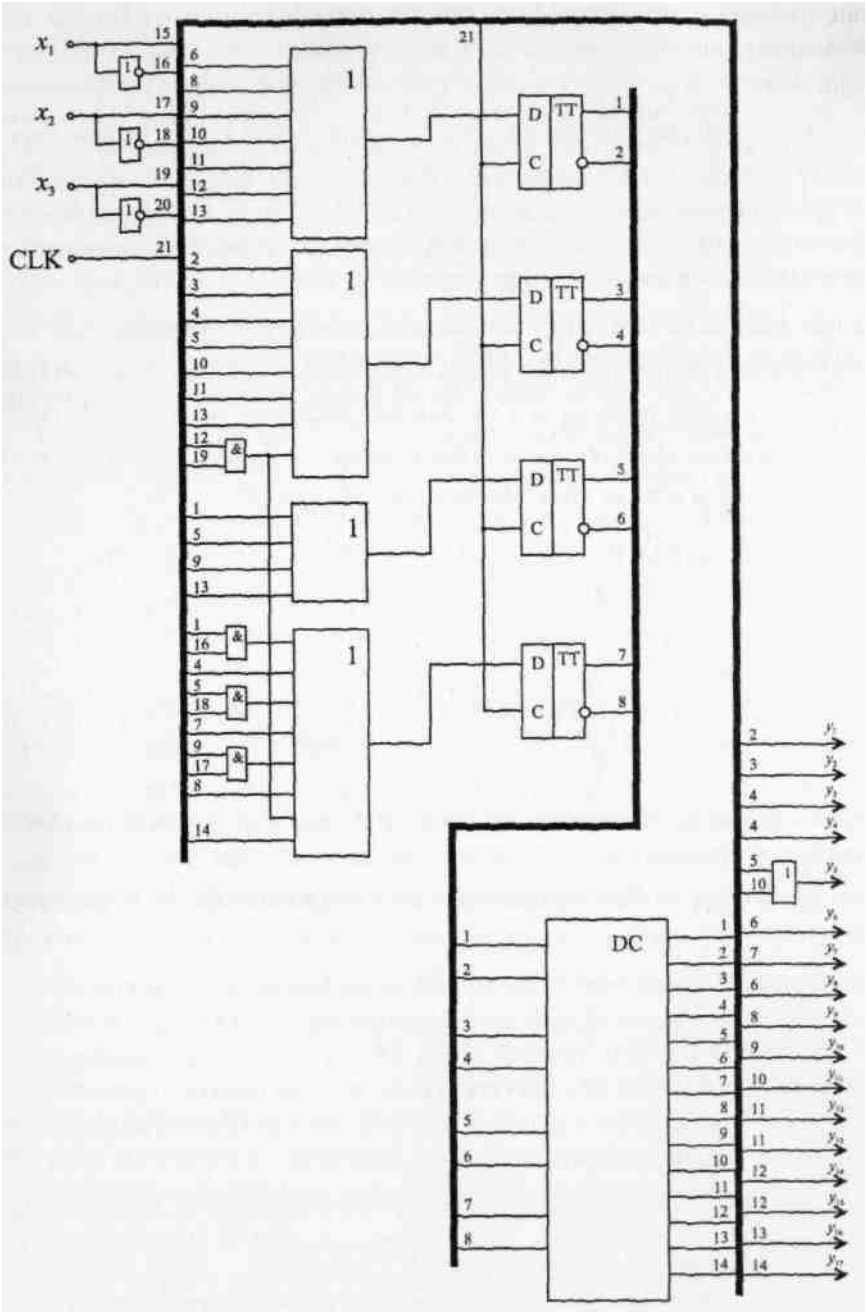



**Рис. 4.7.** Микропрограммный автомат Мура — функциональная схема

## 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 потом 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.*

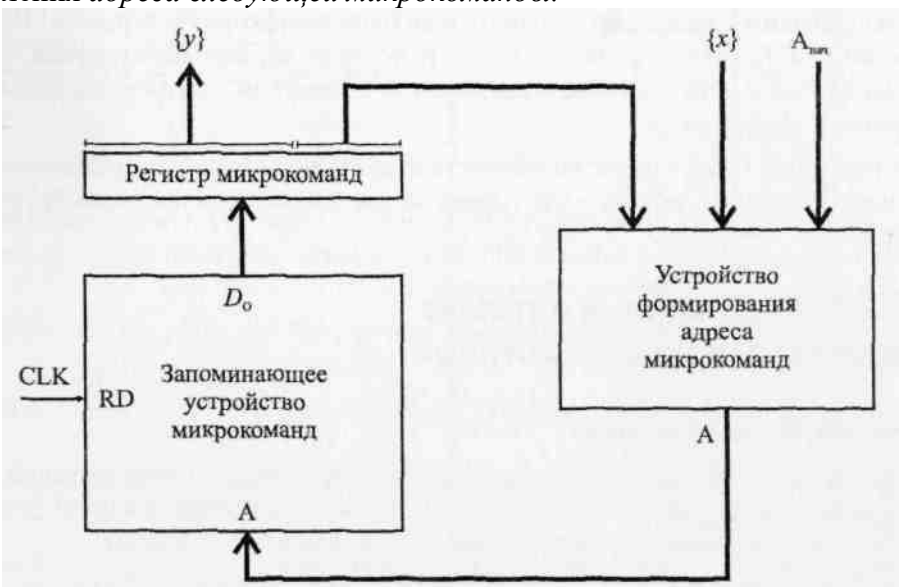



**Рис. 4.8.** Микропрограммный автомат с программируемой логикой

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

**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*).

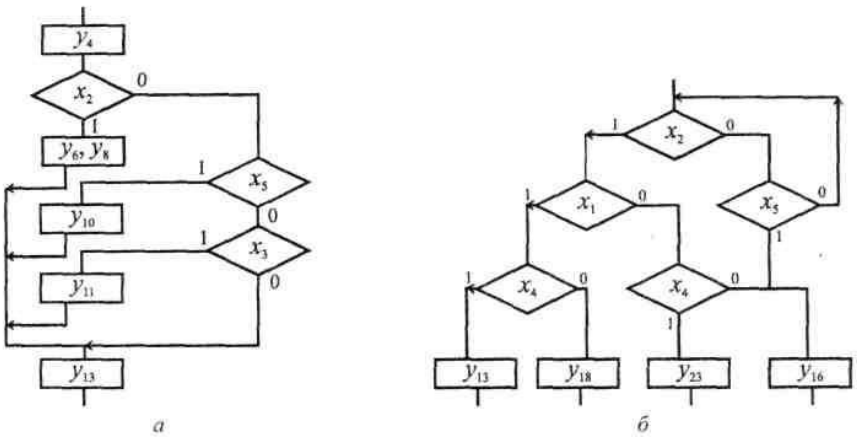
В первом случае осуществляется *безусловный* переход к следующей микрокоманде, и адрес этой *единственной* микрокоманды (A1) должен размещаться в поле адреса выполняемой микрокоманды.

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 А0 микрокоманды, которая будет выполняться, если указанное условие ложно.




*а* *б*

**Рис. 4.9.** Фрагменты ГСА микропрограмм

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

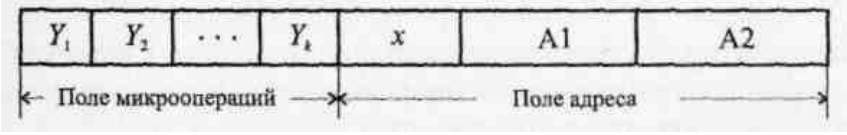



**Рис. 4.10.** Формат микрокоманды при принудительной адресации

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 $y_{13}\ldots y_{20}$. 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.

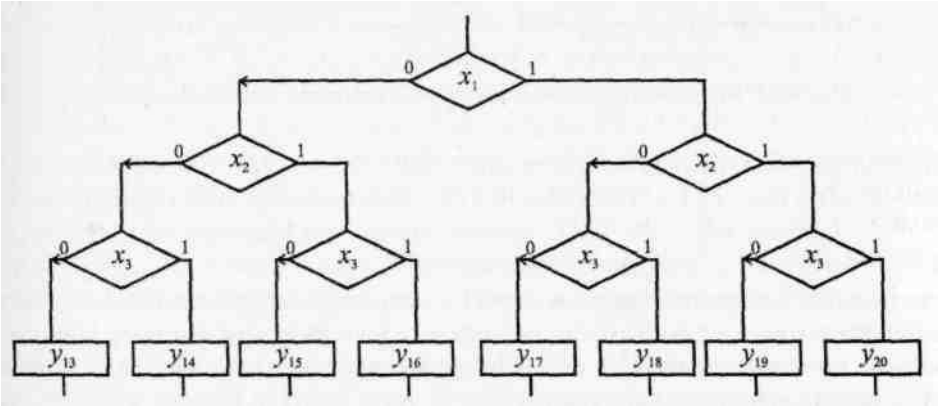



**Рис. 4.11.** Фрагмент ГСА микропрограммы

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 A1, а поля *x* и A0 — 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 A1. Если заданное полем *x* условие истинно, то выполняется микрокоманда по адресу A1, 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.

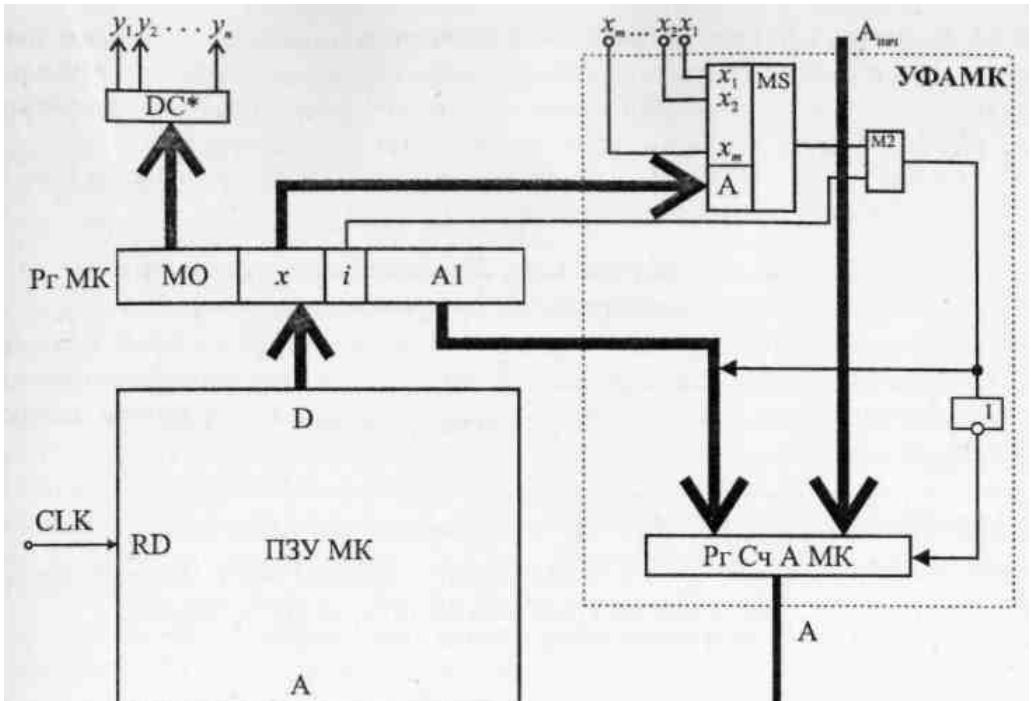


**Рис. 4.13.** Микропрограммный автомат с естественной адресацией

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 А1 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 $Y = \{y_1, y_2, \ldots, y_n\}$.

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 $1 \le s \le n$. 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 $y_i y_i \ldots y_i$, что является явно бессмысленным.

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

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

Каждое подполе поля микроопераций кодирует микрооперации только одного подмножества $Y_i \supset Y$. 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, 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 изображен: 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 *синтоксически правильно* построенная ГСА и требуется разработать устройство реализующее это поведение.

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

Общая структура такого устройства представлена на рис. 4.8. Исходя из описанных выше вариантов организации адресации и способов кодирования пол; микроопераций, выберем *естественную адресацию* и *смешанный способ кодирования* микроопераций. Ограничимся единственным форматом микрокоманды.

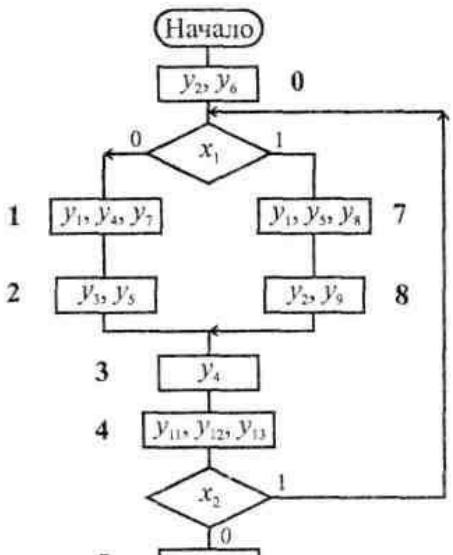


**Рис. 4.14.** Исходная микропрограмма для проектирования автомата

**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.

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

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

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

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 реализации 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.

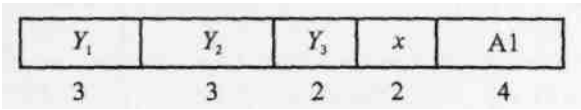



**Рис. 4.15.** Формат микрокоманды

**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 микрокоманде ($y_3$, $y_5$). 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. Поэтому 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).

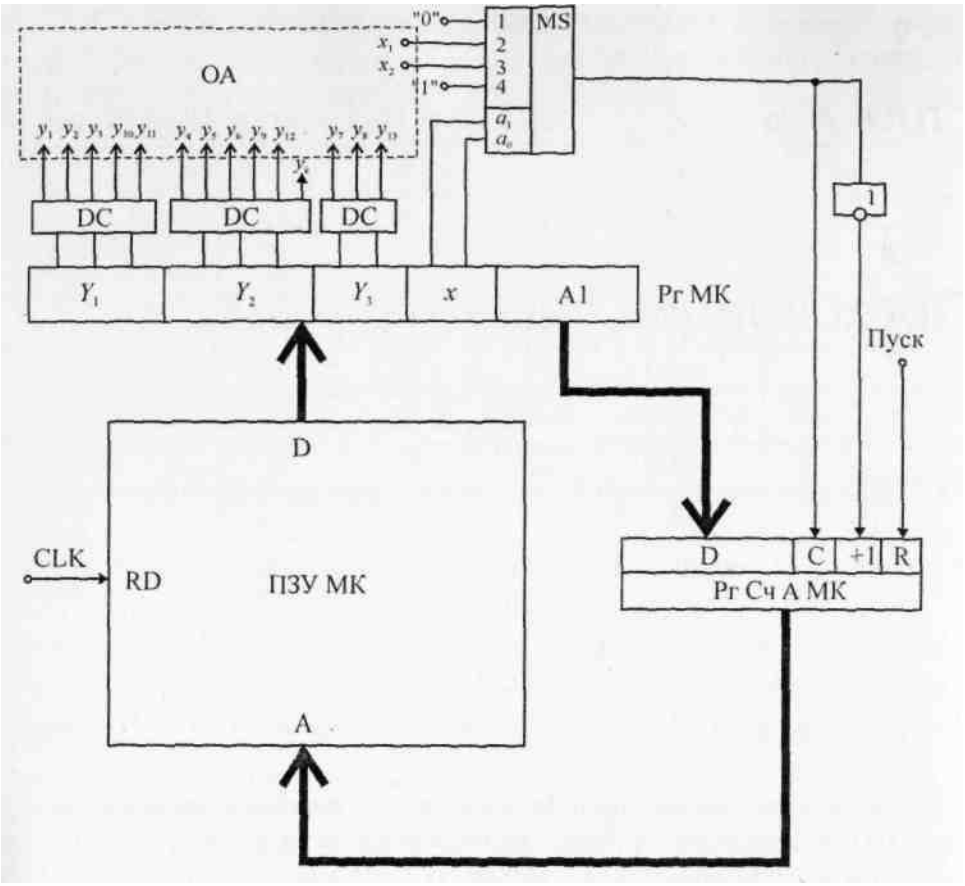



**Рис. 4.16.** Структурная схема управляющего автомата

# 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).

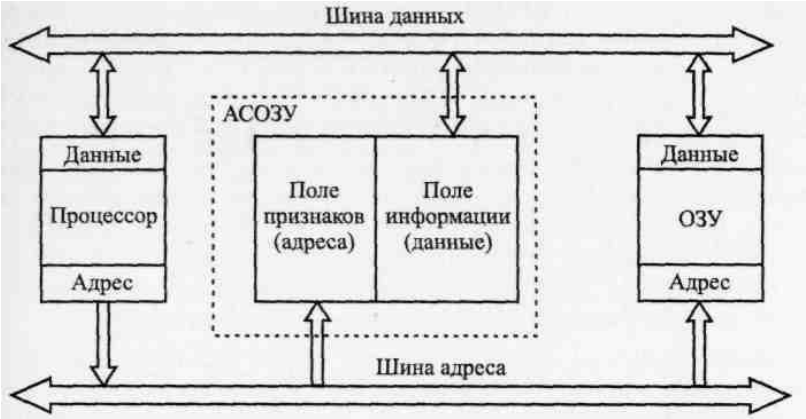



**Рис. 5.2.** СОЗУ с ассоциативным доступом

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. Содержимое 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.

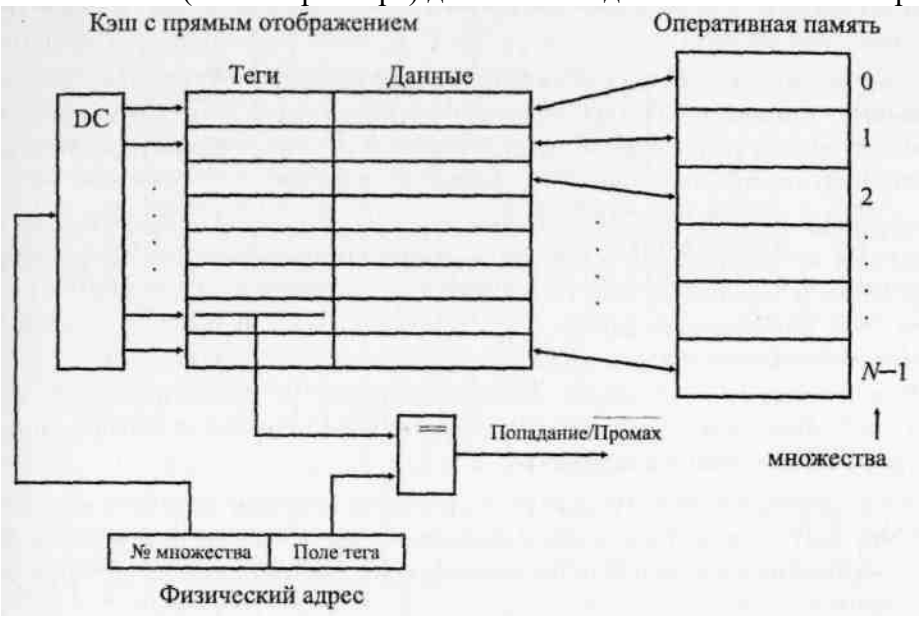



**Рис. 5.3.** Кэш с прямым отображением

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.

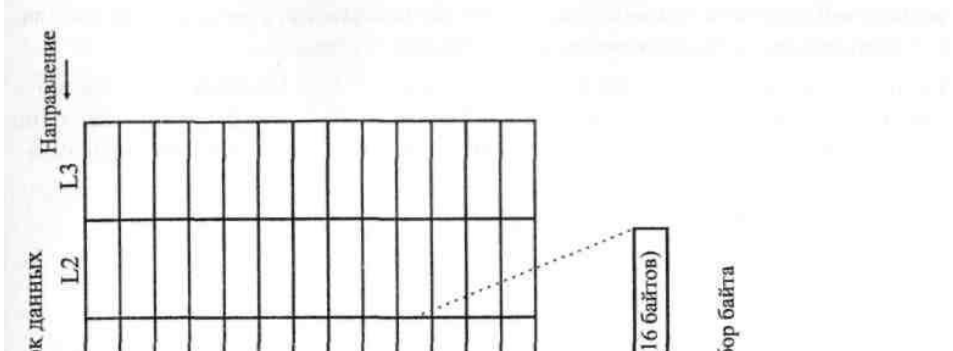
Блок информации из ОЗУ может располагаться в кэш только в одном из 128 множеств, причем в каждом множестве возможно хранение четырех блоков. Адресация кэш осуществляется путем разделения физического адреса на три поля:

- 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-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 BO 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 режим (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 Минховского — 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.