Министерство образования и науки Российской Федерации

Федеральное агентство по образованию

Нижегородский государственный университет им. Н.И. Лобачевского

С.Г. Кузин

ЛОГИЧЕСКИЕ ОСНОВЫ АВТОМАТИЧЕСКОЙ ОБРАБОТКИ ДАННЫХ

Учебное пособие

Рекомендовано Научно-методическим советом по прикладной математике и информатике УМО университетов в качестве учебного пособия для студентов, обучающихся по направлению 510200 – "Прикладная математика и информатика", и специальности 010200 – "Прикладная математика и информатика".

Нижний Новгород
Издательство Нижегородского госуниверситета
2005

УДК 681.3
ББК 32.973
К 21

Научный редактор:
профессор, д.т.н. Ю.Г. Васин
Рецензенты:
профессор, д.т.н. С.И. Ротков; профессор, д.т.н., В.Е. Турлапов

Кузин С.Г. Логические основы автоматической обработки данных. Учебное пособие. Издание второе дополненное и переработанное. Н. Новгород: Изд-во, ННГУ, 2005. 177 с.

В учебное пособие вынесены вопросы, которые в методической литературе для информатиков и прикладных математиков либо не освещаются совсем, либо освещаются поверхностно. Учитывается также преимущественная ориентация читателей на эксплуатацию технических средств вычислительной техники, но не на их разработку. Поэтому объяснение устройства различных блоков компьютера ведется на логическом уровне, не опускаясь до уровня схемотехники.

К учебному пособию может придаваться дискета с мультимедийными анимациями основных алгоритмов функционирования компьютера.

Пособие предназначено студентам и аспирантам, специализирующимся в области информационных систем и прикладной математики.

Предисловие

В настоящее время развитие микроэлектроники привело к массовому использованию микропроцессоров не только в качестве составляющих универсальных вычислительных машин, но также в качестве компонент многих других устройств: сотовые телефоны, устройства передачи данных, самые разнообразные контроллеры и т.д. и т.п.

Сложившаяся ситуация вызывает повышенный спрос специалистов, способных программировать самые разнообразные микропроцессоры. Специфика этой работы, – не только навыки программирования на языках высокого уровня, но также умение программировать непосредственно аппаратуру.

В силу этого, традиционная подготовка программистов на уровне изучения виртуальной машины, реализующей операционную систему и транслятор с языка высокого уровня, явно недостаточна. С другой стороны, изучение одной инженерной конкретики, без понимания фундаментальных основ обработки данных, также не приведет к подготовке квалифицированных специалистов.

Читателю предлагается второе издание учебного пособия, которое преследует цели не столько подробного изложения деталей устройства современных компьютеров, сколько создание квалифицированного представления о фундаментальных основах и методах автоматического преобразования данных.

В отличие от предыдущего, новое издание разбито на две главы: "Фундаментальные принципы автоматической обработки данных" и "Совершенствование архитектуры вычислительной машины". Во второй главе излагаются основные идеи, позволяющие расширить адресное пространство памяти и увеличить быстродействие вычислительной системы

В пособие вынесены вопросы, которые в учебной литературе для информатиков и прикладных математиков либо не освещаются совсем, либо освещаются поверхностно. Учитывается также преимущественная ориентация читателей на эксплуатацию технических средств вычислительной техники, но не на их разработку. Поэтому объяснение устройства и функционирования различных блоков компьютера ведется на логическом уровне, не опускаясь до уровня схемотехники.

Благодарности

Работа подготовлена при поддержке ФЦП "Интеграция" (проект К0392) и гранта поддержки ведущих научных школ РФ № 00-15-9608.

Автор выражает благодарность профессору Стронгину Р.Г., оказавшему большое влияние на содержание и способ изложения материала.

Также автор благодарит Матросову Екатерину, которая реализовала мультимедийные анимации алгоритмов.

Глава I. Фундаментальные принципы автоматической обработки данных

1. Информация и данные

Как в популярной, так и в специальной литературе встречаются две сентенции: компьютер представляет собой устройство для автоматической обработки данных; компьютер представляет собой устройство для автоматической обработки информации. Разберемся подробнее в соотношении понятий "информация" и "данные".

Все, с чем человек имеет дело в реальном мире (предметы, процессы, факты, признаки, свойства и т.п.), мы будем называть **сущностями**. Сущностями могут быть:

- люди, перечисленные в платежной ведомости;
- детали, которые производятся предприятием;
- геометрические фигуры в математических рассуждениях;
- различные процессы и явления;
- характеристики (признаки) людей, деталей, геометрических фигур и т.д.

Таким образом, сущностью называется все, с чем человек имеет дело в реальном или математическом мире.

Сущности, одинаковые с некоторой точки зрения, объединяются в **концептуальные классы**. Например, концептуальный класс ЧЕЛОВЕК; концептуальный класс ДЕТАЛЬ; концептуальный класс ГЕОМЕТРИЧЕСКАЯ ФИГУРА. Сущность, входящую в состав некоторого концептуального класса, будем называть элементом этого концептуального класса.

Информация – мера осведомленности о некотором элементе концептуального класса, представленная в виде данных.

Определение информации как "меры осведомленности" позволяет сравнивать количественно информацию об элементах одного класса. Например, рассмотрим концептуальный класс геометрических фигур – треугольников. Количество информации о треугольнике (осведомленность), если известна его площадь, больше, чем о треугольнике, для которого величина площади неизвестна.

Сказанное выше позволяет определить **цель обработки информации** как увеличение меры осведомленности относительно элемента заданного концептуального класса.

Пусть, например, имеется треугольник, заданный величинами его сторон. Одна из задач обработки информации о данном треугольнике – определение его площади, т.е. увеличение количества информации или полноты знаний об этой геометрической фигуре.

Данные – совокупности символов (конструктивных), используемые для представления информации и организованные в соответствии с формальными правилами и принятыми нормами.

Символ (конструктивный) можно записывать, читать и хранить, а также передавать по линиям связи. По сути дела, конструктивный символ является состоянием некоторого носителя символов. Например, конструктивным символом является буква алфавита, представленная состоянием области бумаги, в которой она записана. Также конструктивным символом является двоичная цифра, представленная состоянием электронного запоминающего устройства

Говоря другими словами, данные представляют собой обозначения объектов и процессов, а также характеристик этих объектов и процессов.

Каждый элемент концептуального класса обладает совокупностью **групповых признаков**, которая является общей для всех элементов этого класса и определяет его принадлежность именно к этому классу. Вместе с тем каждый элемент концептуального класса обладает совокупностью **индивидуальных признаков**, который выделяет конкретный элемент среди других сущностей этого класса.

Например, если групповым является признак: "быть человеком", то индивидуальным может служить признак: "быть Ивановым Иваном Ивановичем".

Информация (осведомленность) о совокупности групповых признаков отождествляется **с дефиницией** концептуального класса. Информация (осведомленность) о совокупности индивидуальных признаков отождествляется с **набором данных** элемента концептуального класса.

Дефиниция концептуального класса представляется в виде совокупности атрибутов – свойств, присущих элементам концептуального класса. Каждый **атрибут** представляет одно какое-либо неотъемлемое свойство элемента и идентифицируется уникальным именем. Таким образом, дефиниция концептуального класса представляется в виде:

$$\mathbf{N \rightarrow S(A_1,A_2,\ldots,A_i,\ldots,A_k);} \qquad \mathbf{(I.1)}$$

где **N** –имя концептуального класса, $\mathbf{A_i}$ – имя атрибута, **S** – совокупность атрибутов.

Примечание. Для одного и того же концептуального класса при решении различных задач можно записывать различные дефиниции, отражающие те его свойства, которые необходимо учитывать при решении каждой конкретной задачи.

Вообще говоря, дефиниция, представляющая групповые признаки концептуального класса, может быть очень большой или даже бесконечной.

Для каждой конкретной прикладной задачи, поставленной для концептуального класса, определяется частная дефиниция, т.е. выбираются лишь некоторые атрибуты, представляющие интерес в процессе решения данной конкретной задачи.

Так, например, для решения прикладной задачи отдела кадров, поставленной для концептуального класса ЧЕЛОВЕК, выбирается частная дефиниция: ФАМИЛИЯ, ГОД_РОЖДЕНИЯ, ПОЛ, ОБРАЗОВАНИЕ, ДОЛЖНОСТЬ, ЗАРПЛАТА. Для решения прикладной задачи спорткомитета, поставленной для того же концептуального класса, может быть выбрана другая частная дефиниция: ФАМИЛИЯ, ГОД_РОЖДЕНИЯ, ПОЛ, ВЕС, РОСТ, ВИД_СПОРТА.

Введение в дефиницию концептуального класса **K** атрибута $\mathbf{A^i}$ подразумевает, что у элементов этого класса существует глобальное свойство, обозначаемое как $\mathbf{A^i}$, и элемент $\xi \in \mathbf{K}$ имеет индивидуальное значение $\mathbf{A^i_\xi}$ этого глобального свойства. Множество всех возможных индивидуальных свойств образует область допустимых значений атрибута $\mathbf{A^i}$. Каждое допустимое значение $\mathbf{A^i_\xi}$ обозначается совокупностью символов $\mathbf{d^i}$, которая образует код значения атрибута - **единицу данных**.

Пример атрибута: РОСТ_ЧЕЛОВЕКА_В_САНТИМЕТРАХ. Один из допустимых кодов значения атрибута (единица данных): 175.

Совокупность индивидуальных признаков объекта (значений всех его атрибутов) образует совокупность единиц данных – **набором данных**.

Пусть совокупность групповых признаков концептуального класса **N** представляется в виде дефиниции: $\mathbf{N \rightarrow S(A_1,A_2,\ldots,A_i,\ldots,A_k)}$. Пусть атрибут $\mathbf{A_i}$ в качестве допустимых кодов значений имеет множество единиц данных $\mathbf{D_i}$. Тогда набор данных, обозначающий элемент χ этого концептуального класса, представляется как: $\mathbf{D(d^{\chi}_1,\ldots,d^{\chi}_i,\ldots,d^{\chi}_k)}$.

Простейшей дефиницией является последовательность. Например, в качестве дефиниции концептуального класса ЧЕЛОВЕК можно записать следующую последовательность атрибутов: ЧЕЛОВЕК → ИМЯ, ФАМИЛИЯ, ГОД_РОЖДЕНИЯ. Набор данных элемента этого концептуального класса может представляться в виде: Человек → Иван, Иванов, 1972. Набор данных другого элемента: Человек → Петр, Петров, 1975.

Из сказанного следует что, информацию об элементе концептуального класса мы отождествляем:

- с дефиницией, т.е. совокупностью атрибутов, определяющих этот концептуальный класс;
- с единицей данных, т.е. определенностью значения каждого атрибута для конкретного элемента концептуального класса;
- с набором данных, т.е. определенностью совокупности единиц данных для конкретного элемента концептуального класса.

Заметим, что атрибуты объекта по типу значений, подразделяются на числовые (количественные, порядковые) и качественные.

В случае количественного атрибута существует эталон измерения величины его значения, и код значения атрибута (единица данных) являет собой запись числа - результат измерения (с указанием явно или по умолчанию единицы измерения). Например, атрибут: РОСТ ЧЕЛОВЕКА.

В случае порядкового атрибута код значения атрибута – запись порядкового числа. Например, атрибут: МЕСТО, ЗАНЯТОЕ НА СОРЕВНОВАНИИ.

Особо отметим специфику образования кода значения числового атрибута (записи числа). Запись числа образуется из символов - цифр и, с одной стороны, является уникальным обозначением числа. С другой стороны позиционная запись числа является самоинтерпретирующейся, т.е. несет в себе информацию о величине числа. Это свойство позволяет определить арифметические действия над числами в виде преобразования записей (кодов) чисел.

Человек, общающийся с компьютером, имеет дело с тремя мирами и двумя уровнями представления данных.

- Реальный мир, который существует вне зависимости от владельца компьютера, и информацию о котором человек желает хранить и обрабатывать.
- Модельный мир – **внешние наборы данных** как представление информации об объектах реального мира в виде моделей. Под моделью (математической, кибернетической, информационной) подразумевается представление информации об объектах реального мира посредством удобных для человека формализмов, отражающих необходимые характеристики этих объектов;
- Компьютерный мир – **машинные наборы данных** как представление информации о моделях в виде, эффективном с точки зрения ее обработки компьютером.

Прикладной математик пытается отобразить реальный мир, определяя необходимые множества данных, а также устанавливая необходимые отношения между данными в этом множестве. Тем самым реальный мир заменяется модельным миром данных. При изучении явлений, предметов, процессов внешнего мира с помощью компьютера определяются структуры хранения данных и алгоритмы преобразования данных. Тем самым модельный мир заменяется компьютерным миром.

Понимая под единицей данных код значения атрибута, необходимо учитывать "потребителя" этого обозначения. Дело в том, что в большинстве случаев существует такая закономерность: обозначение, удобное для человека в модельном мире (внешние данные), неэффективно с точки зрения его автоматической обработки аппаратурой в компьютерном мире (внутренние данные).

Например, рисунок графа легко воспринимается человеком, тогда как для обработки графа целесообразнее представлять его в виде матрицы инциденций или в виде связного списка.

Сказанное выше определяет **двухуровневое представление данных** в человеко-машинных системах обработки данных. Единицы данных (наборы данных) в модельном мире представляются лексемами (предложениями) языков описания данных, ориентированными на восприятие этих лексем (предложений) человеком. Примерами лексем служат: запись вещественного числа в десятичной позиционной системе счисления; последовательность букв и цифр, начинающаяся с буквы – идентификатор языка программирования. Из лексем строятся наборы данных – предложения языка описания данных, которые представляет собой строку символов, построенную в соответствии с правилами синтаксиса этого языка. Например, арифметическое выражение.

С точки зрения компьютера, программа представляет собой также набор данных, который интерпретируется как задание на обработку данных. И здесь также имеем два уровня представления программы: программа как предложение, записанное на языке высокого уровня, и программа как предложение двоичного языка программирования.

Представление данных, удобное и понятное с точки зрения человека, практически всегда неэффективно с точки зрения их обработки устройствами вычислительной машины. Поэтому в памяти компьютера данные представляется посредством структур хранения различной сложности.

Пример. Внешнее представление единицы данных типа "вещественное число": -12.E+3 . Внутреннее представление этой же единицы данных на Рис. I.1:

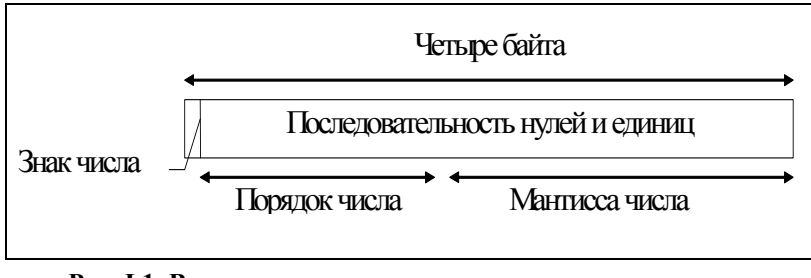


Рис. I.1. Внутреннее представление вещественного числа

Пример. Внешнее представление единицы данных типа "дерево": A(B(D)(E))(C(F)(G)). Внутреннее представление дерева в виде списка на Рис. I.2

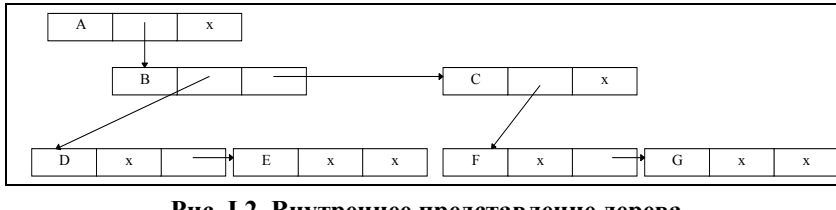


Рис. I.2. Внутреннее представление дерева

Важное замечание. Как внешнее, так и внутреннее представление единицы данных являют собой последовательности символов. Отличие первого от второго заключается в алфавите используемых символов и в способе образования строки. Если внешнее представление может содержать достаточно произвольные символы и строится по «человеческим» законам, то внутреннее представление состоит только из нулей и единиц и строится по «компьютерным» законам. То же можно сказать относительно внешнего и внутреннего представлений набора данных

В модельном мире код значения числового атрибута представляется записью числа (десятичной константой языка описания чисел), а в компьютерном мире - кодом числа (двоичной константой структуры хранения чисел). Например, 125 - запись числа, 1111101 - код числа.

В случае качественного атрибута, множество допустимых его значений образует неметрическое множество свойств объекта. Например, атрибут ЦВЕТ. В модельном мире значением качественного атрибута служит запись (имя) свойства объекта, в компьютерном мире - двоичный код, обозначающий то же свойство. Например, BLUE- запись языка описания цвета, 01000010010011000101010101000101 - двоичный код цвета.

Теперь мы можем утверждать что:

- информация об объектах реального мира представляется нам в виде данных;
- данные - это обозначения характеристик объекта, которые хранятся, передаются по линиям связи на расстояние и обрабатываются аппаратными средствами;
- данные возникают в процессе выделения и анализа свойств сущности. Частный случай получения данных – количественные измерения.

Следует выделить три проблемы, связанные с использованием данных в человеческой деятельности.

Проблема 1. Передача информации на расстояние. Для конкретного объекта выделяется совокупность атрибутов и обозначения характеристик этих атрибутов (данные) передаются от передатчика к приемнику по линиям связи. При этом обращается внимание на скорость и безошибочность передачи данных.

Проблема 2. Хранение информации. Для конкретного объекта выделяется совокупность атрибутов и коды характеристик этих атрибутов (данные) представляются одним из возможных состояний носителя информации (бумага, память компьютера). При этом обращается внимание на надежность и стоимость хранения информации.

Проблема 3. Обработка информации. Для конкретного объекта выделяется совокупность атрибутов, среди которой выделяются первичные атрибуты (величины которых известны априори) и вторичные атрибуты (величины которых необходимо определить в процессе обработки информации). Разрабатывается алгоритм, преобразующий значения первичных атрибутов (данных) в значения вторичных атрибутов (данных).

Таким образом, фундаментальное понятие информации как совокупности представлений человека о некотором объекте концептуального класса на практике заменяется конструктивным понятием набора данных. При передаче информации по линиям связи пересылаются наборы данных. В памяти компьютера хранятся наборы данных. При обработке информации исходные наборы данных (обозначения априори известных величин первичных атрибутов) преобразуются в результирующие наборы данных (обозначения априори неизвестных величин вторичных атрибутов).

Подводя итог, можно сказать, что данные - это представление явлений внешнего мира (объектов, процессов, фактов, признаков, свойств и т.п.) в виде условных обозначений, удобных для истолкования и обработки человеком или компьютером.

Отсюда возникает еще одна проблема, связанная с использованием информации в человеческой деятельности.

Проблема 4. Кодирование информации: конструирование формальных языков и формальных кодов, обозначающих характеристики объектов реального мира; создание наборов данных для обозначения конкретных объектов реального мира с целью обработки информации об этих объектах.

1.1. Контрольные вопросы

1. Понятие концептуального класса и элемента концептуального класса.
2. Основная цель обработки информации.
3. Атрибуты и коды значений атрибутов.
4. Понятие единицы данных и набора данных.
5. Внешнее и внутреннее представление данных.
6. Проблемы использования информации в человеческой деятельности.

2. Представление целых неотрицательных чисел

2.1. Проблемы представления целых неотрицательных чисел

Рассмотрим представление простых единиц данных, т.е. таких единиц, каждая из которых при обработке рассматривается как единое неделимое целое. Самая простейшая из них – запись целого неотрицательного числа (целого числа без знака).

Целое число без знака определяется как характеристика (мера) количества элементов в множестве. Существует еще одно определение целого без знака - характеристика порядка расположенных в ряд элементов.

Определена **алгебра целых чисел без знака** (целых неотрицательных чисел):

$$\mathbf{A_{цбз} = <N^+; \oplus, \otimes>} \qquad \mathbf{(I.2)}$$

Здесь, $\mathbf{N^+}$ - основное множество алгебры (множество целых неотрицательных чисел); $\oplus$ и $\otimes$- операции сложения и умножения соответственно.

Алгебра $\mathbf{A_{цбз}}$ замкнута относительно операций сложения и умножения. Т.е. результат такой операции над двумя целыми неотрицательными числами также является целым неотрицательным числом.

Понятие числа является абстрактным понятием. Для того чтобы иметь возможность пересылать числа по линиям связи и выполнять над числами арифметические операции, каждое число должно быть обозначено совокупностью конструктивных символов. Такое обозначение называется **записью или кодом** числа.

Замечание. Обычно не производят разделения между абстрактным числом и его записью (обозначением, кодом) и говорят "число" понимая под этим термином "запись числа".

Рассмотрим проблему создания системы обозначений целого числа без знака. При создании такой системы необходимо выполнить два условия:

- множество целых чисел без знака бесконечно и, следовательно, необходимо придумать также бесконечное число уникальных символьных записей (обозначений, кодов), позволяющих отличать числа друг от друга;
- как говорилось ранее, записи (обозначения, коды) должны быть не только уникальными, но также допускать достаточно простые алгоритмы выполнения арифметических операций как действий по преобразованию символьных обозначений.

Говоря другими словами, обозначение числа должно нести в себе информацию о его величине, т.е. обозначение числа должно быть самоинтерпретирующимся.

Если первое положение очевидно, то второе иллюстрируется на примере сложения (Рис. I.3).

На множестве абстрактных чисел определена абстрактная операция ⊕ сложения. Она отображает два числа в результат. Два операнда операции обозначаются (кодируются), в результате чего получается запись1 и запись 2. Алгоритм сложения преобразует две записи – операнды в запись результат. При этом соблюдается условие: запись результат отображается (декодируется) в абстрактное число, которое равно результату сложения абстрактных чисел.

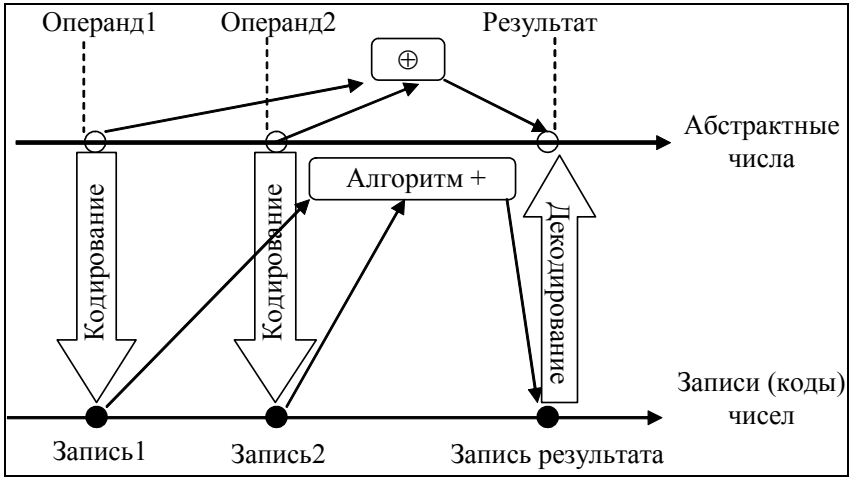


Рис. I.3. Абстрактная операция и алгоритм сложения

В известной римской системе обозначения целых чисел без знака нарушаются оба условия. По этой причине римские числа используются лишь в качестве обозначений (например, номера глав книги), но не используются при вычислениях. Человечество использует также естественно-языковые уникальные обозначения чисел - числительные. В этом случае определены правила образования обозначения любого числа. Но в силу отсутствия алгоритмов выполнения арифметических операций, числительные также используются лишь как идентификаторы.

2.2. Примитивная система представления целых без знака

Предположим, что мы работаем на складе пиломатериалов, и в нашу задачу входит не только сгружать привозимые доски, но считать их количество.

Распространенный прием: сгрузив очередную доску, ставить в записной книжке «палочку».

Таким образом, последовательность «палочек» правомерно рассматривать как обозначение целого неотрицательного числа (Рис. I.4). Будем называть такое обозначение **примитивной записью числа**.

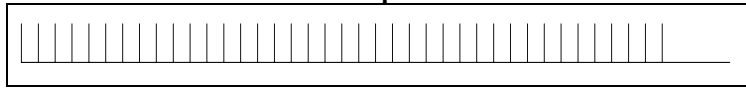


Рис. I.4. Примитивная запись целого без знака

Самое интересное, что для примитивных записей существуют простые алгоритмы сложения и умножения. Так, для того, чтобы сложить два числа, следует ‘склеить’ две примитивных записи, обозначающие эти числа.

В результате получится примитивная запись суммы двух чисел. Нетрудно придумать простой алгоритм умножения целых без знака, представленных примитивными записями.

Таким образом, мы удовлетворили двум условиям представления целых без знака: придумали уникальные обозначения для каждого числа, допускающие простые алгоритмы исполнения операций алгебры этих чисел.

2.3. Позиционная система обозначения целых без знака

Придуманная нами примитивная система обозначения целых без знака страдает одним существенным недостатком: очень большое число знаков (единиц) в записи числа. Причем, длина записи пропорциональна величине числа.

Каждый, наверное, занимался усовершенствованием системы примитивных записей целых без знака. Для этого, последовательность единиц группировалась в десятки, десятки группировались в сотни и т.д. (Рис. I.5)

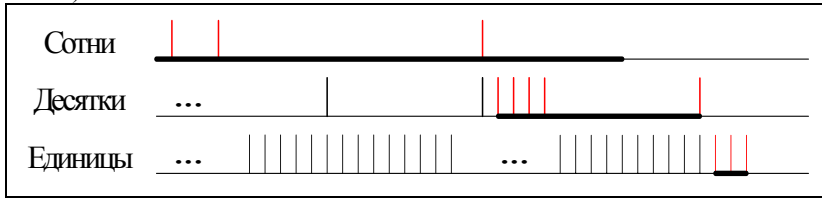


Рис. I.5. Переход к позиционной системе представления целых без знака

Проделав такое позиционирование, мы можем представить число в следующем виде:

$$...((d_3*10+d_2)*10+d_1)*10+d_0 \qquad (I.3)$$

Здесь, $\mathbf{d_3,d_2,d_1,d_0}$ - полное число тысяч, сотен, десятков и единиц соответственно.

Мы перешли к алфавитной позиционной системе счисления - алфавитной системе кодирования целого числа без знака, предложенной, как считается, арабами и основанной на следующих принципах:

- вводится конечное, относительно небольшое число знаков - **алфавит цифр**;
- каждая линейная последовательность цифр образует **запись** (обозначение) некоторого числа.

Используя конечный алфавит цифр, можно получить бесконечное количество различных записей, каждая из которых обозначает одно и только одно целое число без знака.

Однако не следует забывать, что эти цепочки должны быть самоинтерпретирующимися, т.е. нести в себе информацию о величине числа. Этому условию удовлетворяет алфавитная позиционная система обозначения (кодирования) целых чисел без знака. Ее также называют **позиционной системой счисления**.

Прежде всего, выделим два целых числа без знака, которые играют особую роль: число, обозначаемое как **0**, характеризует отсутствие элементов в множестве; число, обозначаемое как **1**, характеризует минимально - возможное увеличение или уменьшение числа элементов в множестве.

Концепция позиционной **B**-ичной системы кодирования (счисления) целых чисел без знака заключается в четырех пунктах.

1. Выбирается произвольное, большее единицы, число, которое называется **основанием** системы счисления и обозначается как **B**. Числа меньшие **B** образуют **базу** системы счисления.
2. Для чисел базы выбираются уникальные обозначения, которые называются **цифрами**.
3. Любое целое число без знака единственным образом представляется с помощью чисел базы системы счисления и операций сложения - умножения в виде **канонического арифметического выражения:**

$$\mathbf{ЗНАЧЕНИЕ_в(цбз)=(d_{n-1}*B^{n-1}+d_{n-2}*B^{n-2}+...+d_1*B^1+d_0*B^0)_в.} \quad \mathbf{(I.4)}$$

4. Последовательность коэффициентов канонического арифметического выражения также единственным образом характеризует величину числа и называется **записью числа** в **B**-ичной системе кодирования (счисления):

$$\mathbf{ЗАПИСЬ_в(цбз) = (d_{n-1}d_{n-2}...d_1d_0)_в} \quad \mathbf{(I.5)}$$

При этом, о коэффициенте $\mathbf{d_i}$ принято говорить как об **i**-м **разряде числа**, а об **n** - как о количестве разрядов числа. Диапазон чисел **0÷n-1** называется **разрядной сеткой числа (**Рис. I.6**).**

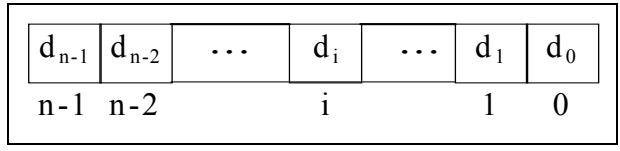


Рис. I.6 . Разрядная сетка числа

Примечание. Подобная система обозначений называется позиционной, потому что в каноническом арифметическом выражении каждое число базы, представленное цифрой, вносит в значение числа вклад, пропорциональный позиции, в которой эта цифра располагается.

В практике вычислений, именно записи используется для обозначения чисел.

Таблица I.1. Запись числа и каноническое выражение

Система счисления	Запись	Каноническое выражение
10	$[125]_{10}$	$[1*10^2+2*10^1+5*10^0]_{10}$
8	$[125]_8$	$[1*8^2+2*8^1+5*8^0]_8$

Очевидно, что одно и то же число может быть представлено в различных системах счисления. В модельном мире человек использует десятичную позиционную систему счисления. В компьютерном мире используется двоичная позиционная система счисления. Кроме того, используются вспомогательные восьмеричная и шестнадцатеричная системы счисления.

Десятичная система счисления также используется для вычислений при переводе одного и того же числа из одной системы счисления в другую. Прежде всего, она используется для представления оснований различных систем счисления:

$$[10]_2 = 2,\ [10]_8 = 8,\ [10]_{16} = 16. \qquad (I.6)$$

Существует формула, позволяющая определить число **q** различных **n** - разрядных чисел в системе счисления с основанием **B**:

$$q = B^n. \qquad (I.7)$$

Для того чтобы все эти числа выписать, следует воспользоваться простым правилом:

- образуется таблица, состоящая из **n** столбцов и B^n строк;
- столбцы нумеруются справа - налево: **n-1**, **n-2**,...,**1**, **0**;
- в столбце **0** записывается последовательность: **010101**....;
- в столбце **1** записывается последовательность: **00110011**...;
- в столбце 2 записывается последовательность:
- **0000111100001111**...;
- и т.д.

Так, например, имея в своем распоряжении 3 двоичных разряда, можно записать в них $2^3 = 8$ различных двоичных числа.

Таблица I.2. Трехразрядные двоичные числа

разряды числа		
2	1	0
0	0	0
0	0	1
0	1	0
0	1	1
1	0	0
1	0	1
1	1	0
1	1	1

Ниже приведена таблица цифр (заштрихованы) и записей чисел типа целое без знака в различных системах счисления.

Таблица I.3. Различные системы счисления

	Основание системы счисления			
Число	10	2	8	16
Нуль	0	0	0	0
Один	1	1	1	1
Два	2	10	2	2
Три	3	0011	3	3
Четыре	4	0100	4	4
Пять	5	0101	5	5
Шесть	6	0110	6	6
Семь	7	0111	7	7
Восемь	8	1000	10	8
Девять	9	1001	11	9
Десять	10	1010	12	A
Одиннадцать	11	1011	13	B
Двенадцать	12	1100	14	C
Тринадцать	13	1101	15	D
Четырнадцать	14	1110	16	E
Пятнадцать	15	1111	17	F
Шестнадцать	16	10000	20	10

Любая позиционная система счисления (кодирования) решает проблему обозначения целых чисел без знака:

- любое целое без знака может быть обозначено последовательностью цифр - записью числа;
- записи (обозначения) самоинтерпретирующиеся, т.е. допускают простые алгоритмы выполнения арифметических операций: ОБОЗНАЧЕНИЕ (первое слагаемое) + ОБОЗНАЧЕНИЕ(второе слагаемое) = ОБОЗНАЧЕНИЕ (сумма двух слагаемых).

Позиционное представление целого числа без знака основано на том, что значение любого такого числа может быть однозначно выражено через значение чисел базы и значение основания системы счисления. При таком выражении используются операции алгебры целых неотрицательных чисел. Таким образом, значение любого числа представляется в виде канонического арифметического выражения (арифметического выражения специфического вида - Рис. I.7)

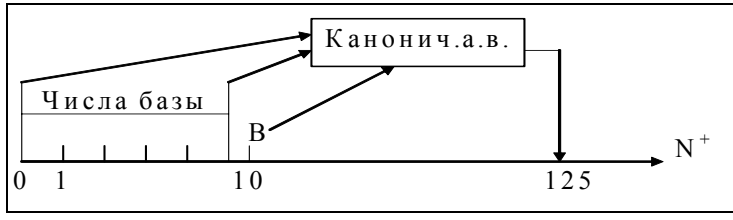


Рис. I.7. Каноническое арифметическое выражение

Можно также определить **B**-ично десятичную систему обозначения чисел. В этом случае цифры **B**-ичной системы представляются десятичными числами.

Например: $(AF7)_{16} \rightarrow (10,15,7)_{16\text{-}10}$. Следует иметь в виду, что простых алгоритмов выполнения арифметических операций для такой системы не существует

2.4. Перевод записи числа из одной системы счисления в другую

В практике компьютеризации используются две основные системы счисления (обозначения) целых чисел без знака: десятичная и двоичная. Существуют достаточно простые алгоритмы перевода записи числа из одной системы счисления в другую.

Алгоритм I.1. Перевод B → 10.

Перевод **B**-ичной записи числа в десятичную запись числа.

Для получения записи $\mathbf{(d)_{10}}$ числа **d** из записи $\mathbf{(d)_B}$, следует:

1. Построить в системе с основанием **B** соответствующее каноническое выражение $\mathbf{[e]_B}$, представляя основание как **10**;
2. Представляя элементы этого канонического выражения $\mathbf{[e]_B}$ в десятичной системе, перевести его в неканоническое выражение $\mathbf{(e)_{10}}$ десятичной системы;
3. Вычислить неканоническое выражение $\mathbf{(e)_{10}}$ и получить десятичную запись числа $\mathbf{(d)_{10}}$.

Пример.

Перевод (запись числа)$_8$ → (запись числа)$_{10}$;

$(d)_8$: запись числа в 8-й системе: $(125)_8$

$[e]_8$: каноническое выражение числа: $[1*10^2+2*10^1+5*10^0]_8$

$(e)_{10}$: выражение числа в 10-й системе: $(1*8^2+2*8^1+5*8^0)_{10}$

(d)10: запись числа в 10-й системе: $(85)_{10}$

Пример.

Перевод (запись числа)$_{16}$ → (запись числа)$_{10}$

$(d)_{16}$: запись в 16-й системе: $(125)_{16}$

$[e]_{16}$: каноническое выражение числа $[1*10^2+2*10^1+5*10^0]_{16}$

$(e)_{10}$: выражение числа в десятичной системе: $(1*16^2+2*16^1+5*16^0)_{10}$

$(d)_{10}$: запись в десятичной системе: $(293)_{10}$

Пример.

$(125)_2$ - недопустимая запись двоичного числа

Пример.

$(253)_{10} = (FD)_{16} = (375)_8 = (11111101)_2$

Примечание. Для выполнения перевода из одной системы счисления в другую полезно помнить десятичные записи степени числа 2.

0	1	2	3	4	5	6	7	8	9	10
1	2	4	8	16	32	64	128	256	512	1024

Алгоритм I.2 . Перевод 10 → B.

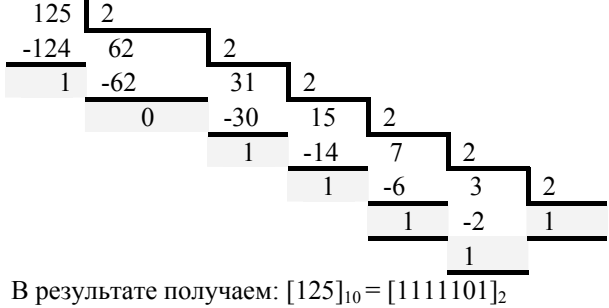
Перевод десятичной записи числа в **B**-ичную запись числа. Используются операции целочисленного деления:

X mod Y - результатом является остаток от деления X на Y;

X div Y - результатом является целая часть от деления X на Y. Вычисления выполняются в десятичной системе.

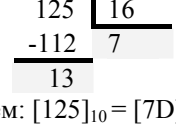
В результате получается запись числа в системе с основанием **B**, записанная в обратном порядке (Рис. I.8).

Пример. Перевод (запись числа)$_{10}$ → (запись числа)$_2$



В результате получаем: $[125]_{10} = [1111101]_2$

Пример. Перевод (запись числа)$_{10}$ → (запись числа)$_{16}$



В результате получаем: $[125]_{10} = [7D]_{16}$

Несмотря на достаточную простоту алгоритмов перевода чисел из одной системы кодирования в другую, “ручное” их выполнение при вводе (выводе) данных в память (из памяти) компьютера неприемлемо для человека. Поэтому используются программы, выполняющие эти действия без вмешательства человека. И рядовой пользователь компьютера, вводя и читая десятичные числа с использованием экрана дисплея, может даже не подозревать о существовании двоичной системы кодирования этих чисел.

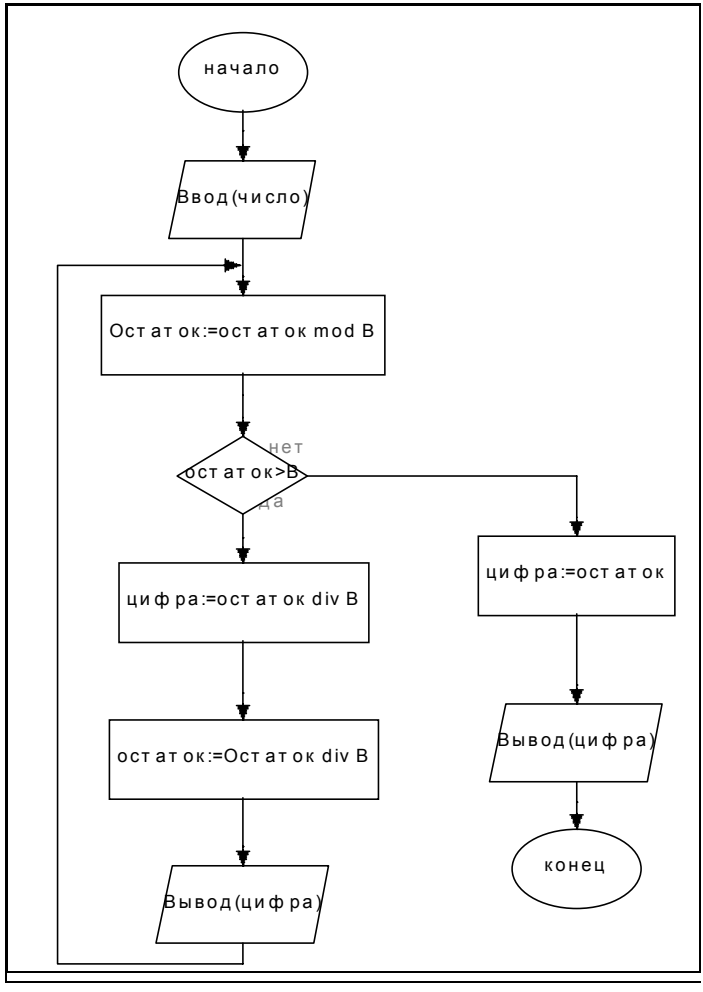


Рис. I.8. Перевод десятичной записи числа в B-ичную запись того же числа

2.5. Перевод записи числа из одной системы в другую, если основания систем кратны двум

Не вдаваясь в подробности, заметим, что десятичная система представления чисел удобна для человека, а двоичная - эффективна с точки зрения реализации памяти компьютера. В дальнейшем мы увидим, что двоичное представление данных, также эффективно с точки зрения реализации электронных схем компьютерных вычислений.

Однако существует закономерность: чем меньше основание системы счисления, тем больше цифр требуется для записи одного и того же числа. Поэтому, специалисты, работающие с компьютером на уровне двоичных кодов, предпочитают записывать числа в восьмеричной или шестнадцатеричной системах счисления.

Следует отчетливо понимать, что это вспомогательные системы представления чисел, удобные для человека в силу своей компактности. Оправданность их использования следует из того, что алгоритмы перевода записи числа из одной системы счисления в другую чрезвычайно просты, если основания этих систем кратны двойке.

Алгоритм I.3. Перевод (запись числа)$_2$ → (запись числа)$_8$.

Двоичная запись числа разбивается справа - налево на триады (каждая по три разряда). Если в самой левой триаде меньше трех цифр – слева добавляются нули. Трехразрядное двоичное число каждой триады заменяется восьмеричной цифрой в соответствии с Таблица I.3

Алгоритм I.4. Перевод (запись числа)$_8$ → (запись числа)$_2$

Каждая цифра восьмеричной записи числа заменяется соответствующим двоичным трехразрядным числом.

Пример. $(2753)_8 \leftrightarrow (010.111.101.011)_2$

Алгоритм I.5. Перевод (запись числа)$_2$ → (запись числа)$_{16}$

Двоичная запись числа разбивается справа - налево на тетрады (каждая по четыре разряда). Если в самой левой тетраде меньше четырех цифр – слева добавляются нули. Четырехразрядное двоичное число каждой триады заменяется восьмеричной цифрой в соответствии с Таблица I.3.

Алгоритм I.6. Перевод (запись числа)$_{16}$ → (запись числа)$_2$

Каждая цифра шестнадцатеричной записи числа заменяется соответствующим двоичным четырехразрядным числом.

Пример.

$(5EB)_{16} \leftrightarrow (0101.1110.1011$

2.6. Контрольные вопросы

1. Как определяется позиционная система счисления (кодирования) целых неотрицательных чисел?
2. Алгоритм перевода числа из системы счисления с основанием **10** в систему счисления с основанием **B**.
3. Алгоритм перевода числа из системы счисления с основанием **B** в систему счисления с основанием **10**.
4. Перевод чисел из одной системы счисления в другую, если основания систем кратны двум.

3. Оперативная память

3.1. Двоичный запоминающий элемент

Хранение (запоминание) данных обеспечивается носителем данных в совокупности с операцией чтения и операцией записи данных, хранящихся на этом носителе.

Носитель данных представляет собой объект (или процесс), способный находиться в одном из n устойчивых состояний равновесия. С каждым состоянием связана одна и только одна единица данных $\mathbf{d_i}$**: i=1…n**. Если в данный момент времени носитель данных находится в состоянии **i**, то подразумевается, что он хранит единицу данных $\mathbf{d_i}$.

Операция чтения подразумевает определение состояния равновесия, в котором находится носитель данных, и выдачу в качестве результата связанной с этим состоянием единицы данных. **Операция записи** производит установку носителя данных в состояние, с которым связана записываемая единица данных.

Двоичный символ - минимальная единица данных (бит, двоичная цифра, логическая константа), которая может храниться устройствами компьютера и передаваться между этими устройствами. Все остальные единицы данных образуются как совокупности двоичных символов. Двоичный символ можно рассматривать как двоичную переменную, которая имеет два допустимых значения: $\mathbf{d \in \{0,1\}}$.

В компьютере двоичный символ – бит, представляется либо электрическим потенциалом (низкий - ноль, высокий - единица), либо электрическим импульсом (отрицательный - ноль, положительный - единица). Для передачи двоичных символов - бит между устройствами служит **линия связи**.

Носитель двоичного символа (триггер) - электронная схема, которая в каждый момент времени может находиться в одном из двух возможных состояний. Будем обозначать носитель двоичного символа - триггер как **b**. С каждым состоянием связано одно из двух допустимых значений **{0,1}**. То состояние, в котором находится триггер в рассматриваемый момент времени **t**, будем называть текущим состоянием и обозначать $\mathbf{b^t}$.

Будем также говорить, что если триггер находится в состоянии $\mathbf{b^t}$, то он хранит (помнит) двоичную цифру $\mathbf{D(b^t) = d^t}$, связанную с этим состоянием.

Двоичным запоминающим элементом (ДЗЭ) мы называем триггер в совокупности с электронными схемами, реализующими операции чтения - записи двоичного символа. В каждый момент времени ДЗЭ находится в одном из двух возможных состояний.

Операция записи бита в ДЗЭ - установка ДЗЭ в состояние, связанное с записываемым двоичным символом:

ДЗЭ := Линия записи **(I.8)**

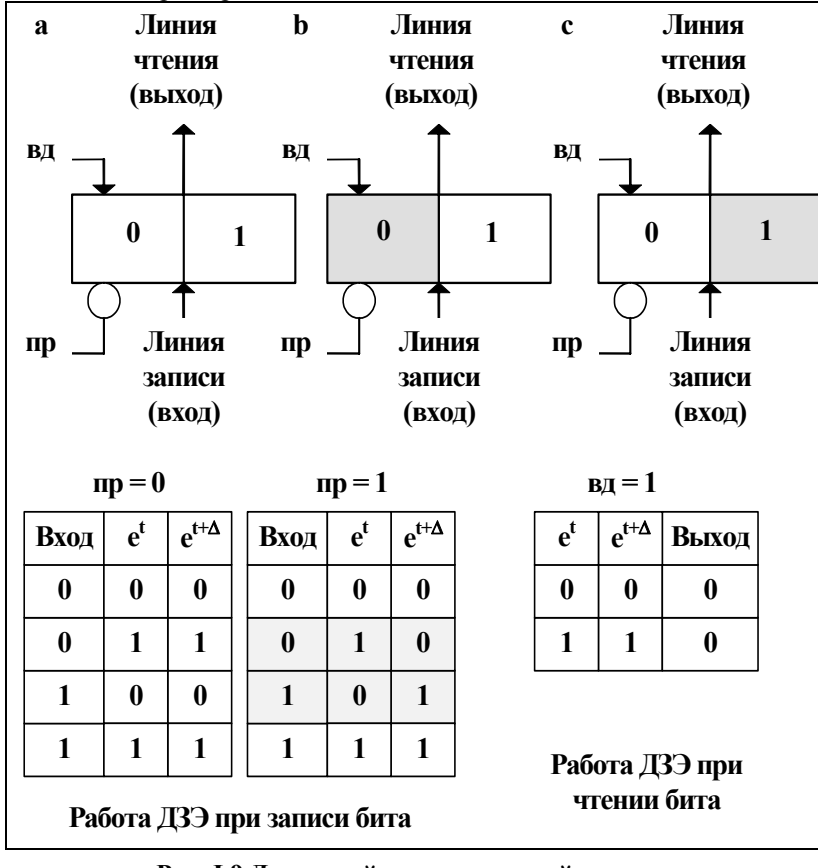
Операция чтения бита из ДЗЭ – определение двоичного символа, связанного с текущим состоянием ДЗЭ:

Линия чтения := ДЗЭ **(I.9)**

Аналогия. Металлическую монету можно рассматривать как двоичный запоминающий элемент, имеющий два состояния - “орел” и “решка”. Текущее состояние - состояние, которое наблюдается при взгляде на монету сверху.

Напишем на стороне “орел” цифру 1, а на стороне “решка” - цифру 0. Тогда, в каждый момент времени носитель двоичного знака хранит ту цифру, которая видна при взгляде сверху. Чтобы записать единицу (ноль) надо положить монету “орлом” (“решкой”) вверх.

На Рис. I.9 приведено условное обозначение ДЗЭ, основу которого составляет триггер.



пр = 0

Вход	e^t	$e^{t+\Delta}$
0	0	0
0	1	1
1	0	0
1	1	1

пр = 1

Вход	e^t	$e^{t+\Delta}$
0	0	0
0	1	0
1	0	1
1	1	1

Работа ДЗЭ при записи бита

вд = 1

e^t	$e^{t+\Delta}$	Выход
0	0	0
1	1	0

Работа ДЗЭ при чтении бита

Рис. I.9 Двоичный запоминающий элемент

Текущее состояние триггера принято обозначать штриховкой. На этом же рисунке изображены линия записи (вход), по которой поступает двоичный символ, и линия чтения (выход), на которую передается с триггера двоичный символ.

Не следует думать, что любой поступающий на линию записи двоичный символ запоминается на ДЗЭ. Для того чтобы его запомнить необходимо подать на управляющий вход (**пр**) единичный сигнал разрешения приема с линии записи. По сути дела, этот сигнал инициирует выполнение операции записи в ДЗЭ. Точно так же хранящейся в ДЗЭ двоичный символ выдается на линию чтения только при поступлении на управляющий вход (**вд)** единичного сигнала разрешения выдачи на линию чтения. По сути дела, этот сигнал инициирует выполнение операции чтения в ДЗЭ.

Более подробно сказанное выше показано в таблицах чтения – записи (Рис. I.9).

Подводя итог, можно сказать:

- двоичный запоминающий элемент (ДЗЭ) хранит (помнит) последний, принятый с линии записи двоичный символ;
- при считывании хранящийся в двоичном запоминающем элементе двоичный символ не изменяется.

Таким образом, записав символ, можно его считывать многократно.

3.2. Двоичный запоминающий регистр

Двоичное **n** - разрядное слово определяется как последовательность двоичных символов длины **n**:

$$\mathbf{w = d_{n-1}, d_{n-2}, \ldots, d_i, \ldots, d_1, d_0.} \qquad \textbf{(I.10)}$$

Каждый элемент $\mathbf{d_i}$ такой последовательности являет собой **i**- й разряд слова. Разряды нумеруются справа - налево (от младшего к старшему). Длина слова определяется как число составляющих его разрядов.

Двоичный носитель слова (ДНС) мы определяем как последовательность, состоящую из **n** двоичных запоминающих элементов (Рис. I.9).

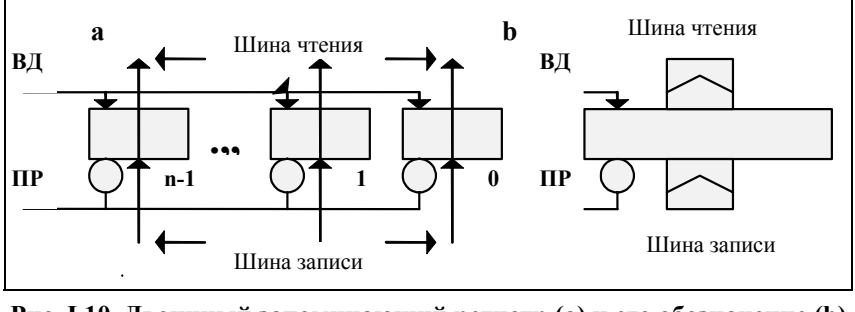


Рис. I.10. Двоичный запоминающий регистр (а) и его обозначение (b)

Элемент $\mathbf{b_i}$ этой последовательности называется **i-** м разрядом носителя слова.

Каждый разряд носителя слова в текущий момент времени находится в состоянии $\mathbf{b^t_i}$ и хранит двоичный символ $\mathbf{D(b^t_i)}$, связанный с этим состоянием. Последовательность: $\mathbf{b^t_{n-1},b^t_{n-2},...,b^t_i,...,b^t_1,b^t_o}$ текущих состояний разрядов носителя слова мы называем **текущим состоянием носителя слова.**

Последовательность двоичных символов:

$$\mathbf{w^t = D(b^t_{n-1}),D(b^t_{n-2}),...,D(b^t_i),...,D(b^t_1),D(b^t_o)\ ;} \qquad \mathbf{(I.11)}$$

соответствующую текущему состоянию носителя слова, мы называем **словом**, хранящимся в носителе слова в текущий момент времени.

Всего имеется $\mathbf{2^n}$ состояний **n** - разрядного носителя слова, с каждым из которых связано одно и только одно **n** - разрядное двоичное слово. Следовательно, **n** - разрядный носитель слова способен хранить $\mathbf{2^n}$ различных **n** - разрядных двоичных слов.

Аналогия. Последовательность **n** металлических монет можно рассматривать как носитель слова. Последовательность “орлов” и “решек”, видимых сверху, - реализуемое состояние носителя. Последовательность нулей и единиц, нарисованных на этих “орлах” и “решках”, - хранящееся в носителе двоичное слово.

Носитель слова в совокупности с реализованными для него операциями чтения - записи мы называем **двоичный запоминающий регистр.**

Носитель слова превращается в двоичный запоминающий регистр **r** следующим образом (Рис. I.10a):

- линии чтения объединяются в шину чтения ширины **n**;
- линии записи объединяются в шину записи ширины **n**;
- управляющие линии (**вд)** объединяются в общую управляющую линию **ВД**;
- управляющие линии (**пр)** объединяются в общую управляющую линию **ПР**.

Операция записи слова в запоминающий регистр **r** - установка носителя слова в состояние, представленное словом на шине записи:

Запись(r, Линия_записи) **(I.12)**

Операция чтения слова из запоминающего регистра **r -** выдача на шину чтения слова, представленного текущим состоянием носителя слова:

Чтение(r, линия_чтения) **(I.13)**

В результате получается двоичный запоминающий регистр (Рис. I.10b), способный принимать и запоминать двоичное слово с шины записи (по сигналу **ПР**), а также выдавать хранящееся слово на шину чтения (по сигналу **ВД**).

Разряды носителя слова также называют разрядами регистра. Последовательность целых неотрицательных чисел: **0,1,...,i,...,n-2,n-1** - называется **разрядной сеткой регистра.**

В дальнейшем, двоичный запоминающий регистр мы будем называть просто регистром, а хранящееся в нем слово - содержимым регистра. Таким образом, регистр обеспечивает одновременную запись (одновременное чтение) всех разрядов слова.

3.3. Двоичная память

Создание двоичного запоминающего регистра решает проблему хранения (запоминания) любого двоичного **n**-разрядного слова. Однако, при вычислениях необходимо одновременно хранить множество слов. Возникает проблема создания памяти (запоминающего устройства) для хранения **Q** слов, каждое из которых состоит из n разрядов (символов).

Двоичный Q словный текст мы определяем как последовательность длины **Q** , состоящую из **n**-разрядных двоичных слов:

$$\mathbf{text = w0,w1,...,wj,...,wQ-2,wQ-1} \qquad \mathbf{(I.14)}$$

Двоичный носитель текста (ДНТ) мы определяем как последовательность, состоящую из **Q** регистров (Рис. I.11а).

Примечание. Носитель текста принято также называть запоминающим массивом.

Каждый регистр носителя текста $\mathbf{r_j}$ в текущий момент времени находится в состоянии $\mathbf{r^t_j}$ и хранит двоичное слово $\mathbf{W(r^t_j)}$, связанное с этим состоянием.

Последовательность: $\mathbf{r^t_0, r^t_1, ..., r^t_j, ..., r^t_{Q-2}, r^t_{Q-1}}$, текущих состояний регистров носителя текста называется текущим состоянием носителя текста.

Последовательность двоичных слов:

$$\mathbf{Text^t = W(r^t_0), W(r^t_1), ,,,,. W(r^t_j), ..., W(r^t_{Q-2}), W(r^t_{Q-1}),} \qquad \mathbf{(I.15)}$$

соответствующих текущему состоянию носителя текста, мы называем текстом, хранящимся в носителе текста в текущий момент времени.

Носитель текста в совокупности с реализованными для него операциями чтения - записи мы называем **двоичным запоминающим устройством** или двоичной памятью (в дальнейшем, просто памятью). О составляющих носитель текста регистрах мы говорим как о регистрах памяти. Текст, хранящейся в носителе текста, принято называть содержимым памяти.

По аналогии с предыдущим, следовало бы определить для носителя текста операции чтения-записи, обновляющие или читающие текст целиком. Однако технически сложно и дорого реализовать операцию записи, которая одновременно изменяет содержимое всей памяти, т.е. всю хранящуюся в ней последовательность слов. Точно так же можно сказать об операции чтения всего содержимого памяти.

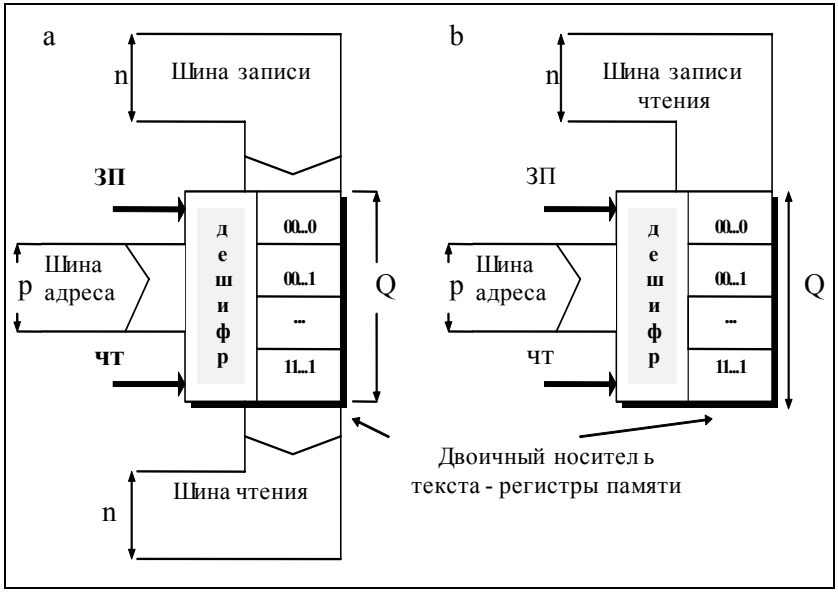


Рис. I.11. Оперативная память с двумя шинами данных (a) и с одной шиной данных (b)

К счастью, в этом нет необходимости. Практика показывает, что любой вычислительный процесс можно представить как последовательность чтения и записи содержимого отдельно взятых регистров памяти.

Поэтому определяются операции чтения - записи в регистр памяти с указанным адресом. Причем, в качестве **адреса регистра** используют его порядковый номер в носителе текста. Последовательность целых чисел: **0,1,...,j,...,Q-2,Q-1** - называется **адресным пространством памяти**.

Операция записи слова в регистр памяти с заданным адресом:

Запись(адрес, новое слово) **(I.16)**

Операция чтения из регистра памяти с заданным адресом:

Чтение(адрес) **(I.17)**

Используя операции чтения - записи можно определить операцию обмена словами между регистрами памяти:

Запись(адрес2,Чтение(адрес1)) **(I.18)**

Более привычно представление операции обмена данными в виде оператора присваивания:

адрес2 := адрес1 **(I.19)**

Регистр памяти, для которого реализуется запись - чтение называется **доступным регистром**, а его адрес - **адресом доступа**.

Носитель текста превращается в память следующим образом (Рис. I.11а):

- шины чтения всех регистров памяти объединяются в единую шину чтения ширины **n**;
- шины записи всех регистров памяти объединяются в единую шину записи ширины **n**;
- вводится шина адреса (ширины **p**), по которой подается в память двоичный код адреса доступа (в дальнейшем, адрес доступа);
- конструируется электронное устройство - дешифратор, реализующий доступ к регистру памяти с заданным адресом (обеспечивающий возможность чтения или записи для заданного регистра памяти);
- предусматривается управляющая линия разрешения записи (**ЗП**) и управляющая линия разрешения чтения (**ЧТ**).

Для преобразования двоичного кода адреса в управляющий сигнал доступа используется **дешифратор**, который представляет собой электронную схему, имеющую **p** двоичных линий входа и 2^p двоичных линий выхода. Когда на вход с шины адреса поступает одна из 2^p возможных двоичных последовательностей кода адреса **adr**, на одном и только одном двоичном выходе появляется единица.

Таким образом, дешифратор однозначно преобразует двоичную входную последовательность кода адреса **adr** в соответствующий ей выход. Линия этого выхода и делает доступным регистр с адресом **adr**.

Алгоритм I.7. Исполнение операции записи слова в регистр памяти с заданным адресом.

1. На шину адреса подается адрес доступа.
2. Дешифратор обеспечивает доступ к регистру памяти с заданным адресом.
3. На шину записи подается слово, которое необходимо записать в доступный регистр.
4. На управляющую линию **ЗП** подается единичный сигнал разрешения записи.
5. Двоичное слово с линии записи передается в доступный регистр памяти и становится его содержимым.

Алгоритм I.8. Исполнение операции чтения слова из регистра памяти с заданным адресом.

1. На шину адреса подается адрес доступа.
2. Дешифратор обеспечивает доступ к регистру памяти с заданным адресом.
3. На управляющую линию **ЧТ** подается единичный сигнал разрешения чтения.
4. Двоичное слово из доступного регистра передается на шину чтения.

Кратко затронем вопрос о способе доступа к регистрам памяти. Говорят, что память устроена как **память прямого доступа** в случае, когда в операциях явно указывается адрес регистра памяти. Память прямого доступа называется равнодоступной (памятью со случайным доступом) в случае, когда время доступа к регистру памяти не зависит от того, какой регистр был доступен перед этим. **Равнодоступная память** (память со случайным доступом), подключенная непосредственно к устройству выполнения операций обработки данных (центральному процессору), называется оперативной памятью (оперативным запоминающим устройством - ОЗУ). Ее англоязычное название - Random Access Memory (RAM).

По принципу действия и технической реализации RAM подразделяется на статическую SRAM, основой которой является триггер, и на динамическую память DRAM, основой которой служит конденсатор. Существенная особенность оперативной памяти - исчезновение хранящихся данных при выключении питания.

В настоящее время более употребительна оперативная память с одной шиной передачи данных (Рис. I.11.b).

Из сказанного выше ясно, что, принято различать логические единицы данных (знак, слово, текст) и технические устройства для хранения этих единиц (двоичный запоминающий элемент, регистр, память).

С целью измерения и сравнения размеров единиц данных вводятся следующие единицы.

Минимальной единицей измерения логической единицы данных служит **бит** - двоичный символ. Более крупная единица измерения – **байт,** которая определяется как единица данных в восемь двоичных символов - разрядов. **Слово –** единица данных , состоящая из целого числа байт.

Ниже приводятся более крупные единицы измерения размеров данных.

Килобайт - 210 байт = 1024 байт.

Мегабайт - 220 байт.

Гигабайт - 230 байт.

Терабайт -240 байт.

Вышеназванные единицы служат не только для измерения размеров данных, но также для измерения объемов запоминающих устройств, которые хранят эти данные. Термин «байт» используется в двух смыслах. Во - первых, это единица измерения размера данных– байт данных. Во - вторых, это запоминающий регистр емкостью в один байт - байт памяти. Какой смысл в каждом конкретном случае придается тому или иному термину, ясно из контекста.

Во всех современных компьютерных системах регистр оперативной памяти имеет размер в один байт. Говорят, что оперативная память построена «по байтовому» принципу. Таким образом, минимальная единица оперативной памяти, которую можно адресовать, - один байт.

Оперативная память современных персональных компьютеров имеет емкость от 16-ти мегабайт и выше. Чтобы реально представить этот объем, скажем, что в одном мегабайте памяти можно сохранить 500 страниц печатного текста. (Цифра эта зависит от способа хранения и является весьма ориентировочной.)

Кроме емкости, другой характеристикой памяти является быстродействие. Быстродействие памяти определяется как время, необходимое для выполнения одной операции чтения (записи). Очевидно, чем больше быстродействие оперативной памяти – тем больше скорость обработки данных центральным процессором.

При конструировании компьютера определяется базисная техническая единица памяти - слово. Для современного персонального компьютера определяется слово размером в четыре байта (32 двоичных разряда). Для старых моделей персональных компьютеров определялось слово размеров в два байта (16 двоичных разрядов) и даже один байт (8 двоичных разрядов). Суперкомпьютеры имеют слово размером в восемь байт (64 двоичных разряда). На базе слова определяются производные логические единицы данных: полуслово, двойное слово и т.д.

Особо остановимся на зависимости адресного пространства памяти от длины адреса доступа. Для заданной длины **p** адреса доступа легко определить число различных адресов, которое равно $\mathbf{2^p}$. Таким образом, очевидно соотношение для определения размера адресного пространства: $\mathbf{Q = 2^p}$. Ни меньшее, ни большее адресное пространство при заданном **p** создавать не имеет смысла. Таким образом, для шестнадцатиразрядной архитектуры размер адресного пространства 64 килобайта. Для тридцатидвухразрядной архитектуры размер адресного пространства: 232=4 Гб.

Примечание. Ширина шины данных (шины адреса) не всегда совпадает с длиной слова (длиной кода адреса). Из экономических соображений слово длины **n** (код адреса длины **p**) передается по дешевой шине меньшей ширины в несколько тактов.

Например, тридцатидвухразрядное слово может передаваться по шестнадцатиразрядной шине данных за два такта. Естественно, экономя в стоимости - проигрывают в быстродействии.

3.4. Почему компьютерная память двоичная

С теоретической точки зрения, запоминающий регистр может строиться из **n** носителей символа, каждый из которых имеет **B** состояний равновесия и способен хранить любую из **B** цифр. В этом случае запоминающий регистр в целом может хранить **n**-разрядное **B**-ичное число. Однако память современных компьютеров строится на основе носителей двоичного символа. И для этого есть свои причины.

Прежде всего, очевидно, что чем меньше число состояний у носителя символа, - тем надежнее он работает. Минимальное число состояний у носителя символа равно двум.

К счастью, критерий надежности не приходит в противоречие с критерием минимальной стоимости изготовления регистра. Существует разумное предположение, что стоимость изготовления регистра пропорциональна числу знаков регистра, которое определяется как произведение: **z = B×n**, где **B**- число состояний носителя знака, **n**-число разрядов регистра.

Пусть z = 20. Рассмотрим два крайних варианта: двухразрядный регистр из носителей знака с десятью состояниями равновесия и десятиразрядный регистр из носителей знака с двумя состояниями равновесия. В первом случае, регистр может хранить N =102 = 100 различных слов. Во втором случае: N = 210 = 1024 различных слова.

Закономерна постановка вопроса об оптимальном, с точки зрения эффективности, числе состояний носителя знака. Несложные расчеты показывают, что график зависимости **N** (число различных слов) от **B** (число состояний носителя знака) при постоянном **z** (стоимость изготовления регистра) имеет следующий вид:

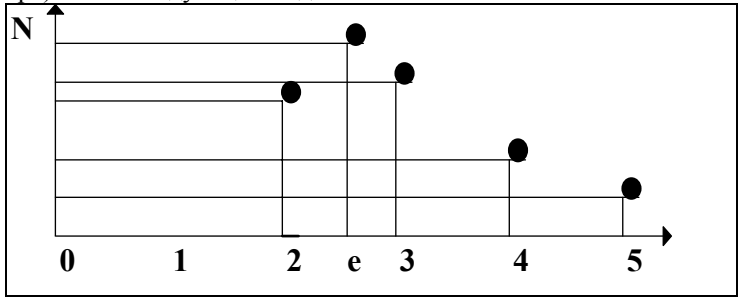


Рис. I.12. График зависимости N от B

Из графика видно, что оптимальным, с точки зрения стоимости изготовления, является нецелое основание e = 2.71... Среди целых оснований оптимум достигается при B=3. Но разница между N(3) и N(2) незначительна. Поэтому, с учетом критерия надежности работы, в качестве регистра компьютера используется двоичный запоминающий регистр, что накладывает двоичную специфику на всю архитектуру компьютера.

Подводя итог, можно сказать следующее:

1. Основой оперативной памяти служит носитель текста (запоминающий массив) из $\mathbf{2^p}$ запоминающих регистров - байтов памяти;
2. В каждый момент времени каждый байт памяти хранит байт данных (8 - разрядное двоичное слово);
3. В каждый момент времени память в целом хранит двоичный текст размером $\mathbf{2^p}$ байт;
4. Оперативная память принимает **p** - разрядный двоичный код адреса с шины адреса и обеспечивает прямой доступ к каждому из $\mathbf{2^p}$ регистров массива запоминающих регистров по заданному адресу;

5. Оперативная память является равнодоступной (время прямого доступа к байту памяти не зависит от предыстории);
6. Оперативная память обеспечивает выполнение операций чтения/записи для доступного регистра.

3.5. Контрольные вопросы

1. Понятие двоичного символа, двоичного слова и двоичного текста.
2. Технические средства запоминания одного двоичного символа.
3. Технические средства запоминания одного двоичного слова.
4. Технические средства запоминания множества двоичных слов.
5. Понятие разрядной сетки и адресного пространства.
6. Понятие прямого доступа и равнодоступности.
7. Оптимальность двоичной памяти.
8. Свойства оперативной памяти.

4. Типы данных

4.1. Понятие типа данных

Мы построили равнодоступную память (память со случайным доступом), состоящую из **Q = 2p** адресуемых регистров - байт памяти. В каждом байте памяти может храниться 8 - разрядное двоичное слово (содержимое байта памяти - байт данных). Однако при вычислениях компьютер имеет дело не с безликими словами, а с единицами данных. Таким образом, необходимо решить, каким образом различные единицы данных хранятся в компьютерной памяти.

Минимальной адресуемой единицей памяти является байт. Однако, в общем случае, восьми двоичных разрядов недостаточно для размещения любой единицы данных. Поэтому мы будем говорить, что единица данных хранится в **ячейке памяти,** которая состоит из нескольких байт оперативной памяти. Адресом ячейки служит адрес первого байта, ее образующего.

Здесь мы рассматриваем только простые единицы данных, каждая из которых хранится и обрабатывается как единое целое. К числу простых единиц данных относятся числа и символы. В некоторых случаях в качестве простой единицы данных выступает строка символов.

Понятие типа данных является важнейшим понятием программирования. **Тип данных** - совокупность правил, определяющих некоторую категорию данных как:

- множество допустимых для нее значений;
- множество допустимых операций над допустимыми значениями;
- способ хранения значения (формат ячейки хранения).

Каждая единица данных в программе является либо константой, либо переменной. Отличие заключается в том, что значение переменной может меняться в процессе выполнения программы.

Следует различать типы данных, реализованные схемотехнически в компьютере, и типы данных, реализованные программно в поддержку языка программирования. Компьютерные типы данных являются простыми (скалярными). Примеры: целое число, вещественное число, символ.

При разработке языков программирования повторяются простые типы данных и затем на их основе конструируются сложные (структурные) типы данных. Примеры: массив, запись, файл.

Типизация данных необходима с нескольких точек зрения. При вводе единицы данных имеется возможность автоматически определить формат ячейки, необходимой для ее хранения.

Указание типа позволяет проконтролировать корректность применения операции к операндам.

Например, сложение вещественного числа и символа не имеет смысла. Появляется также возможность использовать один символ для обозначения различных операций (перегрузка операций), например, операция сложения целых чисел и операция сложения вещественных чисел выполняются по различным алгоритмам, но обозначаются одним знаком «+». Исходя из описания типов операндов, можно автоматически распознать, какой алгоритм исполнения операции в данном конкретном случае применять. Более того, во многих случаях возможно автоматическое преобразование значений разнотипных операндов к одному формату.

4.2. Тип данных «целое без знака»

Рассмотрим ячейку памяти размером в один байт. Она состоит из 8-ми двоичных разрядов и может хранить 28 = 256 различных двоичных слова. Каждое такое слово можно интерпретировать как код целого неотрицательного числа (целое без знака).

Таким образом, с каждым состоянием можно связать вполне определенное целое без знака. И если байт находится в каком-либо состоянии $\mathbf{r^t}$, мы утверждаем, что он хранит число **цбз($\mathbf{r^t}$)**, связанное с этим состоянием (Рис. I.13)

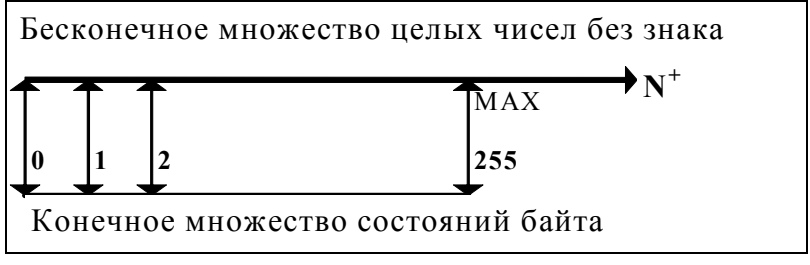


Рис. I.13. Хранение целого без знака

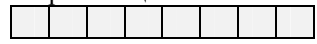
Однако конечность числа состояний байта памяти позволяет связать с его состояниями только первые 256 чисел. Отсюда возникает принципиальное для компьютерных вычислений понятие **максимального представимого числа** (в ячейке памяти). Как следствие - понятие диапазона представимости чисел в ячейке памяти **Ω = (1…MAX)**. Образно говоря, число может "убираться или не убираться" в ячейку памяти.

Если для хранения целого числа без знака использовать ячейку памяти размером в байт, получаем: $MAX_{\text{короткое цбз}} = 255$, $\Omega_{\text{короткое цбз}}$= (0...255).

Таким образом, мы определили формат хранения единицы данных: короткое целое без знака.

Если для хранения целого без знака использовать ячейку памяти размером в два байта, получаем тип данных целое без знака: $MAX_{\text{цбз}} = 2^{16} = 65535$, $\Omega_{\text{цбз}}$= (0...65535).

Короткое целое без знака



Целое без знака

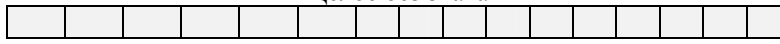


Рис. I.14. Формат хранения целого числа без знака

Существует еще одно ограничение на использование целых без знака, связанное также с конечностью числа разрядов ячейки памяти. Дело в том, что каждое слагаемое может "убираться" в ячейку памяти, а их сумма уже не "убирается" в ячейку памяти. В этом случае, компьютер выдает сигнал "**переполнение разрядной сетки**" и прекращает дальнейшие вычисления.

Как говорилось выше, типы данных реализуются как аппаратным, так и программным путем. Поэтому в различных марках компьютеров по существу одинаковые типы данных могут иметь различные названия. В качестве примера перечислим типы данных, представляющие целые числа без знака, для языка программирования Паскаль (версия 6.0).

Таблица I.4. Целые без знака в языке Паскаль

Тип	Наименование	Диапазон	Ячейка	Операции
byte	байт	0…255	1 байт	<отн>,+,*
word	слово	0…65535	2 байта	<отн>,+,*

Допустимые для этого типа операции: отношение (=,>,< и т.д.), сложение и умножение.

4.3. Типы данных «символ» и «строка символов»

Обозначение (код) символа на модельном уровне являет собой комбинацию графических линий. Число символов, используемых в модельном мире, конечно и они должны быть реализованы на клавиатуре компьютера в виде клавиш. Кроме того, некоторые, используемые лишь компьютером символы, на клавиатуре компьютера не представлены. Множество представимых символов образует алфавит символов компьютера: $\mathbf{S=(s_1,...,s_k,...,s_k)}$.

Поставив в соответствие каждому символу алфавита уникальное целое без знака, получим множество чисел $\mathbf{N_k=(d1,...,dk,...,d_k)}$, каждое из которых обозначает вполне определенный символ. Запись такого числа может храниться в двоичной памяти как код символа.

Теперь необходимо принять решение о размере ячейки памяти, которая используется для хранения символа. На заре эры компьютеризации было принято волевое решение - число различных символов должно быть не более двухсот пятидесяти шести. Для представления любого из $2^8 = 256$ чисел необходима ячейка размером в 8 двоичных разрядов.

Таким образом, появилась единица измерения данных и памяти – **байт**, т.е. слово длиной в восемь двоичных разрядов.

Достаточно быстро выяснилось, что число различных символов, используемых для общения с компьютером, значительно больше двухсот пятидесяти шести. С целью увеличения числа комбинаций двоичных кодов, представляющих символы, были введены два дополнительных бита. Для реализации этих бит на клавиатуре компьютера были введены специальные клавиши. Сначала **Ctrl**, затем **Alt** - число комбинаций двоичных кодов символов возросло до 2^{10}. Но и этого оказалось недостаточно. В настоящее время для хранения символов может использоваться ячейка размером в несколько байт.

Соответствие символов и кодирующих их целых чисел без знака представляется в таблице кодирования. При создании такой таблицы учитываются соображения эффективности передачи кода по каналам связи и помехозащищенности его при такой передаче. С целью удовлетворения различных требований существуют различные таблицы кодирования. Такие таблицы разрабатывались, прежде всего, в расчете на латинский алфавит. Поэтому в любой из них соблюдается упорядоченность кодирования цифр и букв. Так, буква “A” кодируется меньшим числом, чем буква “B” и т.д. Это позволяет использовать арифметические операции для сравнения букв (цифр). Например, для определения того, является ли значение переменной Symbol буквой, достаточно вычислить значение отношения: “A”<= Symbol<=“Z’. При включении русских букв (кириллицы) в таблицы кодирования принцип упорядоченности кодирования соблюдается не всегда. Следует учитывать это при программировании задач обработки символов.

Назовем наиболее распространенные таблицы кодирования символов: ASCII /альтернативная/, CP 866 (упорядоченность русских букв, используется в DOS) - Таблица I.5.

Таблица I.5. Кодирование ASCII

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

Таблица кодирования CP 125 также соблюдает упорядоченность кодирования русских букв, используется в операционной среде Windows. Таблица кодирования КОИ-8 не соблюдает упорядоченность кодирования русских букв, используется в операционной среде UNICS. Таблица кодирования ISO соблюдает упорядоченность кодирования русских букв и является Госстандартом для оформления документации в Российские фонды алгоритмов и программ.

Проблема кодирования (и хранения) символьных строк решается очевидным образом: последовательность образующих строку символов кодируется последовательностью кодов этих символов. Таким образом, если каждый символ занимает при хранении одну ячейку памяти, то строка длиной в **q** символов занимает **q** последовательных ячеек памяти.

К	О	Т
01001011	01001111	01011110
1-й байт	2-й байт	3-й байт

В качестве другого примера назовем типы данных, представляющие символы и строки символов, для языка программирования Паскаль.

Тип	Наименование	Диапазон	Ячейка	Операрции
char	символ	ASCII	1 байт	Отношения
string	строка	дл. стр.<256	дл. стр.	Специфич.

Операции для **char**: отношения, которые позволяют сравнивать и сортировать символы. Операции для **string**: отношения, определение длины строки, конкатенация строк, выделение подстроки, удаление подстроки, вставка подстроки, определение позиции подстроки.

4.4. Тип данных «целое»

При сложении двух целых без знака в качестве результата получается также целое без знака. Операция вычитания определяется как обратная операция к операции сложения. Однако при вычитании результат далеко не всегда является целым без знака. Говоря другими словами, множество $\mathbf{N}^+$ целых чисел без знака незамкнуто относительно операции вычитания (Рис. I.15).

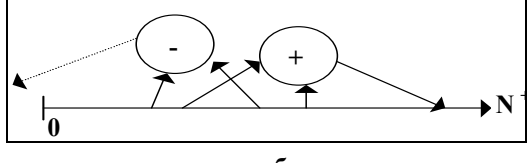


Рис. I.15. Множество целых чисел без знака замкнуто относительно операции сложения и умножения, но незамкнуто относительно операции вычитания

Необходимо ввести понятие отрицательного числа и множества целых чисел со знаком или, просто, **целых чисел Z**. Операция вычитания замкнута относительно этого множества (Рис. I.16).

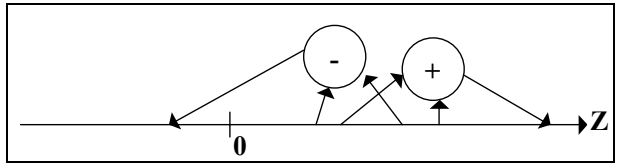


Рис. I.16. Множество целых чисел замкнуто относительно операций сложения, умножения и вычитания.

Очевидным образом вводится понятие алгебры целых чисел:

$$\mathbf{A_{<цч>} = <Z;+,-,*,\oplus,\varnothing,div,mod>} \quad \mathbf{(I.20)}$$

Здесь, **Z** - основное множество алгебры; +,-,* - бинарные операции сложения, вычитания и умножения целых чисел.

Существуют также две унарные операции изменения знака числа: $\oplus$ - унарный плюс и $\varnothing$ - унарный минус. (В практике вычислений обозначения этих унарных операций такое же, как обозначение бинарных операций сложения и вычитания: + и -. Унарность или бинарность операции определяется из контекста записи арифметического выражения.)

Алгебра целых чисел замкнута относительно операций сложения, вычитания и умножения, а также операций изменения знака. Сложнее с операцией деления целых чисел. В общем случае при делении одного целого числа на другое целое число получается нецелое число. Поэтому, вводятся две специфические операции целочисленного деления: **x div y** (результат операции - целая часть от деления) и **x mod y** (результат операции - остаток от деления). Можно утверждать, что алгебра целых чисел замкнута относительно таких операций деления.

Позиционная система счисления (кодирования) целых чисел строится по аналогии с позиционной системой счисления целых чисел без знака. Каноническое арифметическое выражение имеет вид:

$$\mathbf{ЗНАЧЕНИЕ_в(целое) = \pm(d_{n-1}*B^{n-1}+d_{n-2}*B^{n-1}+...+d_1*B^1+d_0*B^0)_в.(\ I.21)}$$

Запись целого числа образуется очевидным образом:

$$\mathbf{\pm< d_{n-1}, d_{n-2},\ldots d_1,d_0 >} \quad \mathbf{(I.22)}$$

Для хранения двоичного кода целого числа используется ячейка памяти, в которой старший разряд представляет знак числа: **0** - знак «плюс». **1** - знак «минус».

Ячейка памяти (размером один байт, два байта, четыре байта) имеет конечное число состояний (Рис. I.17). Отсюда следует, что имеется максимально представимое целое число (положительное) **MAX** и минимально представимое целое число (отрицательное) **MIN**.

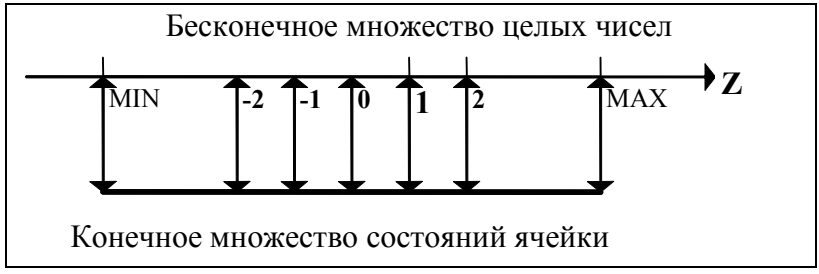


Рис. I.17. Хранение целого числа

Используются три формата хранения целого числа (Рис. I.18):

Рис. I.18. Форматы хранения целого числа (черное поле - знак числа)

Короткое целое, ячейка хранения размером байт: диапазон представимости чисел Ω = -127÷+127, (один разряд для представления знака, 2^7=128 состояний для представления значения).

Целое, ячейка хранения размером два байта: диапазон представимости чисел Ω = - 32768÷+32767(один разряд для представления знака, 2^{15c}остояний для представления значения).

Длинное целое, ячейка хранения - четыре байта диапазон представимости чисел Ω = -2*109÷+2*109 (один разряд для представления знака, 2^{31} состояний для представления значения).

При использовании целых чисел без знака, переполнение разрядной сетки возможно при сложении, вычитании и умножении.

В качестве примера называем типы данных, представляющие целые числа, для языка программирования Паскаль.

Таблица I.6. Допустимые целые типы в языке Паскаль

Тип	Наименование	Диапазон	Ячейка	Операции
shortint	короткое целое	-128...127	1 байт	<отн>,+,-,*
integer	целое	-32768...32767	2 байт	<отн>,+,-,*
longint	длинное целое	-2*109... 2*109	4 байт	<отн>,+,-,*

Допустимые операции: отношения, сложение, вычитание и умножение.

4.5. Тип данных «вещественное»

Вещественное (действительное) число образуется, когда мы производим измерения какой либо числовой характеристики **d** объекта, используя для этого эталонную числовую величину $\mathbf{d_0}$.

В общем случае эталонная величина укладывается в измеряемой величине нецелое число раз. В результате такого измерения получается **вещественное число** как результат операции деления $\mathbf{d \backslash d_0}$.

Очевидным образом вводится понятие алгебры вещественных чисел:

$$\mathbf{A_{<вч>} = <R;+,-,*,/ >.} \qquad \textbf{(I.23)}$$

Здесь, **R**- основное множество алгебры. Очевидно, что эта алгебра замкнута относительно операций сложения, вычитания умножения и деления вещественный чисел.

Позиционная система счисления (кодирования) вещественных чисел строится по аналогии с позиционной системой кодирования целых чисел. Каноническое арифметическое выражение имеет вид:

$$\mathbf{ЗНАЧЕНИЕ_в(вещ) = \pm(d_{p-1}*B^{p-1}+d_{p-2}*B_{p-2}+...+d_1*B^1+d_0*B^0+ d_{-1}*B^{-1}+d_{-2}*B^{-2}+...)_в.}$$

Последовательность коэффициентов этого канонического выражения также однозначно обозначает вещественное число и называется записью вещественного числа.

$$\mathbf{ЗАПИСЬ_B(вещ) = (d_{p-1}d^{p-2}...d_1d^0.d_{-1}d^{-2}...)_B.} \qquad \textbf{(I.24)}$$

Последовательность цифр до точки называется целой частью числа, последовательность за точкой называется дробной частью числа.

Вещественное число может быть представлено в системе счисления с произвольным основанием **B**. На практике используются: десятичная, двоичная, а также вспомогательные восьмеричная и шестнадцатеричная системы счисления.

Если дробная часть числа являет собой бесконечную последовательность цифр, мы имеем дело с иррациональным числом. Пример иррационального числа - число π.

Вещественное число называется рациональным, если его дробная часть содержит конечное число символов.

$$\mathbf{ЗАПИСЬ_B(рац.\ число) = d_{p-1}d^{p-2}...d_1.d_0.d_{-1}d_{-2}...d_{-q}.} \qquad \textbf{(I.25)}$$

Вспомним, что множество целых чисел **Z** дискретно (образует счетное множество). Тогда как множество вещественных чисел **R** не дискретно (образует всюду плотное множество). И, в отличие от целых чисел, в любом, сколь угодно малом интервале оси **R** содержится бесконечное множество вещественных чисел.

Ячейка памяти, какой бы размер ее не выбрали, имеет конечное число разрядов и конечное число состояний. Но это означает, что не существует даже малого отрезка оси вещественных чисел **R**, которому можно поставить в соответствие конечное множество состояний ячейки памяти.

Единственный выход из такой тупиковой ситуации - представлять состояниями ячейки памяти не числа из **R**, а отрезки числовой оси **R** (Рис. I.19).

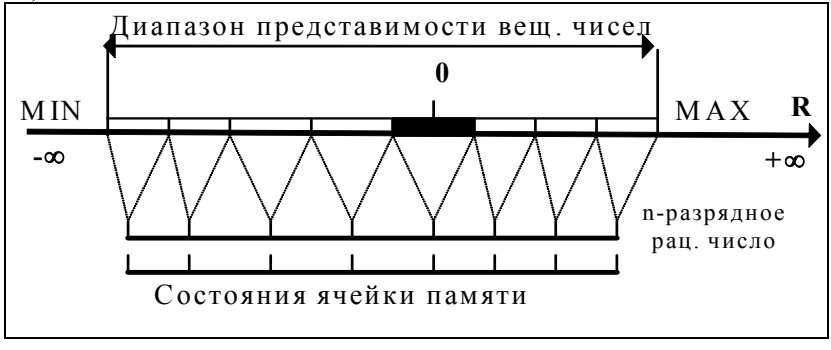


Рис. I.19. Представление "вещественных" чисел

Более подробно, каждый такой отрезок представляется «средним» **n** - разрядным рациональным числом. Множество таких чисел конечно и появляется возможность представить каждое из них состоянием **n** - разрядной ячейки памяти. Очевидно, что существует максимально представимое число и минимально представимое число в ячейке памяти с заданным числом разрядов. Аналогично предыдущему, определяется диапазон представимости вещественных чисел.

Существует два различных формата представления вещественного числа в **n** - разрядной ячейке памяти.

В формате с **фиксированной точкой** на представление целой и дробной части числа отводится соответственно **p** и **q** разрядов (Рис. I.20).

Формат с **плавающей точкой** (полулогарифмический формат) в той же самой разрядной сетке позволяет реализовать значительно больший диапазон представимости вещественных чисел.

Рис. I.20. Представление вещественного числа в формате с фиксированной точкой

Этот формат основан на следующем свойстве человеческого сознания: в записях очень больших числах (масса солнца $2\cdot10^{30}$г.) и в записях очень маленьких числах (масса электрона $9\cdot10^{-28}$г.) человеческий мозг запоминает лишь небольшое количество значащих (отличных от нуля) цифр. Величина же числа определяется количеством нулей. Это выражается в так называемой полулогарифмической записи десятичного числа:

$$\text{<мантисса>}10^{\text{<порядок>}} \qquad \textbf{(I.26)}$$

Здесь, мантисса - десятичное рациональное число с фиксированной запятой (значащие цифры), порядок- число нулей (величина числа).

Строго говоря, представление числа в таком формате неоднозначно. Например: $200\cdot10^{1} = 20000\cdot10^{-2} = 2\cdot10^{2} = 0.2\cdot10^{3}$. Однако, если использовать нормализованную мантиссу, то это представление становится однозначным. Нормализованная мантисса сразу после точки содержит отличную от нуля цифру.

Компьютерный формат вещественного числа с плавающей точкой строится по тому же принципу:

$$\text{<нормализованная мантисса>}16^{\text{<порядок>}}.$$

Здесь, мантисса - двоичная запись вещественного числа в формате с плавающей точкой, порядок - двоичная запись количества нулей (Рис. I.21).

В конкретном языке программирования обычно определяется несколько вещественных типов. Обычно, для хранения числа с плавающей точкой отводится ячейка размером в четыре байта: 8 бит для представления порядка и знака; 24 бита - для мантиссы.

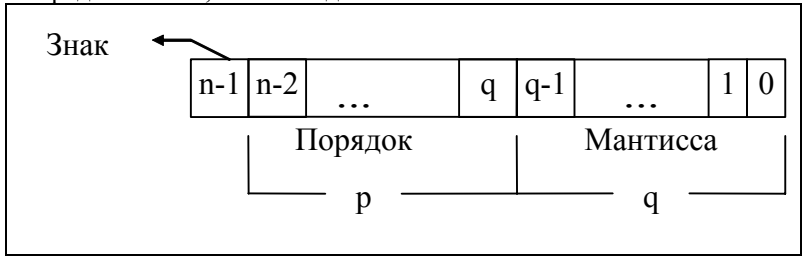


Рис. I.21. Представление вещественного числа в формате с плавающей точкой

Кроме того, допускается тип «двойная точность», значение которого размещается в ячейке размером восемь байт. В некоторых системах все 32 добавочных бита используются для хранения мантиссы. Это увеличивает число значащих цифр в записи числа, но не расширяет диапазон представимости чисел. В других системах увеличивается число битов, отводимых для хранения порядка, что расширяет диапазон представимости чисел.

В качестве примера называем типы данных, представляющие вещественные числа, для языка программирования Паскаль (версия 6.0).

Таблица I.7. Вещественные типы в языке Паскаль

Тип	Наименование	Диапазон	Число знач. цифр	Ячейка
real	вещественный	$10^{-38}...10^{38}$	12	6 байт
single	одинарная точн.	$10^{-38}...10^{38}$	8	4 байта
double	двойная точн.	$10^{-308}...10^{308}$	16	8 байт
extended	повышен. точн.	$10^{-4931}...10^{4931}$	18	10 байт
comp	длинное целое	$9*10^{-18}...9*10^{18}$	19	8 байт

Допустимые операции над значениями вещественного типа: операции отношения, сложение, вычитание, умножение, деление.

4.6. Специфика обработки чисел на компьютере

Основное отличие обработки чисел на компьютере по сравнению с «ручными» вычислениями - представление значений в ячейке с конечным числом разрядов.

Относительно максимального представимого числа, минимального представимого числа и диапазона представимости целых чисел в **n** - разрядной ячейке памяти мы говорили выше. Эта же специфика сохраняется и при использовании вещественных чисел.

Следствием способа представления всюду плотного множества вещественных чисел в ячейке с ограниченным числом разрядов является принципиальная приближенность вычислений. При этом возникают следующие побочные эффекты.

Содержимое ячейки памяти представляет не одно значение, а диапазон значений вещественных чисел.

Возникает проблема, когда не являющееся нулем число находится в окрестность нуля. В этом случае компьютер воспринимает его как "машинный нуль" со всеми вытекающими отсюда последствиями.

В некоторых случаях происходит потеря точности вычислений. Приведем пример для четырехразрядной ячейки, хранящей десятичное число: (10.00/03.00)*03.00 = 09.99.

В некоторых случаях результат вычислений зависит от порядка вычислений.

Пусть x=10, y=12,z=5. В абстрактной алгебре: (x*y)/z = x*(y\z)=24.

Используем по-прежнему четырехразрядную десятичную ячейку. При одном порядке вычислений x*y = 10.00*12/00 = 00.00 - переполнение разрядной сетки, дальнейшие вычисления невозможны.

При другом порядке вычислений получаем правильный результат. y\z = 12.00\05.00 = 02.40; x*(y\z) = 10.00*02.40 = 24.00

4.7. Интерпретация двоичного кода единицы данных

Данные различного типа в памяти вычислительной машины представляются двоичными кодами. Причем, если для единицы данных определенного типа формат хранения ее значения определяется однозначно, то этого нельзя сказать об интерпретации хранящегося в памяти двоичного кода (Рис. I.22).

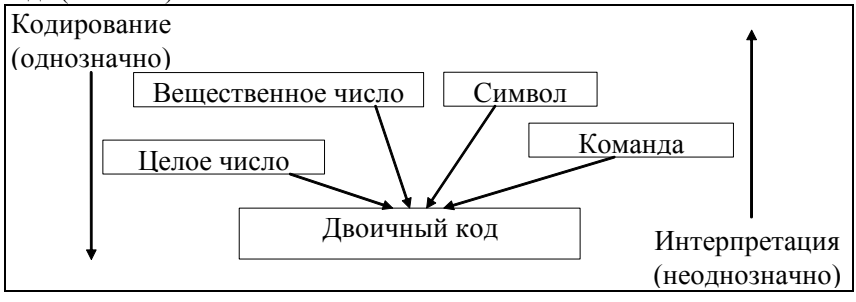


Рис. I.22. Кодирование и интерпретация единицы данных

Таким образом, важной проблемой является интерпретация (истолкование) двоичного кода, хранящегося в регистрах памяти. Наиболее очевидный способ - использовать для указания типа специальный байт (тэг), который является первым в ячейке памяти. Это позволяет при обращении по адресу к ячейке памяти проанализировать первый байт и определить тип хранящейся в этой ячейке единицы данных (Рис. I.23a).

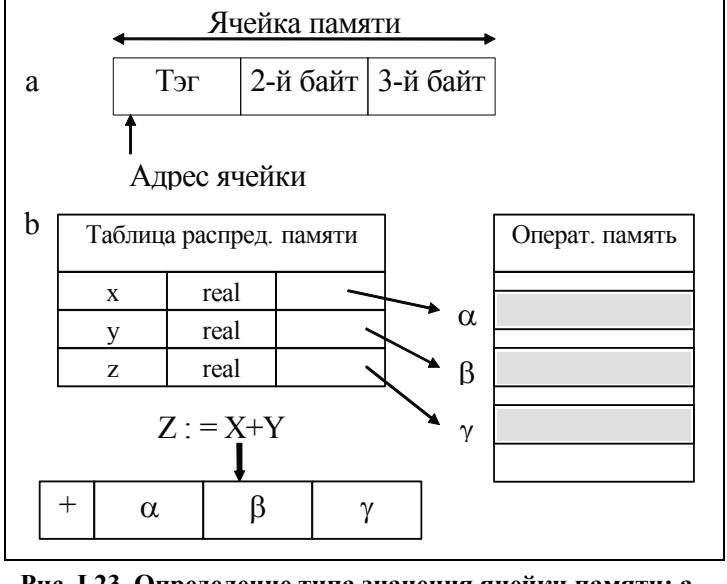


Рис. I.23. Определение типа значения ячейки памяти: a - специальным байтом, b - по месторасположению в памяти

В качестве платы за простоту имеем расточительное расходование оперативной памяти (дополнительный байт на каждую ячейку).

На первом этапе становления компьютеризации оперативная память была дорогая, поэтому конструкторы компьютеров и программного обеспечения пошли другим путем. Тип содержимого ячейки определяется по ее месторасположению в памяти. Так, если операция целочисленного сложения использует ячейки с адресами α, β и γ, то в этих ячейках должны храниться данные целого типа.

Таким образом, возникает задача распределения оперативной памяти, т.е. «нарезка» ее на области, в которых хранятся данные одинаковых типов.

При программировании на уровне машинных кодов или языка ассемблера распределение памяти осуществляется «вручную». При этом программист определяет для себя: какие участки памяти образуют какие ячейки; тип значения в каждой ячейке. Записывая затем программу, программист следит за корректностью использования типов данных в операциях их обработки.

При таком программировании средств автоматического контроля не существует, что приводит к многочисленным ошибкам в программе, которые вызываются несоответствием типов единиц данных.

Языки программирования высокого уровня подразумевают автоматическое распределение памяти под переменные и константы, на основании спецификации (описания) их типов. Компилятор при первом проходе составляет таблицу распределения памяти, и при втором проходе создает для каждого оператора программы соответствующий машинный код (Рис. I.23b).

4.8. Контрольные вопросы

1. Необходимость введения понятия типа данных.
2. Типизация целых чисел.
3. Специфика вычислений с данными целого типа.
4. Типизация символов и символьных строк.
5. Типизация вещественных чисел.
6. Специфика вычислений с данными вещественного типа.
7. Интерпретация хранящегося в памяти двоичного кода.

5. Сложение и вычитание целых неотрицательных чисел

В самом общем случае, арифметические операции над числами представляются нам в виде алгоритма, который определяет преобразование записи первого числа и записи второго числа в запись результата. При исполнении этого алгоритма производится определенная последовательность вычислений. Следует различать два случая:

1. Вычисления при сложении B-ичных чисел производятся в привычной для нас десятичной системе счисления;
2. Вычисления при сложении B-ичных чисел производятся в B-ичной системе.

Очевидно, что при сложении десятичных чисел оба эти случая совпадают.

5.1. Сложение B-ичных целых без знака (вычисления производятся в десятичной системе)

Рассмотрим сложение двух чисел, представленных в **B**-ичной системе счисления. Пусть **Z=X+Y**, где первое слагаемое **X**, второе слагаемое **Y** и сумма **Z** являются целыми неотрицательными числами без знака. Представим **i**-й разряд при сложении в следующем виде:

$$w_{i+1} \leftarrow x_i \leftarrow w_i$$
$$\underline{y_i}$$
$$z_i$$

Здесь, w_i перенос из предыдущего разряда в разряд **i**, w_{i+1} - перенос из **i**-го разряда в следующий разряд. Очевидно, что $w_0=0$.

Алгоритм I.9. Сложение целых без знака.

Преобразуем оба слагаемых в **B**-ично десятичную систему (цифры **B**-ичного числа представляются десятичными числами).

Для **i =0...n-1** производим следующие действия в десятичной системе.

1. Вычисляем вспомогательную сумму c_i:
$$c_i := (x_i + y_i) + w_i$$
2. Вычисляем цифру разряда и перенос в следующий разряд.
ЕСЛИ $(c_i < B)$ **ТО** $z_i := c_i$; $w_{i+1} := 0$ **ИНАЧЕ** $z_i := c_i - B$; $w_{i+1} := 1$
3. Результат вычислений преобразуем в систему с основанием **B**. (Десятичное число заменяем на соответствующую **B**-ичную цифру).

Блок – схема алгоритма изображена на Рис. I.24

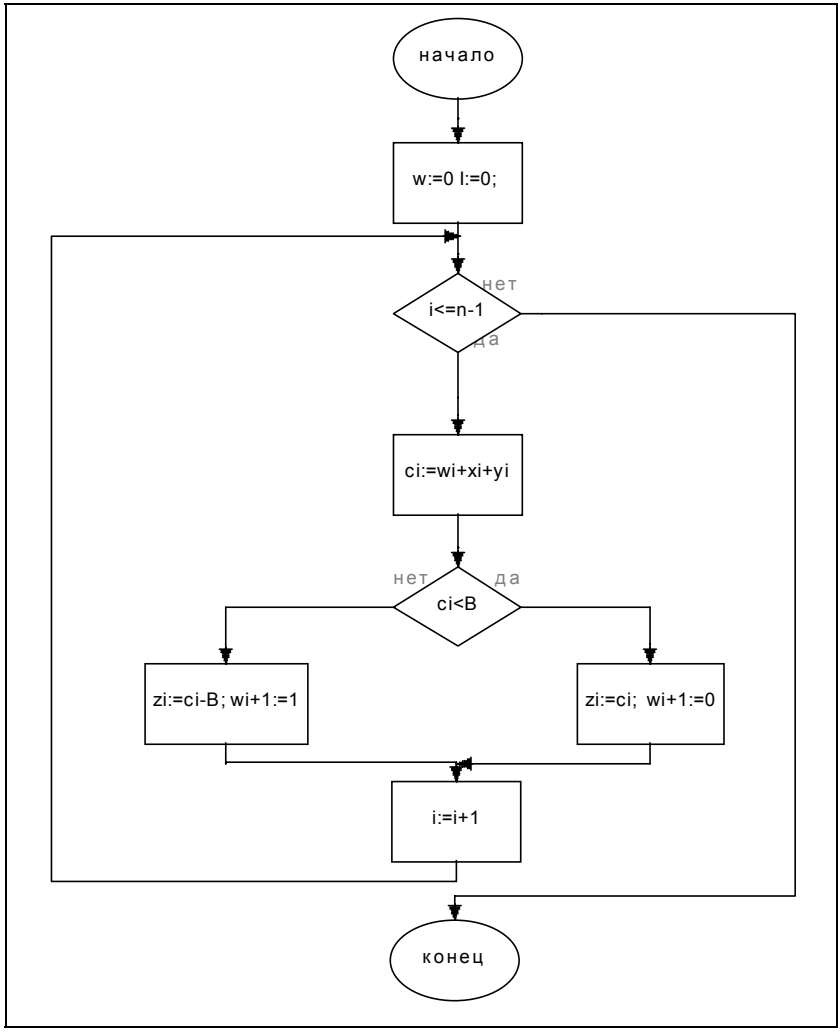


Рис. I.24. Блок - схема алгоритма сложения (вычисления в десятичной системе).

Примеры.

Представление чисел в различных системах кодирования.

Первое число: $x = (345)_{10} = (101011001)_2 = (531)_8 = (159)_{16}$.

Второе число: $y = (125)_{10} = (001111101)_2 = (175)_8 = (07D)_{16}$.

Результат: $z = (345)_{10} + (125)_{10} = (470)_{10}$.

Таблица I.8. Сложение в двоичной системе.

B =2	2-10 система	
101011001	10101100**1**	
+001111101	00111110**1**	$c_0>=2 \rightarrow z_0:=0$; перенос
	0	
	10101100**1**	
	00111110**1**	$c_1<2 \rightarrow z_1:=1$
	10	
	10101**0**01	
	00111**1**01	$c_2<2 \rightarrow z_2:=1$
	110	
	10101**1**001	
	00111**1**101	$c_3>=2 \rightarrow z_3:=0$; перенос
	0110	
	1010**1**1001	
	0011**1**1101	$c_4>=2 \rightarrow z_4:=1$; перенос
	10110	
	101**0**11001	
	001**1**11101	$c_5>=2 \rightarrow z_5:=0$; перенос
	010110	
	10**1**011001	
	00**1**111101	$c_6>=2; \rightarrow z_6:=1$; перенос
	1010110	
	1**0**1011001	
	0**0**1111101	$c_7<2 \rightarrow z_7:=1$
	11010110	
	101011001	
	001111101	$c_8<2 \rightarrow z_8:=1$
	111010110	
111010110		результат

Проверка: $(1*2^8+1*2^7+1*2^6+0*2^5+1*2^4+0*2^3+1*2^2+1*2^1+0*2^0)_2 = (470)_{10}$

Таблица I.9. Сложение в восьмеричной системе.

B = 8	8-10 система	
531	**531**	
+175	17**5**	$c_0<8 \rightarrow z_0:=6$
	6	
	5**3**1	
	1**7**5	$c_1 >= 8 \rightarrow z_1 := 2$: перенос
	26	
	531	
	175	$c_2 < 8 \rightarrow z_2 := 7$
	726	
726		результат

Проверка: $(7*8^2+2*8^1+6*8^0)_{10} = (470)_{10}$

Таблица I.10. Сложение в шестнадцатеричной системе

B = 16	16-10 система	
159	1 5 **9**	
+07D	0 7 **13**	$c_0 >= 16 \rightarrow z_0 := 6$; перенос
	6	
	1 **5** 9	
	0 **7** 13	$c_1 < 16 \rightarrow z_0 := 13$
	13 6	
	1 5 9	
	0 7 13	$c_2 < 16 \rightarrow z_2 := 1$
	1 13 6	
1D6		результат

Проверка: $(1*16^2+13*16^1+6*16^0)_{10} = (470)_{10}$

5.2. Вычитание B-ичных целых без знака (вычисления производятся в десятичной системе)

Рассмотрим вычитание двух чисел, представленных в B-ичной системе счисления. Пусть Z=X-Y, где уменьшаемое X, вычитаемое Y и разность Z являются целыми неотрицательными числами без знака. Представим i-й разряд при сложении в следующем виде:

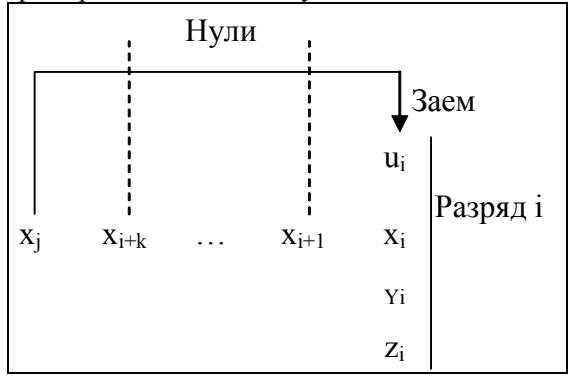


Рис. I.25. Вычитание B–ичных чисел

Здесь, x_i, y_i, z_i разряд уменьшаемого, вычитаемого и разности; x_{i+1}... x_{i+k} - разряды с нулевыми значениями; x_j - первый, после нулевых, разряд с ненулевым значением; u_i - значение займа.

Алгоритм I.10. Вычитание целых без знака.

Преобразуем уменьшаемое и вычитаемое в **B**-ично десятичную систему.

Устанавливаем нулевые значения займов для всех разрядов: $\mathbf{u_i} = 0$.
Для **i** = 0...n-1 производим следующие действия в десятичной системе.
Вычисляем вспомогательную разность $\mathbf{c_i}$:
$\mathbf{c_i = x_i - y_i}$
Вычисляем заем (если он есть) из предыдущего разряда.
ЕСЛИ($\mathbf{c_i<0}$) ТО $\mathbf{x_j := x_{j-1}; x_{i+k} := B-1; ...; x_{i+1} := B-1; u_i := B}$.
Вычисляем значение разности для разряда: $\mathbf{z_i := u_i + x_i - y_i}$.
Результат вычислений преобразуем в систему с основанием **B**.

Примеры.
Первое число: $x = (345)_{10} = (101011001)_2 = (531)_8 = (159)_{16}$
Второе число: $y = (125)_{10} = (001111101)_2 = (175)_8 = (07D)_{16}$
Результат: $z = (345)_{10} - (125)_{10} = (220)_{10}$

Таблица I.11. Вычитание в двоичной системе.

B = 2	2-10 система	
101011001	10101100**1**	
+001111101	00111110**1**	$c_0>=0 \rightarrow z_0 := 0$
	0	
	10101100**1**	
	00111110**1**	$c_1>=0 \rightarrow z_1:=0$
	00	
	10101**0**01	
	00111**1**01	$c_2<0$ заем
	10101**0**001	
	00111**1**101	$z_2 := 1$
	1 00	
	10101**0**001	
	00111**1**101	$c_{3<}0$ заем
	10100**0**001	
	00111**1**101	$z_3 := 1$
	1100	
	10100**0**001	
	00111**1**101	$c_4<0$ заем
	10010**0**001	
	00111**1**101	$z_4 := 1$
	11100	
	10010**0**001	
	00111**1**101	$c_5<0 \rightarrow z_5 := 0$
	011100	
	10**0**100001	

Таблица I.11. Продолжение

	00**1**111101	$c_6<0 \rightarrow$ заем
	01**0**100001	
	00**1**111101	$z_6 := 1$
	1011100	
	0**1**0100001	
	0**0**1111101	$c_{7>}=0 \rightarrow z_7 := 1$
	11011100	
	010100001	
	001111101	$c_8>=0 \rightarrow z_8 := 0$
	011011100	
011011100		Результат

Проверка $(0*2^8+1*2^7+1*2^6+0*2^5+1*2^4+1*2^3+1*2^2+0*2^1+0*2^0)_{10} = (220)_{10}$

Таблица I.12. Вычитание в восьмеричной системе.

B = 8	8-10 система	
531	531	$c_0<0 \rightarrow$ заем
-175	175	
	521	
	175	$z_0=4$
	4	
	521	
	175	$c_1<0 \rightarrow$ заем
	421	
	175	$z_1=3$
	34	
	411	
	175	$c_2>=0 \rightarrow z_2 = 3$
	334	
334		Результат

Проверка: $(3*8^2+3*8^1+4*8^0)8 = (220)_{10}$

Таблица I.13. Вычитание в шестнадцатеричной системе.

B = 16	16-10 система	
159	1 5 9	
-7B	0 7 13	$c_0 < 16 \rightarrow$ заем
	1 4 9	
	0 7 13	$z_0 =12$
	12	

Таблица I.13. Продолжение

	1 4 9	
	0 7 13	$c_1 < 0 \rightarrow$ заем
	0 4 9	
	0 7 13	
	13 12	
	0 4 9	
	0 7 13	$c_2>=0 \rightarrow z_2 :=0$
	0 13 12	
0DC		Результат

Проверка: $(0*16^2+13*16^1+12*16^0)_{10} = (220)_{10}$

5.3. Сложение с использованием таблицы сложения

Приведенный выше алгоритм не является машинным алгоритмом, т.к. не может исполняться без вмешательства человека. Дело в том, что он предполагает знание и выполнения “в уме” алгоритма разрядного сложения и, более того, алгоритма вычитания **B**-ичных чисел. Имеется возможность автоматизировать эти вычисления за счет использования таблицы сложения и заменить вычисления "в уме" формальной процедурой поиска в таблице сложения (вычитания).

Рассмотрим сначала поразрядное сложение, т.е. сложение в разряде **i**. Правила поразрядного сложения отражаются в **B**-ичной таблице сложения. Столбцы и строки такой таблицы соответствуют цифрам. На пересечении строки и столбца располагаются две цифры: вторая цифра - цифра суммы в разряде i; первая цифра - перенос в следующий разряд. Наиболее привычная из таблиц сложения - десятичная.

Таблица I.14. Десятичная таблица сложения

10	0	1	2	3	4	5	6	7	8	9
0	00	01	02	03	04	05	06	07	08	09
1	01	02	03	04	05	06	07	08	09	10
2	02	03	04	05	06	07	08	09	10	11
3	03	04	05	06	07	08	09	10	11	12
4	04	05	06	07	08	09	10	11	12	13
5	05	06	07	08	09	10	11	12	13	14
6	06	07	08	09	10	11	12	13	14	15
7	07	08	09	10	11	12	13	14	15	16
8	08	09	10	11	12	13	14	15	16	17
9	09	10	11	12	13	14	15	16	17	18
10	10	11	12	13	14	15	16	17	18	19

Таблица I.15. Восьмеричная таблица сложения

8	0	1	2	3	4	5	6	7
0	00	01	02	03	04	05	06	07
1	01	02	03	04	05	06	07	10
2	02	03	04	05	06	07	10	11
3	03	04	05	06	07	10	11	12
4	04	05	06	07	10	11	12	13
5	05	06	07	10	11	12	13	14
6	06	07	10	11	12	13	14	15
7	07	10	11	12	13	14	15	16
10	10	11	12	13	14	15	16	17

Таблица I.16. Шестнадцатеричная таблица сложения

16	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
0	00	01	02	03	04	05	06	07	08	09	0A	0B	0C	0D	0E	0F
1	01	02	03	04	05	06	07	08	09	0A	0B	0C	0D	0E	0F	10
2	02	03	04	05	06	07	08	09	0A	0B	0C	0D	0E	0F	10	11
3	03	04	05	06	07	08	09	0A	0B	0C	0D	0E	0F	10	11	12
4	04	05	06	07	08	09	0A	0B	0C	0D	0E	0F	10	11	12	13
5	05	06	07	08	09	0A	0B	0C	0D	0E	0F	10	11	12	13	14
6	06	07	08	09	0A	0B	0C	0D	0E	0F	10	11	12	13	14	15
7	07	08	09	0A	0B	0C	0D	0E	0F	10	11	12	13	14	15	16
8	08	09	0A	0B	0C	0D	0E	0F	10	11	12	13	14	15	16	17
9	09	0A	0B	0C	0D	0E	0F	10	11	12	13	14	15	16	17	18
A	0A	0B	0C	0D	0E	0F	10	11	12	13	14	15	16	17	18	19
B	0B	0C	0D	0E	0F	10	11	12	13	14	15	16	17	18	19	1A
C	0C	0D	0E	0F	10	11	12	13	14	15	16	17	18	19	1A	1B
D	0D	0E	0F	10	11	12	13	14	15	16	17	18	19	1A	1B	1C
E	0E	0F	10	11	12	13	14	15	16	17	18	19	1A	1B	1C	1D
F	0F	10	11	12	13	14	15	16	17	18	19	1A	1B	1C	1D	1E
10	10	11	12	13	14	15	16	17	18	19	1A	1B	1C	1D	1E	1F

Таблица I.17. Двоичная таблица сложения

2	0	1
0	00	01
1	01	10
10	10	11

Теперь мы можем привести алгоритм автоматического сложения двух **B**-ичных чисел (Рис. I.26).

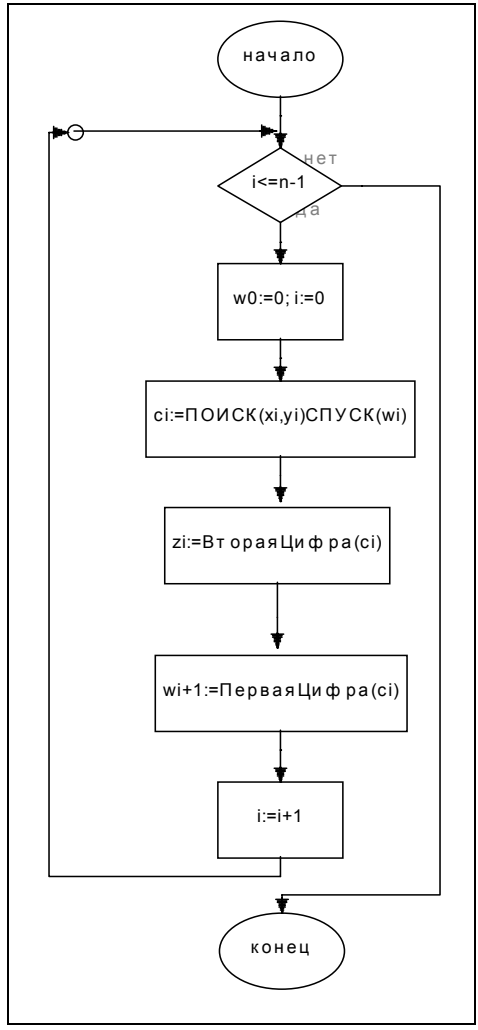


Рис. I.26. Блок схема алгоритма сложения с использованием таблицы сложения

При вычислении разрядной суммы $\mathbf{C_i}$ используется **B**-ичная таблица сложения: реализуется поиск ячейки таблицы по заданным $\mathbf{x_i}$ и $\mathbf{y_i}$ и, в случае наличия единицы переноса из младшего разряда – спуск на одну ячейку вниз. Такой алгоритм является машинным, т.к. может исполняться без вмешательства человека.

По сути дела, этот алгоритм и реализуется электронными схемами компьютера для двоичного сложения, как будет подробно рассмотрено в соответствующем месте (глава I.6.2).

5.4. Контрольные вопросы

1. Алгоритм сложения целых неотрицательных чисел (вычисления в десятичной системе).
2. Алгоритм вычитания целых неотрицательных чисел (вычисления в десятичной системе).
3. Почему эти алгоритмы не являются машинными алгоритмами?
4. Алгоритм сложения целых неотрицательных чисел, использующий таблицы сложения.

6. Автоматическое исполнение команды обработки данных

6.1. Символьный характер арифметических операций

Рассмотрим для примера операцию сложения двух целых чисел. Здесь, **a** и **b** - абстрактные числа.

Дано:

- запись (последовательность цифр) первого слагаемого: **Запись(a)**;
- запись (последовательность цифр) второго слагаемого: **Запись(b)**.
- Выполнить преобразование записи двух слагаемых в запись результата сложения, обеспечивающее равенство:

$$\textbf{Запись(a)+ Запись(b) = Запись (a "сложить" b).} \quad \textbf{(I.27)}$$

Здесь "сложить" - абстрактная операция сложения целых чисел, + - операция преобразования записей, реализующая сложение целых чисел.

Хорошо известен алгоритм сложения «столбиком» чисел в десятичной системе счисления. На его примере мы видим, что сложение сводится к преобразованию двух символьных строк в результирующую символьную строку. Аналогично для других систем счисления.

Более подробно. Первое слагаемое (абстрактное число) представляется в позиционной системе счисления каноническим выражением:

$$E(a) = a_{n-1}B^{n-1}+a_{n-2}B^{n-2}+ \ldots +a_1B^1+a_0B^0. \quad \textbf{(I.28)}$$

Точно также, второе слагаемое (абстрактное число) представляется каноническим выражением:

$$E(b) = b_{m-1}B^{m-1}+b_{m-2}B^{m-2} \ldots +b_1B^1+b_0B^0. \quad \textbf{(I.29)}$$

Алгоритм сложения должен получить каноническое выражение суммы абстрактных чисел:

$$E(a \oplus b) = c_{P-1}B^{P-1}+c_{P-2}B^{P-2}+ \ldots +c_1B^1+c_0B^0 \quad \textbf{(I.30)}$$

Сказанное выше проиллюстрировано на Рис. I.27.

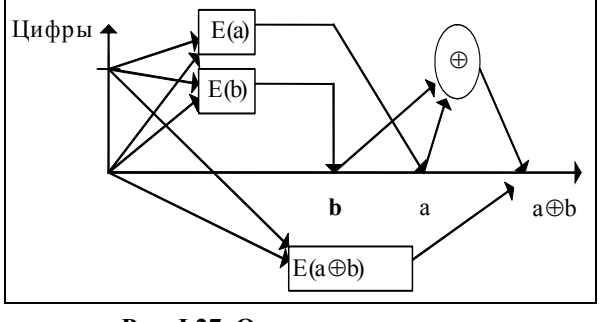


Рис. I.27. Операция сложения

Аналогично, для остальных арифметических операций:

- **Запись(a) - Запись(b) = Запись (a "минус" b).**
- **Запись(a) * Запись(b) = Запись (a "умножить" b).**
- **Запись(a) / Запись(b) = Запись (a "разделить" b).**

Здесь, кавычками обозначены абстрактные операции над числами.

6.2. Сложение целых положительных чисел

Мы знаем, что оперативная память являет собой последовательность байт. Но каждый байт - это последовательность восьми бит. Так что в конечном итоге, содержимое оперативной памяти представляется в виде последовательности бит (двоичной последовательности). Будем называть такую последовательность **двоичным вектором памяти**. Предположим, что записи - операнды сложения, хранятся в ячейках **a**, **b** оперативной памяти, запись - результат сложения хранится в ячейке **c**. Алгоритм сложения реализуется путем преобразования $\mathbf{F_+}$исходного состояния двоичного вектора памяти **M(t)** в заключительное состояние $\mathbf{M(T+\tau)}$ за время τ (Рис. I.28).

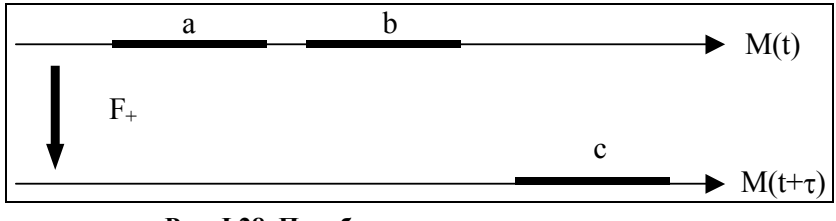


Рис. I.28. Преобразование вектора памяти

Не вводя дополнительных предположений, мы можем лишь утверждать, что каждый **i**-й разряд результата сложения определяется всеми разрядами слагаемых. Таким образом, операцию сложения можно представить в виде множества функций:

$$c_0 = F_+^0 (a_{n-1},...,a_i,...,a_0,b_{n-1},...b_i,...,b_0)$$

$$\dots$$

$$c_i = F_+^i (a_{n-1},...,a_i,...,a_0,b_{n-1},...b_i,...,b_0) \qquad \text{(I.31)}$$

$$\dots$$

$$c_{n-1} = F_+^{n-1} (a_{n-1},...,a_i,...,a_0,b_{n-1},...b_i,...,b_0)$$

Таким образом, для получения **i** - го разряда результата используется функция:

$$c_i = F_+^i (a_{n-1},...,a_i,...,a_0,b_{n-1},...b_i,...,b_0) \qquad \text{(I.32)}$$

Аргументы функции - суть логические переменные. Значение функции - также логическое. Таким образом, значение разряда результата является значением логической (булевской) функции.

Заметим, что **i**-й разряд результата определяется не всеми разрядами слагаемых, а лишь младшими разрядами, от нулевого до **i**-го:

Для учета значений разрядов от **0**-го до **i -1**-го вводится специфическая переменная w_i - перенос в **i**-й разряд. Таким образом, сложение в одном разряде описывается логической функцией:

$$c_i = F^i (w_i, a_i, b_i). \qquad \textbf{(I.33)}$$

Аналогично, перенос в следующий разряд описывается логической функцией:

$$w_{i+1} = W^{i+1}(w_i, a_i, b_i) \qquad \textbf{(I.34)}$$

Теперь можно конструировать электронную схему, вычисляющую значение **i**-го разряда суммы.

Запишем таблицу значений (график) функции сложения F_i и таблицу значений (график) функции переноса W_{i+1}:

Таблица I.18. Таблицы функции сложения и функции переноса

w_i	a_i	b_i	c_i
0	0	0	0
0	0	1	1
0	1	0	1
0	1	1	0
1	0	0	1
1	0	1	0
1	1	0	0
1	1	1	1

w_i	a_i	b_i	w_{i+1}
0	0	0	0
0	0	1	0
0	1	0	0
0	1	1	1
1	0	0	0
1	0	1	1
1	1	0	1
1	1	1	1

В математической логике доказана теорема: любая логическая функция может быть записана в виде композиции трех элементарных логических функций: отрицания, дизъюнкции и конъюнкции.

Отрицание (¬) - функция одной переменной. Ее значение равно единице в случае равенства нулю единственного аргумента и нулю в противном случае.

Дизъюнкция или логическое сложение (∨) - функция двух переменных. Ее значение равно единице в случае равенства единице хотя бы одного аргумента.

Конъюнкция или логическое умножение (&) - функция двух переменных. Ее значение равно единице тогда и только тогда, когда оба аргумента равны единице.

Если имеется табличное задание логической функции, можно получить ее представление в виде композиции отрицания, дизъюнкции и конъюнкции. Алгоритм получения такого представления проиллюстрируем на примере таблицы логической функции сложения (Таблица I.18). В результате исполнения алгоритма получается так называемая **дизъюнктивная нормальная форма записи логической функции** (ДНФ).

Алгоритм I.11. Алгоритм вычисления ДНФ

1. Вычеркиваем все строки таблицы, значение функции в которых равно 0.
2. Выписываем конъюнкции переменных для оставшихся строк:

$$w_i\&a_i\&b_i \quad w_i\&a_i\&b_i \quad w_i\&a_i\&b_i \quad w_i\&a_i\&b_i.$$

3. Если значение переменной в строке таблицы, соответствующей конъюнкции, равно нулю - записываем отрицание этой переменной:

$$\neg w_i\&\neg a_i\&b_i \quad \neg w_i\&a_i\&\neg b_i \quad w_i\&\neg a_i\&\neg b_i \quad w_i\&a_i\&b_i.$$

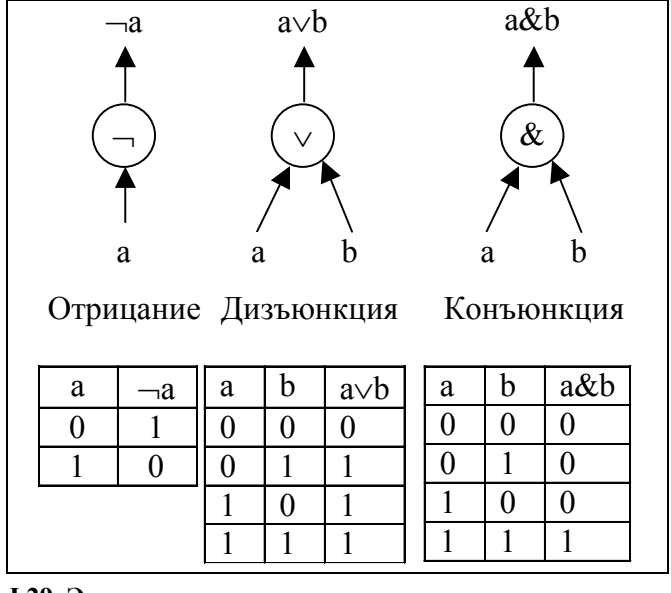
4. Соединяем конъюнкции знаками дизъюнкции - получаем ДНФ логической функции сложения:

$$c_i = \neg w_i\&\neg a_i\&b_i \vee \neg w_i\&a_i\&\neg b_i \vee w_i\&\neg a_i\&\neg b_i \vee w_i\&a_i\&b_i. \quad \text{(I.35)}$$

Проделав аналогичные действия для таблицы переносов, получаем ДНФ функции переноса:

$$w_{i+1} = \neg p_i\&a_i\&b_i \vee p_i\&\neg a_i\&b_i \vee p_i\&a_i\&\neg b_i \vee p_i\&a_i\&b_i. \quad \text{(I.36)}$$

Названные выше три элементарных логических функции замечательны еще тем, что существуют простые электронные элементы, вычисляющие значения этих функций (Рис. I.29).



a	¬a
0	1
1	0

a	b	a∨b
0	0	0
0	1	1
1	0	1
1	1	1

a	b	a&b
0	0	0
0	1	0
1	0	0
1	1	1

Рис. I.29. Электронные элементы - вычислители элементарных функций

Теперь можно нарисовать функциональную электронную схему (Рис. I.30), вычисляющую логическую функцию сложения$c_i = \neg w_i\&\neg a_i\&b_i \vee \neg w_i\&a_i\&\neg b_i \vee w_i\&\neg a_i\&\neg b_i \vee w_i\&a_i\&b_i$. (I.35).

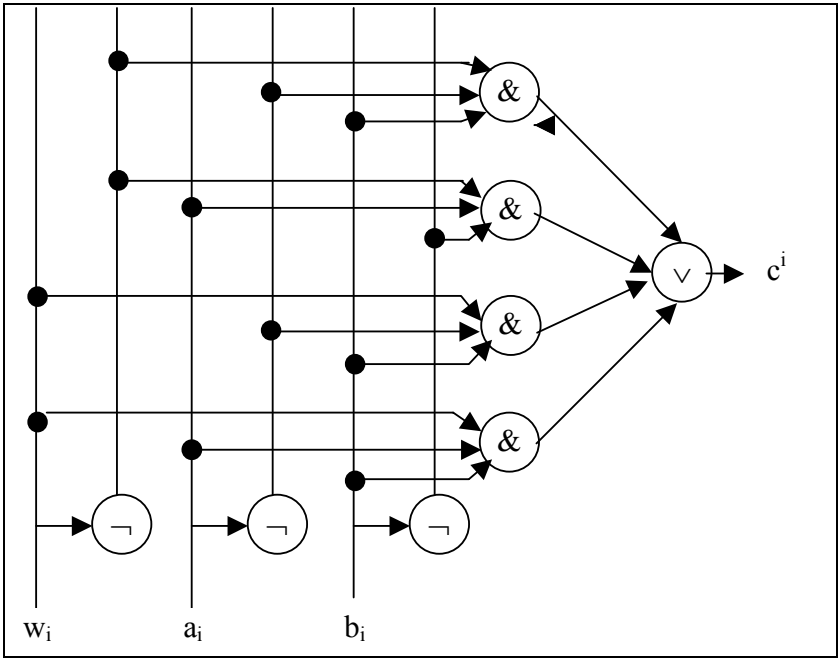


Рис. I.30. Функциональная схема сложения

Аналогичным образом получаем функциональную электронную схему, вычисляющую логическую функцию переноса $w_{i+1} = \neg p_i \& a_i \& b_i \vee p_i \& \neg a_i \& b_i \vee p_i \& a_i \& \neg b_i \vee p_i \& a_i \& b_i$. (I.36).

6.3. Арифметико-логическое устройство и команда обработки данных

Мы будем говорить, что электронная схема, реализующая вычисление логической функции, является физической моделью логической функции. При этом собственно электронная схема задает функцию (физический аналог таблицы функции), а функционирование схемы - реализует процесс вычисления функции.

Заметим, что $\mathbf{a_i}$, $\mathbf{b_i}$, $\mathbf{c_i}$ - биты двоичного вектора оперативной памяти (разряды ячеек **a**, **b**, **c**). Но минимальной адресуемой единицей памяти является регистр памяти (ячейка памяти). Следовательно, возможности адресовать каждый бит памяти у нас нет.

Рассуждая теоретически, можно построить память с адресацией каждого разряда - бита. Однако на практике это приведет к такому увеличению числа образующих память электронных элементов, что ее реализация станет невозможной. Поэтому конструкторы компьютеров пошли другим путем.

Для реализации автоматического исполнения операций (и сложения в частности) конструируется **арифметико-логическое устройство** (**АЛУ**), в состав которого входят три регистра: **R1** - хранение первого операнда, **R2** - хранение второго операнда, **R3** - хранение результата (Рис. I.31). И прежде чем автоматически исполнять операцию, ее операнды выбираются из оперативной памяти и пересылаются на регистры **R1**, **R2**. После выполнения операции ее результат с регистра **R3** пересылается в оперативную память.

При этом между **АЛУ** и оперативной памятью передаются сразу все разряды регистра памяти, для чего требуется лишь адресация ячейки оперативной памяти. А адресация каждого бита необходима лишь в **АЛУ**. Но это адресация лишь **3n** битов, она реализуется лишь для регистров **R1**, **R2**, **R3** и не представляет технической проблемы.

Примечание. Старший разряд регистра (закрашенный черным цветом) используется для хранения **знака числа**.

Таким образом, физические модели функции сложения и функции переноса в качестве своих входов и выходов используют не значения битов оперативной памяти, а значения разрядов регистров **АЛУ**.

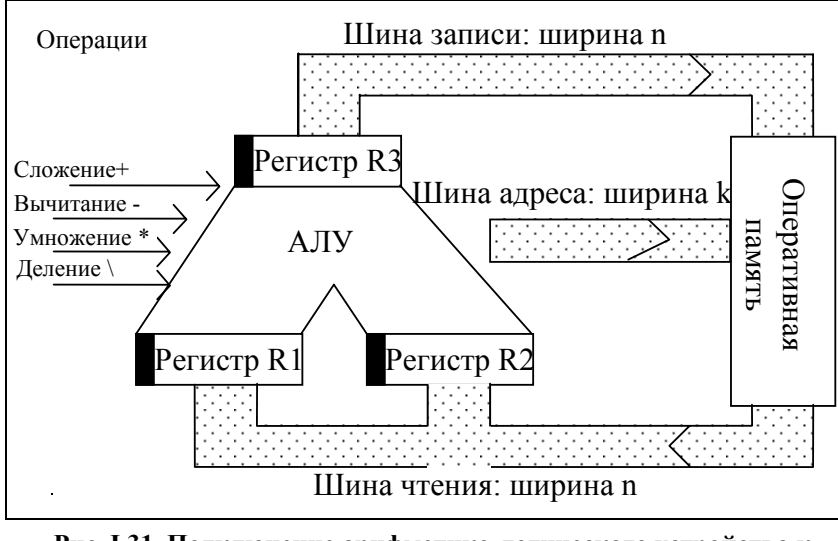


Рис. I.31. Подключение арифметико-логического устройства к оперативной памяти

Объединение модели функции сложения и модели функции переноса образует конструктивный элемент **АЛУ** - функциональную схему **одноразрядного сумматора** (Рис. I.32). Здесь, $\mathbf{R_1^i}$, $\mathbf{R_2^i}$, $\mathbf{R_3^i}$ - разряды регистров **АЛУ**.

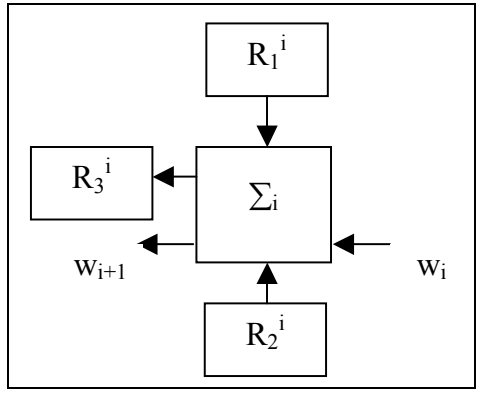


Рис. I.32. Одноразрядный сумматор Σ_i

Последовательность из **n** одноразрядных сумматоров, образует **полный сумматор** (Рис. I.33) - физическую модель арифметической функции сложения (операции арифметического суммирования двух **n** - разрядных двоичных кодов). Говоря другими словами, полный сумматор является устройством, автоматически исполняющим операцию суммирования двух **n** - разрядных целых неотрицательных чисел.

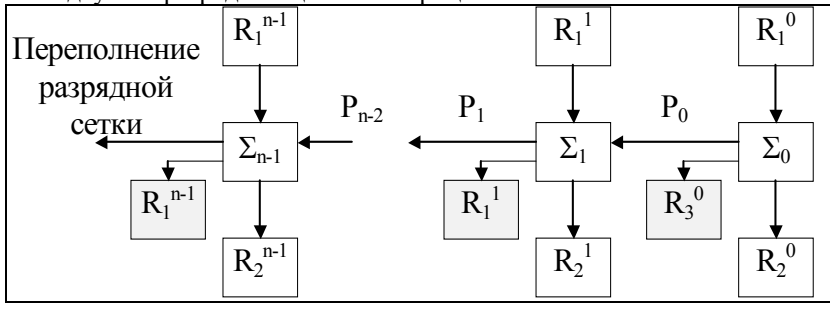


Рис. I.33. Полный сумматор

Замечание 1. Мы уже говорили, что старший разряд сумматора является знаковым разрядом, т.е. представляет знак числа (ноль - плюс, единица - минус). Знаковый разряд участвует в сложении наряду с другими битами (смотри 5.2). В случае сложения положительных чисел, знаковые разряды имеют нулевые значения, и знаковый разряд результата также равняется нулю.

Замечание 2. Если результат сложения «не убирается» в разрядную сетку, наступает ситуация переполнения разрядной сетки. Признаки переполнения обсуждаются в соответствующем месте (глава I.5.2).

Специфика связи **АЛУ** - оперативная память приводит к необходимости введения понятия **команды обработки данных**, в полях которой указываются: операция, адреса операндов, адрес результата.

В данном случае такой командой является команда сложения:

КОП$_+$: A1 ,A2 ,A3. **(I.37)**

Здесь, **КОП$_+$** - код операции сложения, **A1** - адрес в оперативной памяти первого операнда , **A2** - адрес в оперативной памяти второго операнда, **A3** - адрес в оперативной памяти результата.

6.4. Сложение положительных и отрицательных чисел

При сложении чисел разных знаков приходится использовать операцию вычитания. Следуя рассмотренной выше методике, можно построить электронную схему, осуществляющую вычитание двух целых положительных чисел. Но это приведет к существенному увеличению сложности **АЛУ**. Если использовать специальные коды для представления отрицательного числа, этого усложнения можно избежать. Более того, использование этих кодов позволяет не только реализовать сложение чисел произвольных знаков, но также просто реализовать операцию вычитания чисел произвольных знаков.

Прямой, **обратный** и **дополнительный коды числа** определяются следующим образом. Здесь и далее: **S** - код знака числа, **N** - код величины числа.

Положительное число.

(дополнительный и обратный код совпадают с прямым кодом).

S(прямой код) = **S**(обратный код) = **S**(дополнительный код) = 0; **N**(прямой код) = **N**(обратный код) = **N**(дополнительный код) = двоичный код абсолютной величины числа.

Отрицательное число.

S(прямой код) = **S**(обратный код) = **S**(дополнительный код) = 1;

N(прямой код) = двоичный код абсолютной величины числа;

N(обратный код) = Инверсия от **N**(прямой код);

N(дополнительный код) = **N**(обратный код) + 1.

Примечание. В оперативной памяти числа хранятся в прямом коде.

Примеры кодирования чисел приведены в Таблица I.19.

Таблица I.19. Коды положительного и отрицательного числа

Положит. число		Отрицат. число		
S	N	S	N	
+	5	-	5	Число
0	101	1	101	Прямой код
0	101	1	010	Обратный код
0	101	1	011	Дополнительный код

Сложение чисел в дополнительном коде с учетом знака иллюстрирует Таблица I.20. Здесь, **R1** и **R2** - регистры **АЛУ,** хранящие операнды сложения, **R3** регистр **АЛУ** - результат сложения, **С** - промежуточный результат.

Используются функции: **Доп** - преобразование прямого кода отрицательного числа в дополнительный код; **Пр** - преобразование дополнительного кода отрицательного числа в прямой код.

Преобразование **Доп(R)** прямого кода отрицательного числа в дополнительный код реализуется в два этапа:

1. Производя инверсию величины числа **N** (заменяя нули на единицы и единицы на нули) получаем обратный код числа;
2. Прибавляя к обратному коду единицу, получаем дополнительный код.

Таблица I.20. Сложение - чисел в дополнительном коде

R1	R2	
+	+	R3:=R1+R2;
-	+	C:=Доп(R1)+R2; Если Знак(C)=1 то R3:=Пр(C) иначе R3:=C
+	-	C:=R1+Доп(R2); Если Знак(C)=1 то R3:=Пр(C) иначе R3:=C
-	-	C:=Доп(R1)+Доп(R2); R3:=Пр (C)

Преобразование **Пр(С)** дополнительного кода отрицательного числа в прямой код также реализуется в два этапа:

1. Инверсия дополнительного кода величины числа **N**;
2. Прибавление единицы к результату.

Особо обратим внимание, что при сложении участвует знаковый разряд, тогда как при преобразовании из прямого кода в дополнительный и обратно знаковый разряд не изменяется. Таблица I.21 содержит примеры сложения в дополнительном коде для положительных и отрицательных чисел.

Как уже говорилось ранее, использование дополнительного кода позволяет обойтись в составе **АЛУ** только сумматором. Для перевода из прямого в дополнительный код и обратно используется операция прибавления единицы (реализованная в сумматоре) и операция инверсии (реализация которой неизмеримо проще реализации операции вычитания).

Более того, при использовании дополнительного кода, проблемы "двух различных кодов нуля", как в прямом коде, не существует (Таблица I.21).

Таблица I.21. Иллюстрация сложения в дополнительном коде

	Преобразование в дополнительный код			Преобразование в прямой код		Результат
числа	ПК	ОК	ДК	Инверсия	+1	
+5 -2 +3	0`101 1`010	0`101 1`101	0`101 1`110 0`011	-	-	0`011
-5 +2 -3	1`101 0`010	1`010 0`010	1`011 0`010 1`101	1`010	1`011	1`011
+5 +2 +7	0`101 0`010	0`101 0`010	0`101 0`010 0`111	-	-	0`111
-5 -2 -7	1`101 1`010	1`010 1`101	1`011 0`010 1`001	1`110	1`111	1`111
+0 -0	0`000 1`000	0`000 1`111	0`000 0`000	Код нуля в дополнительном коде единственен		

Код знака числа отделяется от кода величины числа кавычкой: **S'N.**

Очевидно, что вычитание чисел произвольных знаков путем логического анализа вариантов сводится к суммированию чисел в дополнительном коде.

В заключение обсудим проблему переполнения разрядной сетки (Таблица I.22) при сложении чисел произвольных знаков. Можно назвать два равносильных признака переполнения:

1. Слагаемые имеют одинаковые знаки, и знак суммы отличается от знака слагаемых;
2. Значения битов переноса в знаковый разряд и из знакового разряда различны.

Таблица I.22. Переполнение разрядной сетки.

Число	Дополнительный код
-3	1.101
-6	1.010
-9	10.111
+5	0.101
+5	0.101
+10	1.010

6.5. Арифметические команды

Арифметические команды обеспечивают автоматическое выполнение арифметических операций над данными, хранящимися в оперативной памяти. Схемная реализация операций сложения и вычитания рассмотрена выше.

Операции умножения и деления целых чисел выполняются по обычным алгоритмам - "столбиком". Для их реализации **АЛУ** должно обеспечивать, наряду с операциями сложения - вычитания, также операции сдвига двоичного кода влево и вправо. Также можно представить процесс выполнения операций арифметики над вещественными числами (формат с плавающей точкой). При этом выполняется выравнивание порядков, затем производятся операции над мантиссами и, в заключение, определяется значение порядка результата.

Арифметические команды позволяют вычислять значения элементарных вещественных функций многих вещественных аргументов (формализация физических задач).

6.6. Логические команды

Однако не меньшее значение при вычислениях на компьютере имеют **логические команды**, позволяющие решать проблемы принятия решений.

Задачи принятия решений, явно или неявно, сводятся к вычислению логического выражения. Логическое выражение представляет собой комбинацию трех элементарных логических функций (отрицание, дизъюнкция, конъюнкция) и предикатов – отношений (равно, не равно, больше, меньше, больше или равно, меньше или равно). Предикат – отношение определяется следующим образом: <арифметическое выражение> <знак отношения> <арифметическое выражение>. Он имеет два возможных значения: истина, ложь – и связывает "мир"формальной логики с "миром" вещественных функций. Логические выражения используются в качестве условий для операторов условного перехода и операторов цикла.

Для программирования логических вычислений используются три команды: инверсия, логическое сложение и логическое умножение.

Общепринято, что в качестве операнда таких команд выступает не отдельный бит памяти, а слово памяти. Команды отношения для двух слов (сравнение, больше, меньше и т.д.) определяются очевидным образом. Тогда как логические команды определяются специфически.

Команда инверсии (поразрядное отрицание) во всех битах слова заменяет

Команда логического сложения (поразрядная дизъюнкция) выполняется по следующему алгоритму: $\mathbf{C_i = a_i \vee b_i ; i = 1 \div 8.}$

Команда логического умножения (поразрядной конъюнкции) выполняется по следующему алгоритму: $\mathbf{C_i = a_i \& b_i ; i = 1 \div 8.}$

6.7. Команда обработки данных и ее автоматическое исполнение

Подводя итог предыдущему обсуждению, скажем, что арифметико-логическое устройство (АЛУ) реализует исполнение **команд обработки (преобразования) данных**. Они подразделяются на четыре класса.

1. Арифметические и логические команды формата: **КОП A1,A2,A3** - выполняют операции сложения и вычитания по следующему алгоритму:
   - выбрать из оперативной памяти содержимое ячейки **A1**- значение первого аргумента;
   - выбрать из оперативной памяти содержимое ячейки **A2** - значение второго аргумента;
   - произвести определяемую **КОП** (код операции) операцию над аргументами;
   - записать результат в оперативную память по адресу **A3**.
2. Команда пересылки данных формата: **КОП A1,0,A3.**
3. Команды сдвига кода в регистре и команда обращения кода в регистре.
4. Команда ввода формата: **ВВОД 0,0,A3**. Код нажатой на клавиатуре клавиши вводится в байт оперативной памяти с адресом A3.
5. Команда вывода формата: **ВЫВОД A1,0,0**. Символ, код которого хранится в байте оперативной памяти, выводится на терминал.

В Приложении 1 приведена функциональная схема исполнения команды обработки данных.

Для хранения текущей исполняемой команды используется **регистр команд** (**РК**). Учитывая формат команды, он подразделяется на следующие поля: код операции **КОП**; **A1**, **A2** - адреса ячеек оперативной памяти, в которых хранятся операнды операции; **A3** - адрес ячейки оперативной памяти, куда следует записать результат исполнения операции.

Как сказано выше, **АЛУ** соединена с оперативной памятью шинами передачи данных (шина записи и шина чтения). Если **n** - ширина шины передачи данных, то по ней одновременно передаются **n** бит данных. Адресные поля регистра команд соединены с оперативной памятью шиной адреса. Если ширина шины адреса **p**, то размер адресного пространства 2^p. Т.е. имеется возможность адресации 2^p байт оперативной памяти.

Реализация автоматического исполнения находящейся на **РК** команды осуществляется **Устройством Управления Автоматическим Выполнением Команды** (**УАВК**). Это устройство для каждой команды вырабатывает специфическую последовательность электронных импульсов (сигналов), которые принято называть микрооперациями. Каждая **микрооперация** заставляет работать вполне определенный электронный блок компьютера.

Функционально микрооперации объединяются в **микрокоманды**. Таким образом, каждая команда обработки данных представляется как последовательность микрокоманд, а каждая микрокоманда - как последовательность микроопераций.

Пример представления команды сложения в виде последовательности микрокоманд и реализация микрокоманд в виде последовательности микроопераций приведен в Таблица I.23.

Таблица I.23. Микрокоманды и микрооперации, реализующие команду сложения

Микрокоманда	Микрооперации	Комментарий
R1 :=ПАМ(А1)	ВДА1	Выдача адреса первого операнда в память
	ЧТ	Выполнение чтения по адресу
	ПРR1	Прием считанного из ОП кода на регистр R1
R2 :=ПАМ(А2)	ВДА2	Выдача адреса второго операнда в память
	ЧТ	Выполнение чтения по адресу
	ПРR2	Прием считанного из ОП кода на регистр R2
R3 :=R1+R2	+	Выполнение операции сложения
ПАМ(А3) := R3	ВДА3	Выдача адреса результата в память
	ВДR3	Выдача на шину записи результата
	ЗП	Выполнение записи по адресу

Так же в Приложении 1 изображен дешифратор кода операции (**ДШКОП**). Код операции команды, находящейся на регистре команд, вызывает появление единичного сигнала на одном и только одном его выходе. И сигнал на этом выходе предписывает **УУАВК** выполнить вполне определенную последовательность микроопераций, т.е. вполне определенную команду обработки данных.

Рассмотрим исполнение команд ввода - вывода. Предполагается, что к шине записи подключена клавиатура, а к шине чтения - дисплей.

Начав выполняться, **команда ввода** ждет нажатия какой либо клавиши. После нажатия клавиши, двоичный код ей соответствующий, поступает на регистр ввода $\mathbf{R_{ввод}}$ и затем записывается в ячейку памяти с адресом **А3** (Таблица I.24).

Начав выполняться, **команда вывода** ждет окончания вывода предыдущего символа. После этого, двоичный код символа, хранящейся в ячейке с адресом **А1**, передается в регистр вывода $\mathbf{R_{вывод}}$ и выводится в виде символа на дисплей (Таблица I.25).

Таблица I.24. Команда ввода: ВВОД 0, 0, А3

Микрокоманда	Микрооперации	Комментарий
ОЖДКЛ		Ожидание нажатия клавиши
Пам(А3) := $R_{ввод}$	ВДВВОД	Выдача двоичного кода литеры на шину
	ВДА3	Выдача на шину адреса А3
	ЗП	Инициация записи по адресу

Таблица I.25. Команда вывода: ВЫВОД А1, 0, 0

микрокоманда	микрооперации	комментарий
ОЖДвывода		ожидание ввода
$R_{вывод}$:= ПАМ(А1)	ВДА1	выдача адреса вводимого кода
	ЧТ	инициация чтения
	ПРВЫВОД	прием кода на регистр вывода

Аналогично представляются и выполняются все остальные команды обработки данных.

6.8. Контрольные вопросы

1. Каким образом алгоритмы преобразования символьных строк обеспечивают исполнение арифметических операций?
2. Команды обработки данных.
3. Исполнение команды сложения.
4. Арифметико-логическое устройство и его функционирование.
5. Сложение и вычитание целых чисел.
6. Арифметическое умножение и деление.
7. Логические команды.
8. Автоматическое исполнение команды обработки данных.

7. Автоматическое исполнение программы

7.1. Понятие линейной программы

Фундаментальным понятием информатики является понятие алгоритма. **Алгоритм** решения прикладной задачи однозначно определяет способ преобразования исходных данных в результирующие данные. По сути дела, решение прикладной задачи и заключается в таком преобразовании данных.

При конструировании алгоритма предполагается, что существует **исполнитель**, способный выполнять конечное множество элементарных действий по преобразованию данных. Исполнителем может быть как человек, так и техническое устройство. Обозначение элементарного действия по преобразованию данных называется **командой преобразования данных** исполнителя. Спецификой алгоритма является представление сложного преобразования любого допустимого исходного набора данных **d** в результирующий набор данных **r** в виде последовательности элементарных действий – команд исполнителя. Такую последовательность **σ(d)** будем называть **траекторией вычислительного процесса** решения прикладной задачи.

Описание траектории вычислительного процесса на формальном языке программирования и называется **линейной программой**. Очевидно, что линейная программа состоит только из команд обработки данных (арифметические, логические, ввод, вывод) и заканчивается специальной командой останова.

7.2. Принцип программного управления

Следует сказать, что понятие программы не является следствием изобретения компьютеров. Человечество программировало вычисления задолго до появления подобных вычислительных машин. Исполнителем таких программ являлся человек, оснащенный, в лучшем случае, средством автоматического исполнения арифметических операций (арифмометр, счеты, логарифмическая линейка).

Появившийся задолго до изобретения компьютеров арифмометр также способен автоматизировать только исполнение арифметических операций.

Настоящая революция в области автоматизации вычислений произошла после того, как американский ученый фон Нейман сформулировал свои принципы построения автоматической вычислительной машины. По сути дела, данное учебное пособие последовательно излагает эти принципы.

И теперь пришла пора рассмотреть самый революционный из них - принцип программного управления вычислительной машиной.

Принцип программного управления фон-Неймана формулируется в виде следующих пунктов.

1. Программа преобразования данных, представленная в двоичном виде, размещается в той же оперативной памяти, что и данные;
2. Электронная схема устройства управления реализует алгоритм главного цикла компьютера, который обеспечивает последовательную автоматическую выборку из оперативной памяти команд программы и их исполнение.

Примечание. Строго говоря, практически одновременно были предложены два способа хранения программы. Компьютер Гарвардского университета «Марк-1» (1950 год) имел отдельную память для команд. Тогда как в Принстонском университете при участии фон Неймана (начиная с 1947-го года) создавались компьютеры с единой памятью для команд и для данных. Принстонская архитектура стала преобладающей.

Как поместить программу в оперативную память? Вспомним, что программа - последовательность команд. Каждая команда имеет формат:

КОП A1,A2,A3.

Здесь, **КОП** - код операции, **A1**, **A2**, **A3** - адреса ячеек оперативной памяти (**p**- разрядные двоичные последовательности).

Каждый код операции также можно представить **r**- разрядной двоичной последовательностью. Таким образом, для представления команды используется **r+3×p** двоичных разрядов. Как правило, команда "не убирается" в один байт и размещается в ячейке размером в несколько байт. **Адрес команды** определяется как адрес первого байта хранящей эту команду ячейки. **Длина команды** - число занимаемых командой байт.

Таким образом, в оперативной памяти образуются три области:

1. Область хранения программы;
2. Область хранения данных;
3. Свободная область.

Команды линейной программы должны выполняются последовательно, начиная с первой команды. Необходимо физически реализовать отношение следования команд линейной программы. Т.е. необходимо решить проблему указания того, какая команда должна исполняться после выполнения текущей команды. Здесь возможны два способа такого указания.

Во - первых, можно добавить еще одно (четвертое) поле адреса $\mathbf{A_{слк}}$, в котором указывается следующая исполняемая команда:

КОП A1,A2,A3,$A_{слк}$.

В этом случае мы имеем дело с четырехадресной командой, а линейная программа образует связный список (Рис. I.34). Образно говоря, программа становится похожей на нитку бус, разбросанных по памяти, где в качестве бусин выступают команды программы.

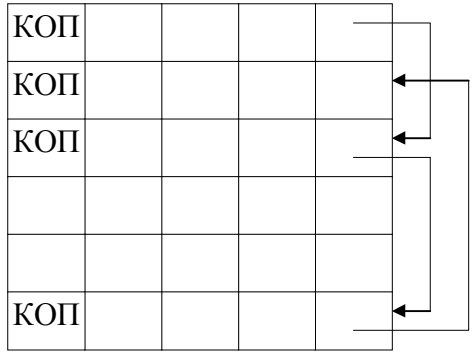


Рис. I.34. Хранение команд в виде связного списка

Преимуществом такого способа хранения программы является возможность "разбрасывания" команд по памяти произвольным образом. А платой за такую произвольность является большой расход памяти, необходимой для хранения адресов следующих команд.

Если принять достаточно естественное условие, что следующая команда линейной программы хранится в следующей ячейке оперативной памяти, то можно обойтись **трехадресной командой**.

При этом достаточно иметь лишь один регистр - **счетчик команд** (**СЧК**) - на котором хранится адрес следующей исполняемой команды. Очевидно, что для перехода к следующей команде счетчик команд необходимо увеличить на длину команды в байтах (**Δ**).

7.3. Автоматическое исполнение линейной программы

Вычислительная машина с трехадресной системой команд исполняет линейную программу (состоящую только из команд обработки данных), последовательно выбирая команды из памяти и передавая их на регистр команд для автоматического исполнения. Таким образом, после исполнения очередной команды начинает исполняться следующая команда.

Архитектура такой трехадресной, программно - управляемой вычислительной машины приведена в Приложении 2. На рисунке показано, что шина чтения подходит к регистру команд, а счетчик команд соединен с шиной адреса.

При исполнении команды обработки данных, устройство управления автоматическим выполнением команд (**УУАВК**) генерирует последовательность микроопераций, которые объединяются в микрокоманды.

Исполнение линейной программы обеспечивается **Устройством Управления Автоматическим Выполнением Программы** (**УУАВП**), которое реализует алгоритм главного цикла вычислительной машины (Таблица I.26).

Алгоритм главного цикла вычислительной машины реализуется посредством генерации повторяющейся последовательности микрокоманд. Как и прежде, микрокоманда реализуется в виде последовательности микроопераций.

Таблица I.26. Реализация алгоритма главного цикла компьютера

	Микрокоманда	Микрооперация	Комментарий
1	СЧК := адрес первой команды.	ВДА2	Выдача адреса первой команды на шину адреса
		ПРСЧК	Прием адреса первой команды на счетчик команд
2	РК := ПАМ(СЧК)	ВДСЧК	Выдача адреса следующей команды на шину адреса
		ЧТ	Инициация чтения из оперативной памяти
		ПРРК	Прием считанной из оперативной памяти команды на РК
3	СЧК := СЧК+Δ	+ΔСЧК	Подготовка к чтению следующей команды
4	ВДКОП	ВДКОП	Расшифровка кода операции
5	ЕСЛИ КОП(РК) = "ОСТАНОВ"	останов	Окончание исполнения программы
6	запуск УУАВК	запуск команды	Автоматическое исполнение команды
7	повторение 2	пуск	Повторение главного цикла

Особого рассмотрения заслуживает вопрос присвоения перед началом исполнения счетчику команд адреса первой исполняемой команды программы. Мы предполагаем, что этот адрес заносится непосредственно на регистр команд (**РК**) с пульта - управления вычислительной машиной. И после этого нажимается кнопка "Пуск", вызывающая запуск **УУАВП**

7.4. Понятие программы и ее автоматическое исполнение

Выше мы рассмотрели представление вычислительного процесса решения прикладной задачи в виде единственной траектории, что привело нас к понятию линейной программы.

Однако, практически все преобразования данных отличаются следующим свойством: способ преобразования исходного набора данных зависит от конкретных значений данных этого набора.

Рассмотрим пример определения корней квадратного уравнения: $\mathbf{Ax^2+Bx+C = 0}$. Известно, что, в зависимости от конкретных значений коэффициентов уравнения, его корни вычисляются по трем различным траекториям.

Таким образом, можно утверждать, что, в общем случае, для каждой прикладной задачи существует множество траекторий вычислительного процесса. И это множество сплошь и рядом бесконечное. ***Конечное компактное описание множества траекторий вычислительного процесса мы называем моделью вычислительного процесса решения прикладной задачи.***

Программа – одна из возможных моделей вычислительного процесса. Она представляет собой линейный текст (последовательность команд), записанный в соответствии с правилами формального языка программирования. Другой известной моделью вычислительного процесса является блок - схема.

Учитывая вышесказанное, становится очевидным, что линейных программ практически не бывает. И можно говорить лишь о линейных участках программ. Однако текст программы по-прежнему остается линейным. Для того чтобы в конечном компактном виде (в виде линейного текста программы) представить множество траекторий вычислительного процесса, в систему команд исполнителя должны быть включены команды, изменяющие порядок выполнения команд преобразования данных. Такие команды принято называть **командами управления**.

Примечание. Таким образом, последовательность записи команд в тексте программы не совпадает с последовательностью выполнения команд при вводе конкретных исходных данных – траекторией вычислительного процесса. Последовательность записи команд в тексте программы единственная – траекторий множество.

Множество команд (обработки данных и управления) образует **систему команд компьютера**.

Как говорилось выше, в систему команд вычислительной машины входят команды управления. Прежде всего, это **команда безусловной передачи управления:**

БПУ 0,A2,0.

Смысл этой команды состоит в том, что после ее исполнения начинает исполняться не следующая команда, а команда с адресом **A2**.

Устройство управления автоматическим выполнением команды для исполнения команды безусловного перехода генерирует последовательность из двух микроопераций:

ВДА2; ПРСЧК.

После этих действий продолжение главного цикла обеспечивает исполнение команды с адресом **A2**.

Команда условной передачи управления имеет формат:

УПУ 0,A2,0.

В качестве условия, которое определяет передачу управления на команду с адресом **A2**, используется отношение к нулю результата предыдущей операции. Результат предыдущей операции может быть равен нулю, не равен нулю, больше нуля, меньше нуля, больше или равен нулю, меньше или равен нулю. Для каждого такого отношения имеется своя команда условного перехода со специфическим кодом операции **УПУ**.

Если отношение выполняется, то команда условной передачи управления работает как команда безусловной передачи управления, т.е. происходит переход по адресу **A2**. В противном случае последовательность выполнения команд не нарушается и исполняется команда, следующая в тексте программы за командой условного перехода.

7.5. Система команд вычислительной машины

7.5.1. Классификация команд

Множество команд, каждую из которых может выполнить центральный процессор, образует **систему команд вычислительной машины**. В общем случае формат команды состоит из одного поля кода операции и **n** адресных полей, в каждом из которых записывается адрес оперативной памяти. Существенной характеристикой команды является ее адресность, т.е. число адресных полей.

Код операции	addr1	. . .	addrn

Система команд существенным образом определяет тип (архитектуру) вычислительной машины. Однако, вне зависимости от конкретной архитектуры, команды вычислительной машины подразделяются на два класса: команды обработки данных и команды управления выполнением вычислительным процессом. К командам обработки данных относятся:

- команды преобразования данных (пересылка, арифметические и логические команды).
- команды сдвига кода;
- команды ввода данных в оперативную память;
- команды вывода данных из оперативной памяти;

Команды управления реализуют безусловную и условную передачу управления, а также обращение к подпрограммам и возврат из подпрограмм.

Команды обработки данных можно классифицировать по числу адресов; наиболее часто используются одноадресные и двухадресные команды. Команды управления, как правило, одноадресные.

7.5.2. Команды обработки данных

Одноадресные команды пересылки реализуют обмен данными между оперативной памятью и регистром арифметического устройства. Причем последний задается по умолчанию.

Load addr – загрузка содержимого ячейки оперативной памяти в регистр.

Story addr – пересылка содержимого регистра в ячейку оперативной памяти.

Двухадресные команды пересылки осуществляют обмен данными между ячейками оперативной памяти и\или регистрами арифметического устройства.

MOV $A_{ист}$, $A_{пр}$ – обмен данными между источником и приемником.

Арифметические и логические команды в современных компьютерах являются также двухадресными.

ADD $A_{ист}$, $A_{пр}$ – выбирает значения из $A_{ист}$ и $A_{пр}$, образует их сумму, которую записывает в $A_{пр}$.

Команды ввода – вывода либо имеют специфический одноадресный формат, либо совпадают по формату с двухадресной командой пересылки **MOV**.

7.5.3. Команды управления

Одноадресная команда безусловной передачи управления **JMP addr** заносит адрес "скачка" на счетчик команд, что обеспечивает в качестве следующей команды с этим адресом. Одноадресная команда **CALL addr** – реализует вызов подпрограммы с запоминанием адреса возврата. Безадресная команда **RET** обеспечивает возврат из подпрограммы.

Команды условной передачи управления предназначены для проверки определенного условия и, в случае его выполнения, осуществляют передачу управления.

В качестве условий используются разряды регистра флагов автоматически устанавливаемые процессором в соответствии с результатом выполнения каждой команды.

7.6. Принципы функционирования вычислительной машины - компьютера

Мы рассмотрели фундаментальные основы построения вычислительных машин (компьютеров), сконструированных и функционирующих на основании принципов фон - Неймана. Эти принципы сформулировал в середине сороковых годов двадцатого века американский ученый фон - Нейман, который входил в группу создателей одной из первых автоматических цифровых вычислительных машин.

Классическое построение автоматической цифровой вычислительной машины показано на Рис. I.35. Внешние устройства подразделяются на устройства ввода – вывода (монитор – дисплей, клавиатура и т.п.) и на внешние запоминающие устройства (магнитные диски, лазерные диски и т.д.)

Конструкция автоматической цифровой вычислительной машины, очевидная с современной точки зрения, содержит две фундаментальные идеи, которые сыграли выдающуюся роль в развитии вычислительной техники и которые не утратили своего значения и сейчас.

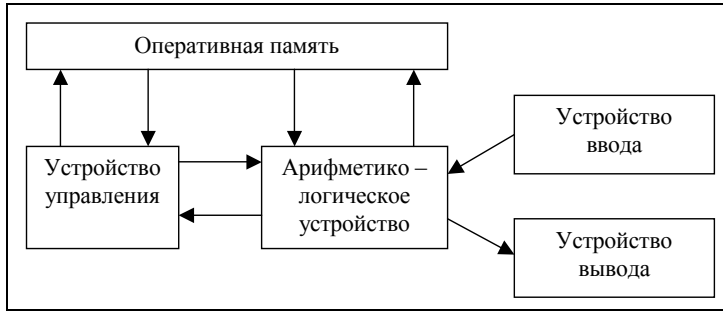


Рис. I.35. Классическое построение автоматической цифровой вычислительной машины.

Первая из них состоит в том, что программа вводится через те же внешние устройства, что и данные и хранится в той же памяти, что и данные. Это обеспечивает:

1. Возможность автоматического исполнения программы;
2. Оперативную перестройку машины с одной задачи на другую без внесения каких - либо изменений в схему машины, делая компьютер универсальным инструментом обработки данных.

Вторая очень важная идея состоит в том, что составляющие программу команды закодированы в виде двоичных последовательностей и по форме ничем не отличаются от данных, с которыми оперирует компьютер. Это дает возможность в необходимых случаях рассматривать программу как набор данных, который можно преобразовывать посредством исполнения другой программы. Следствием этого является возможность построения средств автоматического программирования и средств автоматического управления вычислительным процессом.

Принципы фон - Неймана в сжатом виде формулируются следующим образом:

1. Информация кодируется в двоичной форме и представляется в виде наборов данных. Набор данных разделяется на единицы данных, называемые словами.

2. Разнотипные слова различаются по способу использования, но не по способу кодирования.
3. Слова размещаются в ячейках памяти машины и идентифицируются номерами ячеек, называемыми адресами ячеек (адресами слов).
4. Алгоритм представляется в форме последовательности управляющих слов, которые определяют наименование операции и слова, участвующие в операции. Каждое такое управляющее слово называется командой. Алгоритм, представленный в виде последовательности машинных команд, называется программой.
5. Выполнение преобразования данных, предписанных алгоритмом, сводится к последовательному выполнению команд в порядке, однозначно определяемом программой и конкретным набором исходных данных.

Таким образом, очень распространенное сейчас слово "компьютер" означает: цифровая, универсальная, программно - управляемая вычислительная машина.

Здесь следует сделать два замечания. Во - первых, не все вычислительные машины являются цифровыми. Существует класс аналоговых вычислительных машин, работа которых основана на общности законов, описывающих процессы различной природы.

Так, например, можно построить электронную схему, колебания которой подчиняются тем же закономерностям, что и колебания маятника. И, вместо того, чтобы качать реальный маятник, можно изучать его поведение на электронной аналоговой вычислительной модели (машине) . Это возможно потому, что колебания маятника и колебания электронной схемы описываются одним и тем же дифференциальным уравнением. Самое интересное, что таким же дифференциальным уравнением описывается поведение биологической системы "хищник - жертва" и даже колебательные химические реакции.

В сороковых - пятидесятых годах XX-го века электронные аналоговые вычислительные машины создавали серьезную конкуренцию только что появившимся компьютерам. Но аналоговые машины не выдержали конкуренции с компьютерами по двум причинам:

1. Ограниченность круга решаемых задач (нет возможности решать логические задачи);
2. Малая точность вычислений.

Какое-то время аналоговые машины еще выигрывали гонку в быстродействии вычислений. Но колоссальные темпы роста объема памяти и быстродействия компьютера обеспечили ему приоритет и по этому параметру.

Во - вторых, не все программно - управляемые машины являются универсальными, т.е. допускают ввод и исполнение любой программы. Есть специализированные компьютеры, которые ориентированы на решение одной единственной задачи.

Например, компьютер управления артиллерийским зенитным огнем должен решать единственную задачу встречи снаряда с летящим объектом. Смысл использования таких компьютеров - относительная дешевизна изготовления и простота эксплуатации.

И, наконец, насчет еще одного названия: цифровая, универсальная, программно - управляемая вычислительная машина в нашей стране называлась электронной вычислительной машиной (ЭВМ).

7.7. Контрольные вопросы

1. Команды обработки данных и команды управления.
2. Понятие траектории вычислительного процесса.
3. Понятие линейной программы и программы.
4. Принцип программного управления.
5. Команда, микрокоманда и микрооперация.
6. Главный цикл программно - управляемой вычислительной машины.
7. Устройство автоматического исполнения программы.
8. Принципы фон – Неймана.

Глава II. Совершенствование архитектуры вычислительной машины - компьютера

1. Организация управления компьютером

1.1. Принципы и алгоритмы управления

Рассматривая проблемы реализации автоматического исполнения команды и автоматического исполнения программы, мы остановились на следующей терминологии:

- программа → последовательность команд;
- команда → последовательность микрокоманд;
- микрокоманда → последовательность микроопераций.
- микрооперация → двоичный электронный сигнал, включающий или выключающий вполне определенную подсхему в электронной схеме компьютера.

Таким образом, для того, чтобы выполнить программу, необходимо сгенерировать последовательность микроопераций, обеспечивающую реализацию алгоритма главного цикла (Таблица II.1).

Таблица II.1. Микрокоманды и микрооперации, реализующие алгоритм главного цикла компьютера

	Микрокоманда	Микрооперация (мо)	Комментарий
1	СЧК := адрес первой команды.	мо1: ВДА2	Выдача адреса первой команды на шину адреса
		мо2: ПРСЧК	Прием адреса первой команды на счетчик команд
		мо3:ПСКГЦ	Пуск главного цикла
2	РК := ПАМ(СЧК)	мо4:ВДСЧК	Выдача адреса текущей команды на шину адреса
		мо5:ЧТ	Инициация чтения из оперативной памяти
		мо6:ПРРК	Прием считанной команды на РК
3	СЧК := СЧК+Δ	мо7: +ΔСЧК	Подготовка к чтению следующей команды
4	ВДКОП	мо7: ВДКОП	Расшифровка кода операции
5	ЕСЛИ КОП(РК) = "ОСТАНОВ"	мо8: останов	То же, что нажатие кнопки "Останов"
6	Запуск УУАВК	мо9: запуск УУАВП	Автоматическое исполнение команды
7	Повторение 2	мо10: пуск	Повторение главного цикла

Аналогично, для того, чтобы автоматически выполнить команду, необходимо сгенерировать последовательность микрокоманд, реализующую алгоритм ее исполнения (Таблица II.2).

Таблица II.2. Микрокоманды и микрооперации выполнения команды сложения

	Микрокоманда	Микрооперации	Комментарий
1	R1 :=ПАМ(A1)	мо11: ВДА1	Выдача адреса первого операнда в память
		мо5: ЧТ	Выполнение чтения по адресу
		мо12: ПРR1	Прием считанного из ОП кода на регистр R1
2	R2 :=ПАМ(A2)	мо1: ВДА2	Выдача адреса второго операнда в память
		мо5: ЧТ	Выполнение чтения по адресу
		мо13: ПРR2	Прием считанного из ОП кода на регистр R2
3	R3 :=R1+R2	мо14: +	Выполнение операции сложения
4	ПАМ(A3) := R3	мо15: ВДА3	Выдача адреса результата в память
		мо16: ВДR3	Выдача на шину записи единицы данных
		мо17: ЗП	Выполнение записи по адресу

1.2. Схемотехническая реализация управления

Каждый алгоритм (в том числе алгоритм главного цикла и алгоритм выполнения команды) может иметь или схемотехническую или программную реализацию. В первом случае конструируется электронная схема, функционирование которой обеспечивает исполнение алгоритма. Во втором случае алгоритм реализуется в виде программы для некоторой программно - управляемой вычислительной машины.

При схемотехнической реализации для каждого конкретного алгоритма необходимо конструировать уникальную схему. Это дорого, но обеспечивает минимальное время исполнения алгоритма. При программной реализации аппаратура вычислительной машины создается один раз, а для реализации конкретного алгоритма конструируется соответствующая программа. Но время исполнения алгоритма становится больше за счет многошагового исполнения алгоритма.

Схемотехническая реализация управляющего устройства компьютера приведена на Рис. II.1. Имеется генератор тактовых импульсов, который задает повторяющуюся дискретную последовательность: $\mathbf{t_1,t_2,...t_k,t_1,t_2...t_k}$...

Аппаратура, реализующая алгоритм главного цикла, после нажатия кнопки "Пуск" генерирует последовательность микроопераций (Таблица II.1):**мо1, мо2, мо3, мо4, мо5, мо6, мо7, мо8, мо9 мо4, мо5, мо6, мо7, мо8, мо9**

Микрооперация **мо7** устанавливает счетчик команд на следующую команду и одновременно инициирует выдачу кода операции с регистра команд **РК** на дешифратор команды. Если на регистре команд находится команда останова, то на одном из выходов дешифратора команд появляется единичный сигнал, играющий роль микрооперации **мо8** (прекращение выполнения главного цикла, т.е. останов компьютера).

В противном случае, микрооперация **мо9** запускает устройство автоматического исполнения команды. Это устройство генерирует ту или иную последовательность микроопераций, обеспечивающую исполнение команды, находящейся на регистре команд.

Например, для исполнения команды сложения должна быть сгенерирована следующая последовательность микроопераций (Таблица II.2): **мо11, мо5, мо12, мо1, мо5, мо13, мо14, мо15, мо16, мо17, мо10**.

Заметим, что эта последовательность заканчивается микрооперацией **мо10**, обеспечивающей повторение главного цикла (Таблица II.1).

И так продолжается до тех пор, пока в программе не встретится команда останова, которая породит микрооперацию **мо8** (то же, что и кнопка "Останов").

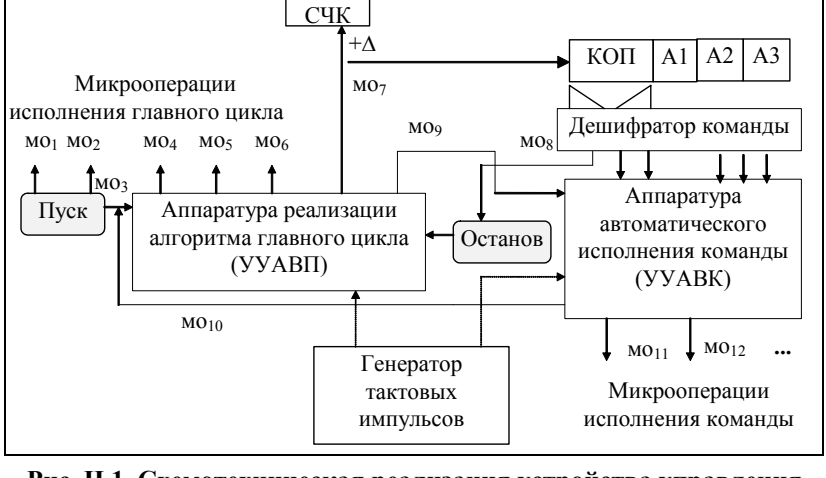


Рис. II.1. Схемотехническая реализация устройства управления компьютером

1.3. Микропрограммная реализация управления

Выше мы рассмотрели схемотехническую реализацию алгоритма управления компьютером в виде электронной схемы (Рис. II.1). Основной недостаток этого способа - чрезвычайно сложная и неоднородная электронная схема управления. Однако, управление первых компьютеров осуществлялось вышеизложенным способом до тех пор, пока не перешли к более экономной и гибкой методике микропрограммного управления.

Микропрограммирование основано на представлении команды в виде последовательности микрокоманд, каждая из которых при своем выполнении генерирует последовательность микроопераций (Глава II.1.1). Каждой команде компьютера поставлена в соответствие **микропрограмма**, состоящая из последовательности микрокоманд, способных сгенерировать необходимую для исполнения команды последовательность микроопераций.

Устройство микропрограммного управления (**МПУ**) представляет собой встроенный в основной компьютер специализированный управляющий компьютер, в качестве оперативной памяти использующий **постоянное запоминающее устройство** (**ПЗУ**). В **ПЗУ** хранятся микропрограммы, соответствующие командам основного компьютера.

По сути дела, исполнение команды основного компьютера реализуется посредством исполнения соответствующей ей микропрограммы устройством микропрограммного управления. При выполнении микропрограммы генерируется последовательность микроопераций, необходимая для исполнения команды основного компьютера. Алгоритм главного цикла вычислительной машины также представляется в виде микропрограммы.

Все микрокоманды имеют одинаковый формат (Рис. II.2):

Рис. II.2. Формат микрокоманды

Число управляющих бит в микрокоманде равно числу всех микроопераций, необходимых для реализации управления. Напомним, что микрооперация - это двоичный сигнал, осуществляющий включение - выключение вполне определенного элемента электронной схемы. Если некоторый управляющий бит имеет значение "**1**", то это означает генерацию соответствующей микрооперации при выполнении микрокоманды.

Таким образом, микрокоманды отличаются друг от друга количеством управляющих бит, в которых находятся единицы.

Особенностью микрокоманды является наличие в ней поля адреса следующей микрокоманды. Таким образом, микропрограмма представляет собой связный список микрокоманд.

Последовательность микрокоманд (Таблица II.1) назовем микропрограммой главного цикла. Последовательность микрокоманд (Таблица II.2) назовем микропрограммой команды сложения. Аналогично, для каждой команды компьютера существует соответствующая микропрограмма.

Архитектура устройства микропрограммного управления изображена на Рис. II.3. Работу устройства микропрограммного управления синхронизует генератор тактовых импульсов.

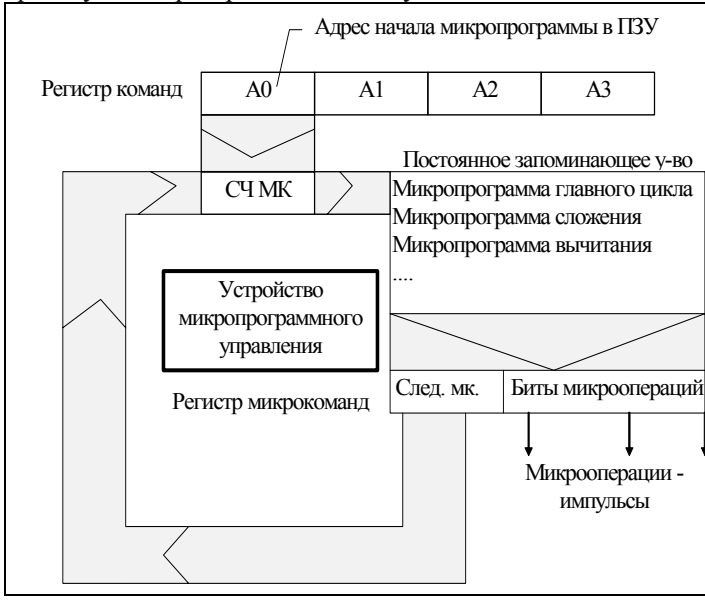


Рис. II.3. Структура микропрограммного управления

Пусть находящаяся на регистре микрокоманд (**РМК**) микрокоманда имеет единицы в **s** управляющих битах. Тогда эта микрокоманда выполняется за (**s+1**) такт. Первые **s** тактов порождают микрооперации, соответствующие единичным управляющим битам, а последний такт обеспечивает прием на регистр микрокоманд **РМК** следующей микрокоманды (с адресом **АМК**).

Мы по-прежнему считаем, что команда компьютера является трехадресной. Только вместо поля кода операции (**КОП**) используется поле адреса начала соответствующей микропрограммы в **ПЗУ** (Рис. II.3).

Естественно, что программу составляет программист, а микропрограмма создается конструктором на стадии проектирования компьютера. Множество микропрограмм (представляющих систему команд компьютера) "зашивается" в ПЗУ при изготовлении компьютера, и это множество одинаково для всех машин одного типа.

Если микропрограммное управление являет собой программно - управляемый компьютер, то существует алгоритм главного цикла микропрограммного управления (Таблица II.3).

Таблица II.3. Микропрограмма главного цикла

1: СЧМК := α
2: РМК := ПЗУ(СЧМК)
3: Исполнение микрокоманды - генерация микроопераций
4: РМК := АМК.РМК
5: Повторение 2

Алгоритм главного цикла устройства микропрограммного управления реализуется электронной схемой, которая гораздо проще электронной схемы аппаратного управления компьютером (Рис. II.1). Функционирование устройства микропрограммного управления обеспечивает исполнение любой программы обработки данных, введенной в оперативную память компьютера.

Содержимое ПЗУ микропрограммного управления приведено в Таблица II.4, выполнение программы показано в Таблица II.65 и Таблица II.76.

Таблица II.4. Содержимое постоянного запоминающего устройства

Адрес ячейки ПЗУ	Содержимое ячейки - микрокоманда	Микропрограмма
α	СЧК := адрес первой выполняемой команды	главного цикла
α+1	РК := ПАМ(СЧК)	""
α+2	СЧК := СЧК + Δ	""
α+3	ЕСЛИ КОП(РК) = "СТОП" ТО останов	""
α+4	СЧМК:= РК.А0	""
β	R1:=ПАМ(А1)	сложения
β+1	R2:=ПАМ(А2)	""
β+2	R3:=R1+R2	""
β+3	ПАМ(А3):=R3	""
β+4	СЧМК:= α+1	""
γ	R1:=ПАМ(А1)	вычитания
γ+1	R2:=ПАМ(А2)	""
γ+2	R3:=R1-R2	""
γ+3	ПАМ(А3):=R3	""
γ+4	СЧМК:= α+1	""

Таблица II.5. Программа основного компьютера в оперативной памяти

Адрес в ОП	Адрес микро-программы	Первый ад-рес	Второй адрес	Третий адрес
1	β	A1	A2	A3
2	γ	A1	A2	A3
3	β	A1	A2	A3

Таблица II.6. Исполнение программы основного компьютера

СЧК	СЧМК	РК	РМК
	α		СЧК:=1
1	α+1		РК:=ПАМ(СЧК)
1	α+2	β,A1,A2,A3	СЧК:=СЧК+1
2	α+3	β,A1,A2,A3	не останов
2	α+4	β,A1,A2,A3	СЧМК:= β
2	β	β,A1,A2,A3	R1:=ПАМ(A1)
2	β+1	β,A1,A2,A3	R2:=ПАМ(A2)
2	β+2	β,A1,A2,A3	R3:=R1+R2
2	β+3	β,A1,A2,A3	ПАМ(A3):=R3
2	β+4	β,A1,A2,A3	СЧМК:= α+1
2	α+1	β,A1,A2,A3	РК:=ПАМ(СЧК)
2	α+2	γ,A1,A2,A3	СЧК:=СЧК+1
3	α+3	γ,A1,A2,A3	не останов
3	α+4	γ,A1,A2,A3	СЧМК:=γ
3	γ	γ,A1,A2,A3	R1:=ПАМ(A1)
3	γ+1	γ,A1,A2,A3	R2:=ПАМ(A2)
3	γ+2	γ,A1,A2,A3	R3:=R1-R2
3	γ+3	γ,A1,A2,A3	ПАМ(A3):=R3
3	γ+4	γ,A1,A2,A3	СЧМК:= α+1
…	…	…	…

Мы уже говорили, что микропрограммная реализация управления компьютером более простая и дешевая по сравнению со схемотехнической реализацией. Имеются еще два преимущества микропрограммного управления. Вспомним, что система команд компьютера определяется набором микропрограмм, которые хранятся, как правило, в постоянном запоминающем устройстве (**ПЗУ**).

Ничто не мешает использовать при работе на одном и том же компьютере различные **ПЗУ**.

Это позволяет:

- на современном компьютере - эмулировать команды старого компьютера, обеспечивая преемственность программного обеспечения;
- на современном - компьютере - эмулировать команды вновь разрабатываемого компьютера, обеспечивая одновременную разработку технического и программного обеспечения вновь создаваемого компьютера.

1.4. Контрольные вопросы

1. Команда, микрокоманда и микрооперация.
2. Электронная реализация управления.
3. Понятие микропрограммы.
4. Микропрограммное управление.
5. Микропрограммная реализация алгоритма главного цикла компьютера.

2. Иерархия памяти компьютера

Практически с момента начала эксплуатации первых серийных вычислительных машин возникла проблема дефицита объема оперативной памяти. Емкость оперативной памяти от одного до четырех килобайт не позволяла размещать в ней программы нужного размера и данные требуемого объема. Расширение объема оперативной памяти ограничивалось как техническими сложностями, так и высокой стоимостью изготовления одного регистра. Как первое, так и второе определялось используемыми физическими процессами запоминания и технологиями изготовления элементарной базы равнодоступных запоминающих устройств.

Долгое время для конструирования оперативной памяти использовались ферритовые кольца, прошитые проводами адресации, чтения и записи. Для реализации запоминающих устройств большой емкости и с гораздо меньшей стоимостью изготовления одного регистра был использован принцип записи сигнала на движущийся магнитный носитель, который использовался в аудимагнитофонах. Сначала это были магнитные ленты и магнитные барабаны, затем появились магнитные диски. В настоящее время к ним добавились лазерные диски и флэш-память. Такой класс устройств получил название **внешние запоминающие устройства** (по отношению к оперативной памяти).

Более того, полезность внешних запоминающих устройств заключается в том, что они являются долговременными запоминающими устройствами, в отличие от оперативной памяти, в которой при отключении электрического питания данные исчезают.

Второе же свойство внешних запоминающих устройств, как уже отмечалось выше – большая емкость и дешевизна хранения данных. За все надо платить, и платой за эти достоинства является большое время доступа, т.е. малое быстродействие внешних запоминающих устройств.

Дело в том, что внешние запоминающие устройства (за исключением флеш – памяти) используют движущиеся носители данных и, в силу этого, не являются равнодоступными. Время доступа к байту на магнитном диске или лазерном диске варьируется в очень больших пределах, но в среднем является неизмеримо большим по сравнению с временем доступа к байту оперативной памяти (Рис. II.4a). Поэтому прямой "побайтовый" обмен центрального процессора с внешним запоминающим устройством нерационален.

Доступ к данным на внешних устройствах центральный процессор получает не непосредственно, а через оперативную память (Рис. II.4b). Массив данных, которыми за один сеанс обмениваются внешнее запоминающее устройство и оперативная память, называется **записью**. Запись может состоять из нескольких полей. Группы записей, размещенные на внешних устройствах, называются **файлами**. Каждый файл идентифицируется уникальным именем.

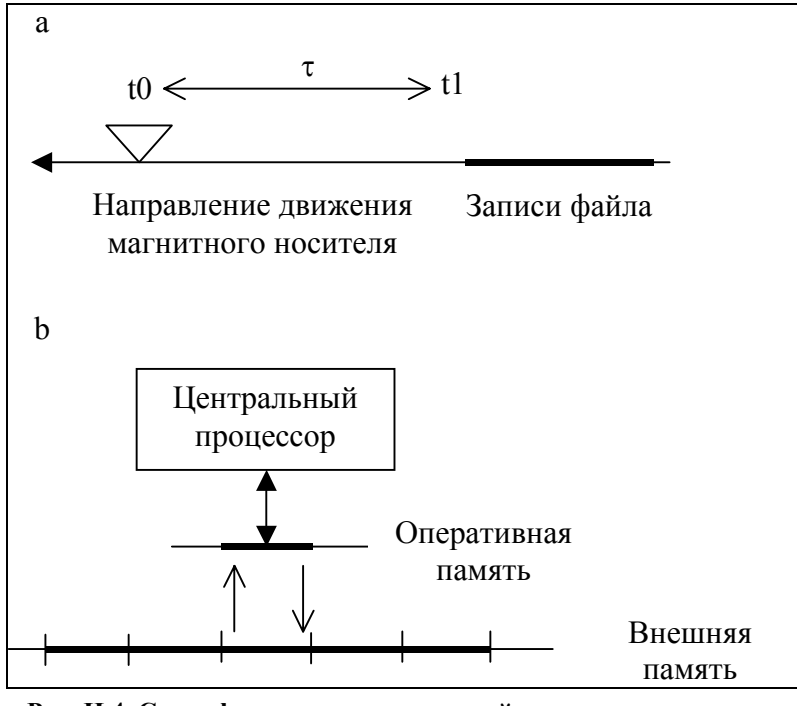


Рис. II.4. Специфика доступа к внешней памяти на магнитном носителе (a): t0 –начал доступа, t1 – начало чтения первого байта файла, τ - время подвода головки к началу файла. Технология блочной обработки большого массива данных (b).

Выгода обмена записями заключается в том, что основное время доступа занимает подвод читающих/записывающих головок к месту носителя данных, где локализуется первый байт записи (Рис. II.4). Следующие байты записи читаются/записываются гораздо быстрее.

Групповой обмен данными определяет специфический способ обработки больших массивов данных, хранящихся на внешней памяти. Хранящиеся на внешних устройствах данные разбиваются на записи, которые поочередно вызываются в оперативную память для обработки. После обработки текущей записи в оперативной памяти она может быть возвращена на место во внешнюю память.

Различают несколько типов файлов.

Последовательный файл состоит из множества линейно упорядоченных компонент - записей. Специфика работы с последовательным файлом – добавление в конец новой записи, и последовательное чтение записей файла, начиная с первой.

При выполнении операций чтения/записи используется последовательный доступ к записям файла. Для этого вводится **указатель** текущей записи в файле.

Основные операции над последовательным файлом таковы:

Открытие. По имени файла пользователь получает доступ к заказанному файлу.

Закрытие. Прекращает доступ к файлу.

Запись. Вставляет новую запись в конец файла, передает ей значение из переменной связи программы.

Чтение. Передает значение текущей записи файла в переменную связи программы. Передвигает указатель текущей записи вправо (делает текущей следующую запись).

Проверка конца файла. Логическая тестовая функция принимает значение "истина", когда указатель текущей позиции достиг конца файла. Поскольку длина файла не фиксирована, необходимо вычислять эту функцию перед каждой операцией чтения.

Обычно в виде последовательных файлов хранят таблицы. В этом случае в качестве записи выступает строка таблицы. Один из столбцов таблицы содержит ключи – неповторяющиеся единицы данных, каждая из которых однозначно идентифицирует одну строку таблицы. Файл с таблицей формируется путем последовательной записи в него строк (операция запись). Штатная работа с таблицей – поиск строки по заданному ключу. Такой поиск реализуется путем последовательного чтения записей (начиная с первой), пока не встретится запись с заданным ключом. Очевидно, что последовательный файл идеально приспособлен для такой работы.

Текстовый файл – частный случай последовательного файла, где в качестве записей используются символы.

Файл прямого доступа представляет собой множество записей**,** каждая из которых снабжена уникальным именем. Таким образом, возможен прямой доступ по имени к компонентам файла.

Примечание. В силу специфики внешних устройств прямой доступ не является равнодоступным или случайным (как для оперативной памяти).

Индексно-последовательный файл представляет собой комбинацию прямого и последовательного доступа: реализовав прямой доступ к какой-либо записи, следующие за ней записи можно обрабатывать в режиме последовательного доступа.

Двоичный файл – состоит из одной записи и содержит либо двоичный код программы, либо двоичный код данных. Обращение к такому файлу реализуется по имени.

Названные выше типы файлов реализуются в системах программирования совместно средствами аппаратуры и операционной системы. На аппаратном уровне файл представляет последовательность байт определенной длины. С каждым открытым файлом связан указатель на текущий байт, который надо считать или записать. Команды чтения – записи операционной системы считывают или записывают заданное число **n** байт, начиная с позиции определяемой указателем. Возможна предварительная установка указателя на заданный байт. После этого указатель перемещается на **n** байт. Таким образом, на машинном уровне имеется некоторое подобие индексно последовательного файла, записями которого служат байты.

Иерархическая структура памяти компьютера изображена на Рис. II.5.

Рис. II.5. Иерархическая структура памяти компьютера

Регистры выполнены на транзисторных двоичных запоминающих элементах, расположены непосредственно в центральном процессоре и, в силу этого, являются сверхбыстродействующими.

Кэш-память, назначение которой подробнее рассматривается ниже, также выполняется на транзисторных двоичных запоминающих элементах, располагается близко к процессору и является быстродействующей.

Плата за быстродействие регистров и кэш-памяти - высокая стоимость байта.

Оперативная память строится на основе конденсаторных динамических запоминающих элементов, связана с центральным процессором шиной передачи данных и имеет среднее быстродействие.

Магнитные диски (дискеты, винчестеры) хранят двоичные коды в виде намагниченных (единица) и не намагниченных (ноль) участков магнитного носителя, которые образуются когда, участки магнитного диска проходят под записывающей головкой.

Считывание основано на возникновении ЭДС, когда намагниченный участок проходит под считывающей головкой. Характеризуются малым быстродействием, но большой емкостью (десятки гигабайт). Записанные данные могут искажаться с течением времени под воздействием магнитных полей.

Двоичные коды на поверхности лазерного диска (компакт-диск, **CD** –диск, **DVD** –диск) представляются в виде углублений (единицы) и площадок (нули), выжигаемых лучом лазера во время записи. При считывании "ловится" отраженный от углублений и площадок лазерный луч, который преобразуется в электрические импульсы. Преимущество лазерных дисков – реализация сменных носителей данных без опасности искажения информации с течением времени.

Очевидно, что при продвижении по иерархической структуре памяти вниз (Рис. II.5) увеличивается объем памяти и время доступа, но уменьшается стоимость одного байта.

В эту иерархию памяти компьютера не входит постоянное запоминающее устройство **ПЗУ** (только для записи), хранящее коды микропрограммного управления и программу первоначальной загрузки операционной системы.

2.1. Контрольные вопросы

1. Зачем нужны внешние запоминающие устройства?
2. Специфика доступа в запоминающих устройствах на движущихся магнитных носителях.
3. Файлы и способы доступа к файлам.
4. Целесообразность иерархии запоминающих устройств.

3. Операционные системы и системы программирования

Несмотря на большую сложность аппаратуры компьютера, он неспособен обеспечить условия для эффективной работы пользователя - программиста.

Первая проблема - технология программирования в двоичных кодах, язык которых только и понимает аппаратура вычислительной машины, чрезвычайно трудоемка и потенциально содержит большие возможности для совершения ошибок в записи кода.

Вторая проблема - технология подготовки программы и пропуска ее через компьютер состоит из многих этапов, которые могут комбинироваться в различных вариантах.

Решение первой проблемы состоит в разработке **средств автоматизации программирования**, ориентированных на эффективность и надежность программирования. Вторая проблема решается посредством разработки **операционных систем**, управляющих вычислительным процессом.

Средства автоматизации программирования состоят из нескольких компонент.

- **Языки программирования** высокого уровня, позволяют записывать алгоритмы в достаточно понятном виде не только для системного программиста, но и для проблемного специалиста (инженера, физика и т.д.). Кроме того, эти языки освобождают программиста от ручного распределения памяти за счет механизмов ее автоматического распределения, а также содержат средства конструирования сложной программы из более простых компонент (подпрограммы, сопрограммы, модули и т.д.).
- **Трансляторы** (компиляторы, интерпретаторы) представляют собой программы – переводчики с языка высокого уровня на язык двоичных кодов. Кроме того, трансляторы производят синтаксический контроль программы и реализуют сборку сложной программы из более простых компонент.
- **Библиотеки стандартных программ** содержат специальным образом оформленные программные реализации общезначимых алгоритмов, которыми можно пользоваться, не вдаваясь в детали их функционирования. Библиотеки специфичны для каждого конкретного языка программирования и постоянно расширяются фирмами, выпускающими на рынок трансляторы.

Операционные системы расширяют сервисные и функциональные возможности аппаратуры компьютера за счет разработки системных программ.

Таким образом, пользователь управляет действиями аппаратуры компьютера не напрямую, а через достаточно "интеллектуального" посредника.

Примечание. Подобное разделение функций управления между аппаратурой и компьютером определяется теми соображениями, что реализовать приемлемые для человека алгоритмы управления вычислительной системой только аппаратными средствами либо практически невозможно, либо неоправданно дорого. (Вспомните о двух способах реализации алгоритма главного цикла вычислительной машины: аппаратное и микропрограммное решение). Однако удешевление и миниатюризация микроэлектронных технологий имеет своим следствием тенденцию постоянного расширения объема аппаратных реализаций функций управления.

Операционная система представляет собой большую и сложную системную программу, состоящую из двух частей. Относительно небольшая, **резидентная часть (ядро) операционной системы** постоянно размещена в оперативной памяти компьютера и реализует функции, время выполнения которых должно быть минимальным (распределение времени центрального процессора, анализ прерываний и т.д.). Гораздо большая по объему **нерезидентная часть операционной системы** расположена на внешних устройствах.

Выделяются три основных функции операционной системы:

1. **Распределение ресурсов компьютера** между множеством выполняемых (программ). К числу таких ресурсов относятся: оперативная память, время центрального процессора, внешние устройства (диски, принтеры и т.д.), системные программы общего назначения (программы ввода, программы вывода и т.д.). Эти функции выполняет управляющая программа планировщик и управляющая программа супервизор
2. **Управление данными** (каталогизация, хранение, поиск, защита данных). Эти функции реализует **файловая подсистема**.
3. **Реализация "дружественного" интерфейса** (связи) человека с аппаратурой компьютера. Долгое время на вопросы "дружественного" интерфейса не обращалось должного внимания. В силу этого с операционными системами могли общаться только системные программисты.

Прорыв в этом вопросе произошел после формирования концепции процессора управления файлами (Norton Commander) и тем более после создания графического интерфейса (Macintosh,Microsoft). Таким образом, приемлемые для пользователя методы программирования и алгоритмы управления технологическим процессом пропуска программ через вычислительную машину реализуются частично аппаратурой, частично системными программами.

Принято говорить, что пользователь имеет дело с **виртуальной машиной** (совокупность аппаратных и программных средств), обеспечивающих достаточно "комфортную" работу человека с компьютером (Рис. II.6).

Машина называется виртуальной (изменчивой) в силу того, что ее сервисные и функциональные возможности зависят от того системного программного обеспечения, которое установлено на ней в настоящее время.

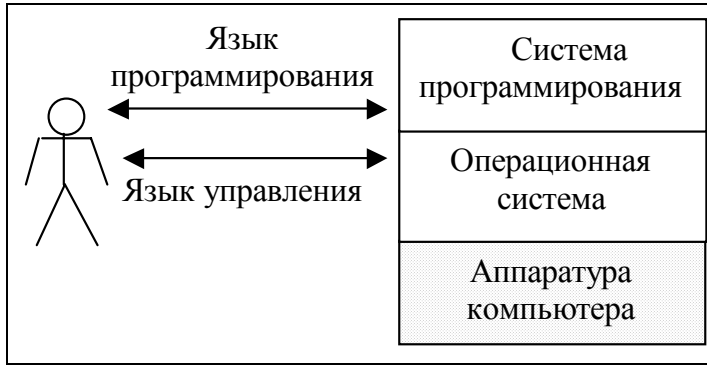


Рис. II.6. Виртуальная вычислительная машина

Общение с виртуальной вычислительной машиной реализуется посредством двух языков. Язык программирования (Паскаль, С++ и т.д.) является средством записи алгоритмов в виде программы. Язык управления заданиями (ЯУЗ) является средством формирования сценария пропуска программ через вычислительную машину.

Приведем основную терминологию, связанную с операционными системами.

Работа - действия по преобразованию данных, приводящие к решению прикладной задачи. **Процесс (задача)** – единица работы, существующая в динамике. Процесс функционирует в соответствии с некоторой программой, являющейся его статическим описанием. Говоря другими словами, процесс возникает, когда программа загружается в оперативную память и начинает претендовать на ресурс времени центрального процессора. Процесс прекращается, когда программа освобождает оперативную память и центральный процессор от своего присутствия.

Существенно, что несколько процессов, порожденных одной программой, могут существовать независимо, выполняясь последовательно или параллельно. В последнем случае программа, порождающая процесс, называется реентерабельной.

Процесс является единицей работы, претендующей на **ресурсы**.

Первые вычислительные машины могли выполнять лишь одну программу, загруженную в оперативную память – реализовывать одну задачу, загруженную в память, с начала до конца. В этих компьютерах применялись примитивные управляющие программы – диспетчеры.

А сами машины, по сути дела, представляли собой примитивные, огромные и дорогостоящие персональные компьютеры коллективного пользования: каждый программист получал весьма небольшой отрезок времени, в течение которого он мог единолично отлаживать или выполнять свою программу.

Останавливая программу для раздумий, программист останавливал центральный процессор. По статистике простои дорогостоящего центрального процессора достигали 70 -80%.

Стремление эффективно использовать центральный процессор привела к созданию **пакетного режима** обработки программ. Все, что надо было сделать с программой, какую информацию выдать и когда ее выдать, программировал пользователь на языке управления заданиями (ЯУЗ) операционной системы. Предложения ЯУЗ, программы и исходные данные переносились на перфокарты, колоды которых (задания) сдавались диспетчеру, после чего программист терял контроль над своей программой. Диспетчер формировал пакет заданий многих пользователей, а операционная система, исходя из своих приоритетов, создавала из заданий пакета задачи, выполняла их и выдавала результаты.

Для технической поддержки пакетной технологии обработки программ был изобретен многозадачный (мультипрограммный) режим работы вычислительной системы. Он допускал одновременное существование нескольких задач. И если по какой-либо причине работающая задача временно не могла продолжаться, запускалась задача готовая к исполнению. Тем самым простои центрального процессора практически исключались.

Таким образом, основное требование пакетного режима – процесс обработки программы со всеми ее деталями должен быть предусмотрен с начала и до конца. Неконструктивность такого подхода сказывалась особенно во время отладки. Но также нередки задачи, ход решения которых либо сложно, либо невозможно предсказать заранее. В силу сложности такого предсказания отладка и решение задачи проводились небольшими порциями, что вызывало неоправданно большие затраты времени на получение результата. Очевидно, что пакетная технология обработки программ невозможна без мощной управляющей программ - операционной системы.

Альтернативой пакетной технологии обработки программ явился **диалоговый (интерактивный) режим** функционирования операционной системы, который не требует знания всех особенностей вычислительного процесса с начала и до конца. Вычислительный процесс прерывается в точках возможного разветвления и запрашивает у пользователя указание для дальнейшего продолжения.

Для технической поддержки диалогового режима был изобретен режим разделения времени центрального процессора. В этом случае многие пользователи одновременно работают за своими абонентскими пультами (терминалами), возможно удаленными от вычислительной машины на значительное расстояние.

Время центрального процессора делится между абонентскими пультами путем выделения каждому небольшого кванта времени (как делит ресурс своего времени гроссмейстер между любителями при сеансе одновременной игры в шахматы). При этом создается эффект квазиодновременной работы многих пользователей с одной вычислительной машиной.

3.1. Контрольные вопросы

1. Средства автоматизации программирования и средства автоматизации управления вычислительным процессом
2. Основные функции операционной системы
3. Понятие виртуальной вычислительной машины.
4. Программа и процесс (задача)
5. Пакетный и диалоговый режим работы операционной системы.

4. Совершенствование адресации оперативной памяти

4.1. Действия и команды

По своему существу алгоритм представляет собой совокупность действий, исполнение которых преобразует исходные данные в результирующие данные. Действия служат дополнением к данным: данные представляют пассивную компоненту, а действия представляют активную компоненту, которая позволяет создавать, уничтожать и преобразовывать данные.

Имеется два типа действий: действия над данными, определяемыми программистом и действия над данными, определяемыми системой. К первому типу относятся действия обработки данных – сложение, вычитание, пересылка, проверка на равенство и т.д. Ко второму типу относятся действия, которые обычно считаются частью управляющей структуры языка программирования: переходы условные и безусловные, вызовы подпрограмм, присваивание имен структурам данных и т.д.

Естественно рассматривать **действие обработки данных** в языке программирования как математическую функцию: для заданного множества операндов она выдает вполне определенный и единственный результат. Общее число операндов + результат будем называть **числом параметров действия**. Например, арифметические действия являются трехпараметрическим (два операнда и результат), действие пересылки – двухпараметрические (операнд источник и результат приемник), действие передачи управления – однопараметрические (результат определяет адрес перехода), операция останова – нульпараметрическое действие.

Программа состоит из совокупности команд некоторой вычислительной машины и являет собой один из возможных способов записи алгоритма. **Команда** вычислительной машины связывает данные и машинные операции таким образом, что каждая операция применяется к соответствующим данным в соответствующее время. Говоря другими словами, команда – предписание (императив) вычислительной машине выполнить определенную машинную операцию с определенными данными.

Таким образом, мы различаем понятия: действие алгоритма (действие) и машинная операция (команда). Также очевидно, что при программировании (записи алгоритма на алгоритмическом языке) возникает проблема проекции действий алгоритма на команды вычислительной машины.

В общем случае команда состоит из операционной и адресной частей, каждая из которых, в свою очередь, может состоять из нескольких полей. **Операционная часть команды** содержит код операции (КОП), который задает вид машинной операции.

Адресная часть команды содержит информацию об адресах исходных данных и адресах результирующих данных, а в некоторых случаях также информацию об адресе следующей команды (команды передачи управления). Команда, имеющая в адресной части **n** полей, называется **n** – адресной командой (Рис. II.7).

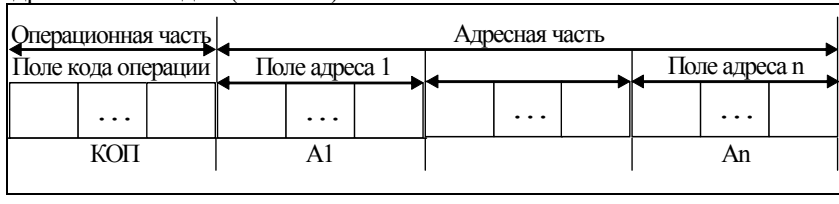


Рис. II.7. Формат n адресной команды

Состав, назначение и расположение полей в команде в совокупности с разметкой номеров разрядов (бит), определяющих границы отдельных полей, называется **форматом команды**. Выбор формата команды является одной из важнейших проблем проектирования компьютера. Команда, как правило, согласуется с длиной обрабатываемых данных (ячеек хранения данных), что позволяет эффективно использовать одни и те же память, шины передачи данных и прочие аппаратные средства, как для данных, так и для команд.

Адресная часть команды, в общем случае, содержит не адреса данных, но информацию, на основании которой эти адреса могут быть вычислены в процессе выполнения команды. Способ задания и использования такой информации обусловливает различные **способы адресации данных.**

Различные способы адресации имеют следующее назначение:

- указывать полный адрес памяти наименьшим количеством разрядов в поле адреса, следовательно, делать команду короче;
- разрешать команде осуществлять доступ к ячейке памяти, адрес которой вычисляется во время исполнения программы, обеспечивая при этом эффективный доступ к массивам и спискам;
- вычислять адреса относительно местоположения команды, чтобы обеспечить загрузку программы в любую область памяти без изменения программы.

4.2. Трехадресные команды

Первые вычислительные машины не имели байтовой структуры оперативной памяти, т.е. ячейки вычислительной машины состояли из фиксированного числа разрядов. В каждой такой ячейке могла храниться либо команда, либо единица данных. В силу этого, все команды вычислительной машины были одинаковой длины.

Первые вычислительные машины использовали **трехадресные команды** следующего формата:

КОП A1,A2,A3; **(II.1)**

где КОП – поле кода машинной операции; A1, A2, A3 – поля адресов. Причем, в поле адреса записывается физический адрес оперативной памяти. Способ адресации, когда в поле адреса записывается физический адрес памяти, называется **прямой адресацией оперативной памяти.**

Для примера, любая команда вычислительной машины имела длину в 48 бит и 12-ти разрядное поле адреса (Рис. II.8):

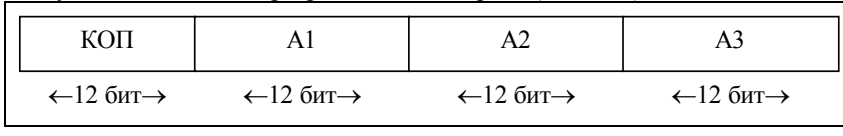


Рис. II.8. Формат трехадресной команды

Проекция трехпараметрических действий (арифметических и логических) на трехадресные команды происходит естественным образом: первые два поля адреса содержат адреса исходных данных, третье поле адреса – адрес результата.

Проекция двухпараметрических действий (пересылка) на трехадресную систему команд происходит следующим образом: первое поле адреса – источник данных, третье поле адреса – приемник данных, второе поле адреса не используется.

Проекция однопараметрических действий (переход) на трехадресную систему команд происходит следующим образом: второе поле адреса - адрес перехода, первое и третье поля адреса не используются.

Трехадресная система команд удобна для "ручного" программирования, т.к. позволяет в привычной для человека форме записывать предписание на выполнение трехпараметрического действия в виде одной команды. Например: взять исходные данные из ячеек с адресами A1 и A2, выполнить сложение, отправить результат в ячейку с адресом A3 (Рис. II.9).

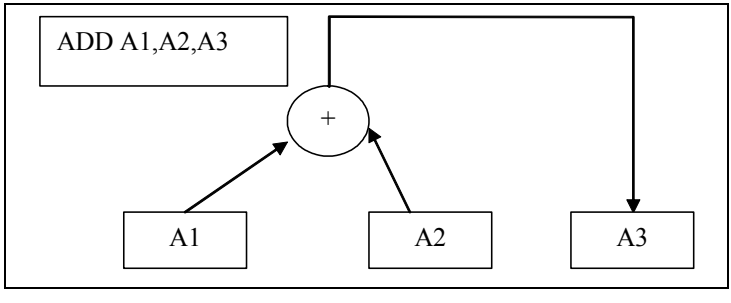


Рис. II.9. Схема выполнения трехадресной команды обработки данных

В рассматриваемом случае трехадресные команды имеют фиксированную и одинаковую длину, и при хранении в оперативной памяти те из них, которые реализуют двухпараметрические и однопараметрические действия, занимают значительное количество бит для представления "пустых" операндов. А оперативная память первых вычислительных машин имела небольшой объем и большую стоимость изготовления. Отсюда понятно стремление оптимизировать форматы команд вычислительной машины.

Существует еще один серьезный фактор, требующий оптимизации форматов машинных команд. Вспомним соотношение числа разрядов адреса **k** и размера адресного пространства оперативной памяти **q**:

$$q = 2^k. \qquad \textbf{(II.2)}$$

Желание расширить объем оперативной памяти вступает в противоречие с этим соотношением. Так, для 12–ти разрядного адреса вычислительной машины типа М-20 размер адресного пространства составлял 4096 ячеек, не больше и не меньше. Если же увеличить число разрядов адреса, то увеличивается длина команды и она "не уберется" в 48-ми разрядную ячейку оперативной памяти.

4.3. Число адресов команды и размер адресного пространства оперативной памяти

Здесь мы рассмотрим тенденцию уменьшения числа адресов команды, что позволяет избежать "пустых адресов" и, в некоторой степени, решает проблему расширения размеров адресного пространства оперативной памяти.

Возьмем для начала трехадресную команду, в полях адреса которой записывается физический адрес оперативной памяти. Пусть, например, имеем 48-ми битовую трехадресную команду (Рис. II.10), поле адреса которой имеет размер в 12 бит. В этом случае имеем размер адресуемого пространства оперативной памяти: $Q = 2^{12} = 1024 = 4$ Кб.

Уменьшив число адресов в этой команде до двух, получаем **двухадресную команду** Адреса такой команды принято называть **Аист** – адрес источника данных и **Апр** – адрес приемника данных. Получаем расширение адресного пространства до $\mathbf{Q = 2^{18} = 256}$ **кб**. Двухадресная команда пересылки не содержит "пустых адресов" **(MOV Аист Апр**). Двухадресная команда передачи управления содержит один "пустой адрес" (вместо двух в трехадресной команде).

А как обеспечить исполнение трехпараметрического действия обработки данных, например, действия сложения? Определим формат двухадресной команды сложения:

ADD Аист,Апр; **(II.3)**

и установим следующую схему ее выполнения (Рис. II.11).

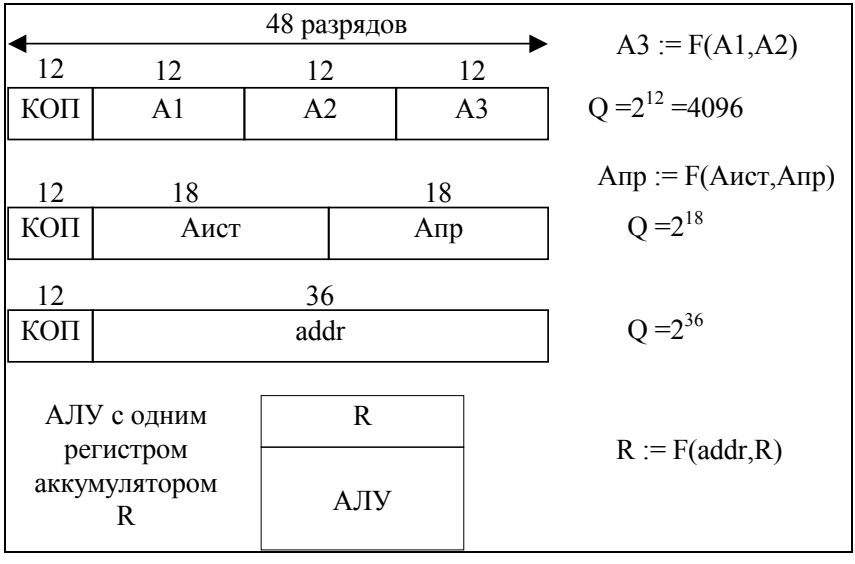


Рис. II.10. Расширение адресного пространства за счет уменьшения числа адресов в команде

Тогда, для реализации трехпараметрического действия необходимо выполнить последовательность из трех двухадресных команд:

MOV A1,temp; ADD A2,temp; MOV temp,A3. **(II.4)**

Здесь используется вспомогательная ячейка оперативной памяти с адресом **temp**, а также команда **MOV** - пересылки между ячейками оперативной памяти (из источника в приемник).

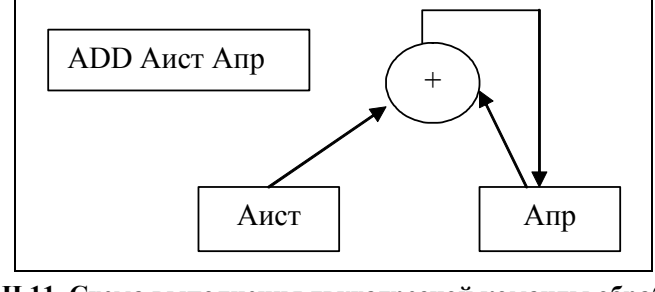


Рис. II.11. Схема выполнения двухадресной команды обработки данных

Такая схема исполнения команды требует более сложной организации вычислительного процесса по сравнению с трехадресными командами. При "ручном" программировании это вызывает определенные сложности.

Но к моменту появления двухадресных команд "ручное" программирование осуществлялось лишь небольшим числом системных программистов. Основная масса программировала на языках высокого уровня, где машинных команд не было вообще, и сложности вычисления арифметических выражений "легли на плечи" трансляторов.

Оказывается, можно использовать и **одноадресные команды**. В этом случае однопараметрическое действие передачи управления реализуется одноадресной командой: (например, **JMP addr**), которая не содержит "пустых адресов". Но для обеспечения возможности исполнения двух и трех параметрических действий обработки данных необходимо изменить архитектуру арифметического устройства (Рис. II.10). Вместо трех регистров (два входных регистра и один регистр результата) используется единственный накопительный регистр **R** или **регистр – аккумулятор**.

Команда использует два операнда: первый хранится в оперативной памяти, а второй являет собой результат выполнения предыдущей операции и хранится на аккумуляторном регистре. Результат исполнения команды остается на аккумуляторном регистре.

В качестве платы за одноадресность, вводятся две специфические одноадресные команды обмена данными между оперативной памятью и регистром-аккумулятором:

LOAD addr - пересылка на аккумулятор **R** содержимого ячейки оперативной памяти с адресом **addr**;

STORY addr – пересылка в ячейку оперативной памяти с адресом **addr** содержимого аккумулятора **R**.

Выполнение двухпараметрического действия пересылки реализуется двумя одноадресными командами:

LOAD Аист; STORY Апр. **(II.5)**

Выполнение трехпараметрического действия сложения осуществляется следующим образом:

LOAD A1; ADD A2; STORY A3. **(II.6)**

В этом случае одноадресная команда сложения имеет формат: **ADD addr** и ее исполнение реализуется по схеме (Рис. II.12):

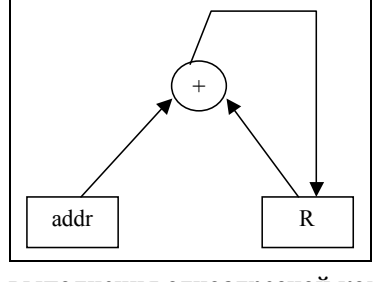


Рис. II.12. Схема выполнения одноадресной команды обработки данных

Рассмотрим пример вычисления арифметического выражения:

z := a+b+c+d. **(II.7)**

С использованием трехадресных команд программа его вычисления выглядит следующим образом: и требует 9 обращений к оперативной памяти (три обращения на команду):

```
ADD a b z;
ADD z c z;
ADD z d z.
```

Здесь необходимо 9 обращений к оперативной памяти (три обращения на команду).

С использованием двухадресных команд программа вычисления выглядит следующим образом:

```
ADD a z;
ADD b z;
ADD c z;
ADD d z.
```

Здесь необходимо 8 обращений к оперативной памяти (два обращения на команду).

С использованием одноадресных команд программа записывается в следующем виде:

```
LOAD a;
ADD b;
ADD c;
ADD d;
STORY z.
```

При этом, команды **LOAD** и **STORY** используют по одному обращению к оперативной памяти. Команда **ADD** использует одно обращение к оперативной памяти и два обращения к регистру – аккумулятору.

Но в качестве регистра аккумулятора используется сверхбыстродействующий регистр, время обращения к которому значительно меньше, чем время обращения к оперативной памяти.

Схема вычислений, при которой промежуточный результат остается на сверхбыстродействующем регистре, существенным образом повышает скорость вычисления арифметического выражения. (Всего требуется 5 обращений к оперативной памяти.)

Таким образом, использование регистра аккумулятора не только позволяет расширить адресное пространство и устраняет проблему "пустых адресов", но также позволяет при соответствующей организации вычислительного процесса увеличить его скорость за счет отсутствия необходимости отсылать промежуточные результаты в "медленную" оперативную память.

4.4. Команды и байтовая структура оперативной памяти

Байтовая структура оперативной памяти позволяет использовать команды переменной длины. В этом случае команда состоит из одного или более байт, причем первый байт представляет код операции. В архитектурах современных компьютеров определяются безадресные, одноадресные и двухадресные команды (Рис. II.13).

Примеры команд:

- безадресные: **HLT** (останов), **RET** (возврат из прерывания);
- одноадресные: **DEC addr** (уменьшение значения ячейки памяти на единицу), **JMP addr** (безусловный переход на команду с заданным адресом);
- **MOV Аист Апр** (пересылка значения из источника в приемник), **ADD Аист Апр** (сложение значения источника с значением приемника – результат в приемник).

Поле адреса в такой команде может состоять из произвольного числа байт, что позволяет в этом поле хранить адрес с нужным числом разрядов. Так, например, четыре байта адреса позволяют адресовать пространство оперативной памяти в 2^{32} байт (4 терабайта). Теоретически это решает проблему адресации оперативной памяти большого объема.

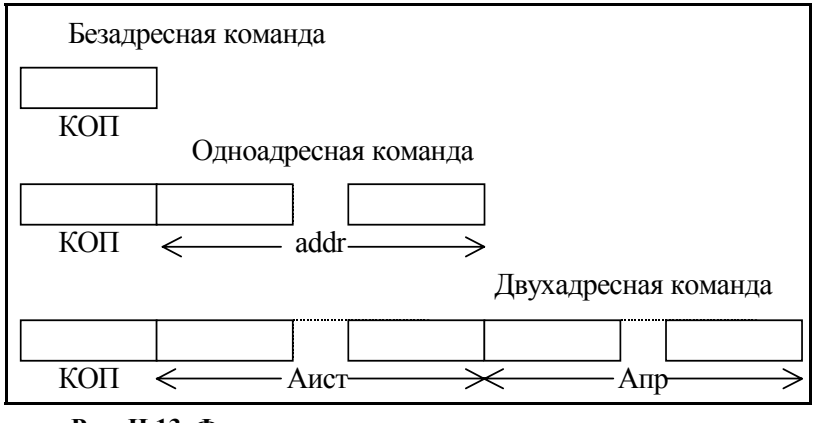


Рис. II.13. Форматы команд современных компьютеров

Но есть практическая сторона вопроса. Вспомним, что время выполнения программы в значительной степени определяется скоростью обмена оперативной памяти с центральным процессором. Длина двухадресной команды составляет: (1+4+4) = 9 байт. Если обмен оперативной памяти с центральным процессором осуществляется побайтно, то для чтения команды на регистр команд требуется девять обращений к оперативной памяти.

Поэтому, длинные команды существенным образом замедляют скорость обработки данных программой.

Примечание. В современных архитектурах количество байт, за одно обращение извлекаемых из оперативной памяти (возвращаемых в оперативную память), определяется шириной шины данных. Так, шестнадцатиразрядная шина данных обеспечивает за одно обращение передачу двух байт и т.д. Это частично решает проблему "длинной "команды, но не снимает ее совсем.

4.5. Использование стековой памяти

При автоматическом выполнении программы, записанной на языке высокого уровня, возникают две проблемы, которые для своего решения требуют использования стековой памяти.

Стековая память представляет собой динамический массив, дисциплина обслуживания для которого формулируется следующим образом: ***последний вошел – первый вышел***.

Носитель текста, на основании которого строится стековая память, так же как и для случая оперативной памяти, представляет собой последовательность строк. Но число строк в этой последовательности изменяется при выполнении операций включения/ исключения.

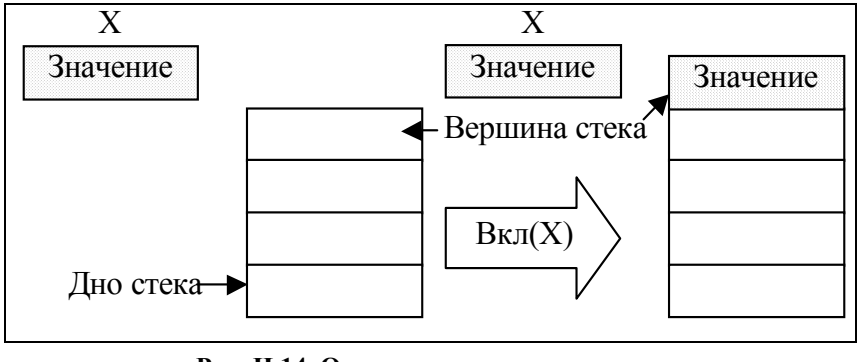


Рис. II.14. Операция включения в стек

При выполнении операций со стеком используется переменная связи **X**. **Операция включения** добавляет строку в вершину стека и записывает в нее значение переменной связи **X** (Рис. II.14). **Операция исключения** присваивает переменной связи X значение строки, находящейся в вершине стека, и исключает эту строку из стека (Рис. II.15).

Необходима также логическая **тестовая функция "Пустой стек"**.

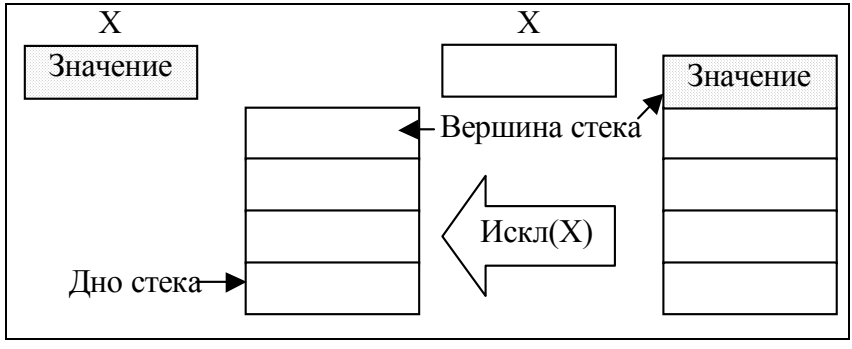


Рис. II.15. Операция исключения из стека

4.5.1. Автоматическое вычисление арифметического выражения.

В системе команд компьютера нет команды вычисления произвольного арифметического выражения. Его вычисление необходимо алгоритмизировать, используя лишь четыре арифметических операции, которые реализуются командами обработки данных компьютера. Т.е. необходим алгоритм, который, получая на вход арифметическое выражение любой сложности, обеспечивает последовательность выполнения арифметических операций, обеспечивающих вычисление его значения.

Алгоритм, очевидно, должен учитывать как приоритеты выполнения арифметических операций, так и изменение этих приоритетов с учетом наличия скобочной структуры произвольной глубины.

Перед вычислением арифметическое выражение преобразуется из инфиксной формы в постфиксную форму. **Инфиксная форма** (привычная для человека) предполагает запись знака бинарной операции между операндами этой операции. Например: (3+7)*5+(4+6)*3 = 80. Вычисление выражения ведется в соответствии с приоритетами, порядок применения которых изменяется скобками.

Постфиксная форма арифметического выражения предполагает запись знака операции после операндов этой операции. Ниже приводится неформальный алгоритм перевода выражения из инфиксной формы в постфиксную форму, который может исполняться только человеком:

- операнды выражения переписываются в том же порядке, в каком они присутствуют в выражении, скобки игнорируются;
- расставляются знаки операций после тех операндов, к которым они могут быть применены.

Для нашего примера получаем постфиксную форму арифметического выражения:

3_7_+_5_*_4_6_+_3_*_ +

Примечание 1. Существует формальный алгоритм перевода из инфиксной формы в постфиксную форму, использующий стековую память.

Примечание 2. Постфиксную форму арифметического выражения называют также "бесскобочной" или "польской". Польская – по имени польского математика Лукашевича, впервые предложившего эту форму записи.

Постфиксная форма записи арифметического выражения не использует скобки и не требует установки приоритетов выполнения операций. Поэтому алгоритм вычисления выражения очень прост и сводится к последовательному просмотру постфиксной записи и к выполнению операций по мере их обнаружения.

Алгоритм I.1. Вычисление арифметического выражения

1. Последовательно просматривать постфиксную запись;
2. **ЕСЛИ** текущий элемент – операнд
3. **ТО** ВКЛ(операнд)
4. **ИНАЧЕ** ИСКЛ(X); ИСКЛ(Y); X:=X<знак>Y;
5. ВКЛ(X);
6. После окончания просмотра постфиксной записи на дне стека остается результат вычисления арифметического выражения.

Для нашего примера состояния стека изменяются, как показано на (Рис. II.16).

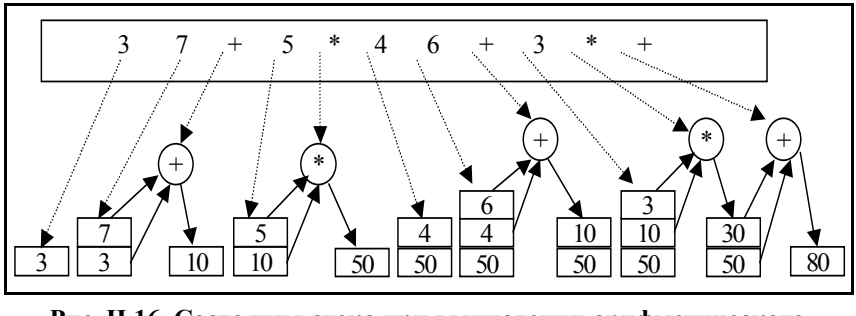


Рис. II.16. Состояния стека при вычислении арифметического выражения

4.5.2. Управление выполнением программы

Программа, записанная на языке высокого уровня, состоит из головной программы и подпрограмм и имеет структуру дерева (Рис. II.17). Порядок выполнения программы определяется обходом дерева (пунктирная линия).

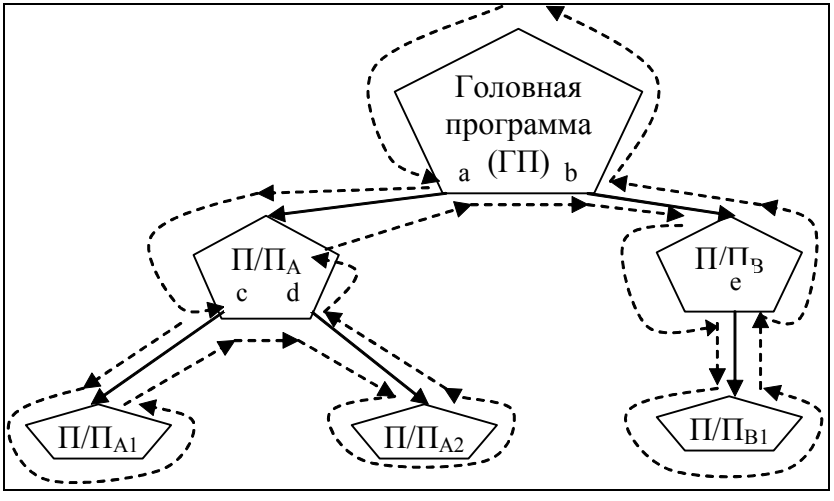


Рис. II.17. Древовидная структура программы

При выполнении обхода дерева необходимо в каждый момент помнить путь, ведущий из корня дерева к текущей вершине. Запоминание такого пути позволяет реализовать возврат из подпрограммы, закончившей свою работу, к программе вызывающей эту подпрограмму. Для такого запоминания идеальным образом подходит стековая память (**Ошибка! Источник ссылки не найден.**).

Таблица II.7. Порядок выполнения программы

Выполнение ГП до адреса "a";	[стек]=пусто
Обращение к П/П$_A$: **ВКЛ**(СЧК+1); безусловный переход на начало П/П$_A$	[СЧК]=a [стек]=a+1
Выполнение П/П$_A$ до адреса "c"	[стек]=a+1
Обращение к П/П$_{A1}$: **ВКЛ**(СЧК+1); безусловный переход на начало П/П$_{A1}$	СЧК=c [стек]=a+1,c+1
Выполнение П/П$_{A1}$ до **RETURN;** Возврат на продолжение П/П$_A$: ИСКЛ(адрес_возврата); Безусловный переход на адрес возврата т.е. на c+1	[адрес_возврата]=c+1 [стек]=a+1;
Выполнение П/П$_A$ до адреса d	[стек]=a+1;
Обращение к П/П$_{A2}$: **ВКЛ**(СЧК+1); безусловный переход на начало П/П$_{A2}$	[СЧК]=d [стек]=a+1;d+1

Таблица II.7. Продолжение

Выполнение $П/П_{A2}$ до **RETURN;** Возврат на продолжение $П/П_A$: ИСКЛ(адрес_возврата); Безусловный переход на адрес возврата т.е. на d+1	[адрес_возврата]=d+1 [стек]=a+1;
Выполнение $П/П_A$ до **RETURN;** Возврат на продолжение ГП: ИСКЛ(адрес_возврата); Безусловный переход на адрес возврата т.е. на a+1	[стек]=пусто
Выполнение ГП до адреса "в";	[стек]=пусто
Обращение к $П/П_B$: **ВКЛ**(СЧК+1); безусловный переход на начало $П/П_B$	[СЧК]=b [стек]=b+1
Выполнение $П/П_B$ до **RETURN;** Возврат на продолжение ГП: ИСКЛ(адрес_возврата); Безусловный переход на адрес возврата т.е. на b+1	[адрес_возврата]=b [стек]=пусто
Завершение ГП	

Условные обозначения: **[x]** – содержимое строки **x**; **СЧК** – счетчик команд.

4.6. Однокомпонентные способы адресации

4.6.1. Прямая и непосредственная адресация

Как уже говорилось неоднократно, самым простым и очевидным способом указания местоположения операнда является расположение в поле адреса команды полного (физического) адреса операнда в оперативной памяти, т.е. использование прямой адресации оперативной памяти (Рис. II.18a). Синоним прямой адресации – **абсолютная адресация.** До сих пор при рассмотрении команд вычислительной машины мы использовали прямую адресацию.

Непосредственная адресация позволяет задавать константу как составную часть команды. В адресном поле команды записывается константа, которая выступает в качестве операнда команды (Рис. II.18b). Например, можно записать число π или какую либо физическую константу. Большинство непосредственных операндов представляет собой небольшие величины со знаком, для размещения которых требуется один байт памяти (редко два байта). Операнд передается в центральный процессор во время чтения команды из оперативной памяти. Непосредственная адресация не является необходимой, но ее использование уменьшает объем программы и сокращает время ее выполнения.

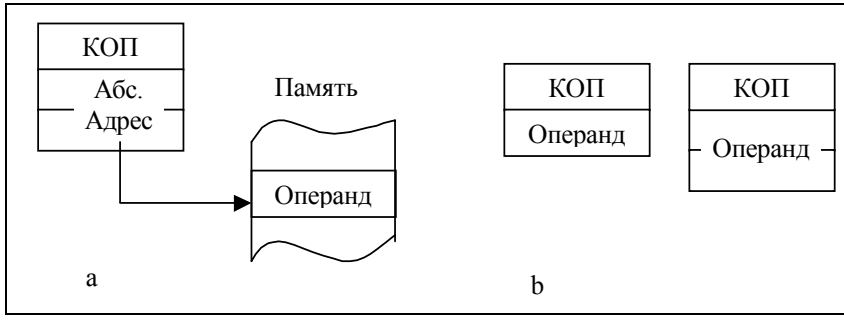


Рис. II.18. Прямая адресация оперативной памяти (a) и непосредственная адресация операнда (b)

4.6.2. Регистры общего назначения и регистровая адресация

Еще раз подчеркнем, что до сих пор мы имели дело только с прямой адресацией оперативной памяти, т.е. все адресные поля команд содержали физические адреса оперативной памяти. Но также использовались два вида памяти, которые адресовались по умолчанию: регистр-аккумулятор **R** и стековая память.

Кроме всего прочего, регистр-аккумулятор, выполненный как сверхбыстродействующая ячейка памяти, при определенной технологии вычислений обеспечивал большее быстродействие за счет возможности не пересылать промежуточные результаты в оперативную память, а хранить их на регистре-аккумуляторе. В дальнейшем оказалось целесообразным ввести в состав центрального процессора массив сверхбыстродействующих ячеек (Рис. II.19a), которые получили название **регистров общего назначения** (**РОН**).

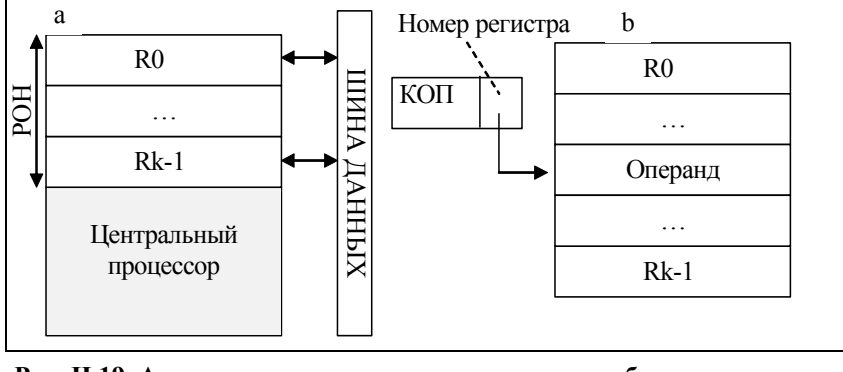


Рис. II.19. Архитектура процессора с регистрами общего назначения (a) и регистровая адресация (b)

На регистрах общего назначения можно хранить различные промежуточные результаты, текущую исполняемую команду, адреса стека и массивов данных и т.д.

По сути дела, в архитектуре компьютера, кроме оперативной памяти, появляется еще один класс памяти – сверхбыстродействующая регистровая. Такая память имеет небольшой объем (8 или 16 регистров) и имеет свою систему адресации. Адрес регистра общего назначения принято называть номером регистра.

Регистровая адресация является ничем иным как прямой адресацией **РОН (**Рис. II.19b**)**. В силу того, что для хранения номера регистра требуется незначительное число разрядов, поле адреса команды и вся команда в целом имеет небольшую длину. Как говорилось ранее, команда может включать в себя несколько адресных полей. Вполне допустимо, что каждое поле может иметь свою специфическую адресацию. Например, первое поле – регистровая адресация, второе поле – прямая адресация.

Теперь имеется два класса памяти (оперативная и регистровая) и возникает немаловажный вопрос распознавания способа адресации, который используется в каждом поле адреса. Здесь используются два основных подхода.

Первый подход. Способ адресации задается кодом операции команды. Например, команда пересылки байта из регистра РОН в оперативную память записывается на языке ассемблера в виде:

MOV mem_byte, R1; (II.8)

и имеет код операции $(88)_{16}$. Тогда как команда пересылки байта из оперативной памяти на регистр **РОН** записывается на языке ассемблера в виде:

MOV BL, mem_byte; (II.9)

и имеет код операции $(8A)_{16}$.

Второй подход. Признак регистровой адресации можно также указывать в поле адреса, которое в этом случае состоит из двух частей:

<префикс способа адресации> <номер регистра>.

Способы адресации современного компьютера (особенно PC) чрезвычайно разнообразны. Подробное их перечисление выходит за рамки настоящего издания. Скажем только, что в системе команд современного компьютера используются обе возможности: что-то определяется кодом операции, что-то префиксом способа адресации в поле адреса.

4.6.3. Косвенная регистровая адресация

Косвенная регистровая адресация появилась впервые в шестнадцатиразрядных машинах как средство, позволяющее небольшим числом разрядов поля адреса адресовать большое количество регистров оперативной памяти.

В случае косвенной регистровой адресации, поле адреса команды содержит адрес регистра общего назначения, который является исполнительным адресом для доступа к ячейке оперативной памяти (Рис. II.20).

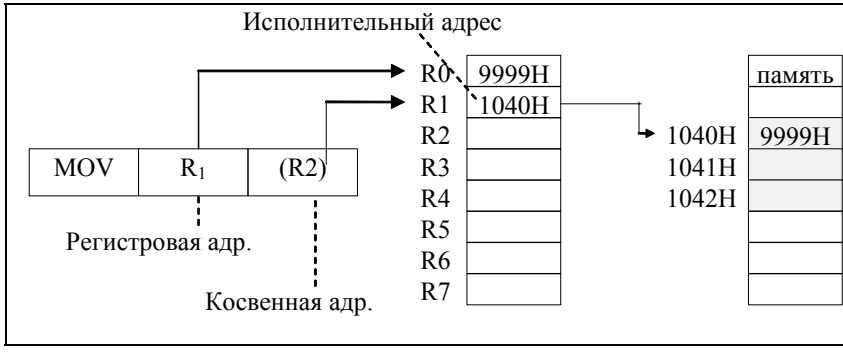


Рис. II.20. Косвенная регистровая адресация

Косвенная регистровая адресация использует понятие **исполнительного адреса** (**Аисп**), которое присутствует во многих сложных способах адресации. При этом адрес доступа к памяти (**Аисп**) указывается не непосредственно в поле адреса команды, а вычисляется по определенному закону в процессе доступа к оперативной памяти.

Более строго, исполнительный адрес - функция, аргументами которой служат значение полей команды и/или содержимое регистров. В случае косвенной регистровой адресации имеем: **Аисп = (RI)**. Здесь - **RI** номер регистра общего назначения, **(RI)** -содержимое этого регистра.

Примечание. Очевидно, что если объем оперативной памяти равен $\mathbf{2^Q}$, то число разрядов **Аисп** должно равняться **Q**.

Например, команда Рис. II.20 определяет пересылку содержимого оперативной памяти адресом 1040H в регистр общего назначения R1 (при условии, что на регистре R2 находится двоичный код 1040H). В этой команде первый операнд имеет прямую регистровую адресацию, второй операнд – регистровую косвенную адресацию.

4.6.4. Автоинкрементная и автодекрементная адресация

Косвенная регистровая адресация позволяет эффективно реализовать последовательный доступ к ячейкам массива за счет последовательного прибавления константы (единицы) к содержимому регистра, содержащего исполнительный (косвенный) адрес. Для более эффективного использования этого свойства введены две разновидности косвенной регистровой адресации.

Автоинкрементная адресация обеспечивает вычисление исполнительного адреса, как и при косвенном регистровом способе, а затем автоматически увеличивает исполнительный адрес, хранящийся в **РОН**, на длину операнда в байтах (Рис. II.21).

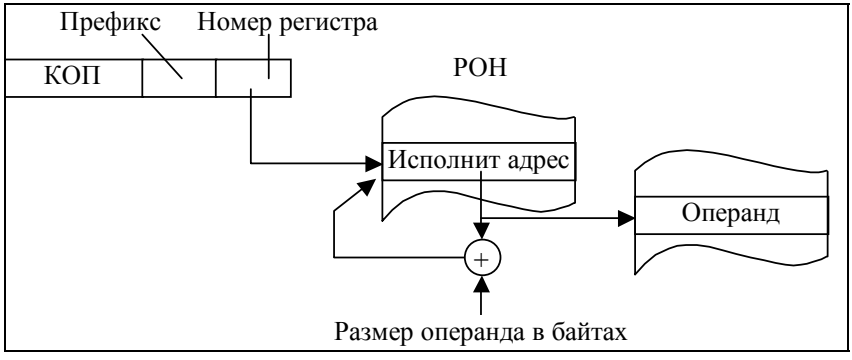


Рис. II.21. Автоинкрементная адресация

Автодекрементная адресация аналогична автоинкрементной адресации и обеспечивает вычитание размера операнда в байтах из исполнительного адреса.

4.7. Многокомпонентные способы адресации

Уменьшение числа адресов команды позволило до некоторой степени расширить адресное пространство оперативной памяти. Но не столь радикально, как того требовали программисты. Необходимость радикального расширения адресного пространства без замедления выполнения команд, а также специфика мультипрограммного режима функционирования вычислительной машины привели к разработке многокомпонентных способов адресации, прежде всего, присоединенной адресации и относительной адресации по базе.

4.7.1. Страничная организация памяти и присоединенная адресация

Прообраз страничной организации памяти появился в трехадресных машинах типа М-20. Как сказано выше, 12 разрядов поле адреса допускали адресацию блока памяти (страницу) размером $2^{12} = 4096$ регистров. Можно поставить еще один блок (страницу) памяти, но как адресовать эту расширенную память? Чтобы разрешить эту коллизию, было предложено разбивать оперативную память на блоки и ввести в состав архитектуры вычислительной машины регистр приращения. В каждый момент времени центральный процессор работал с блоком оперативной памяти, номер которого хранился в двухбитовом регистре приращений. В систему команд вычислительной машины были введены специальные команды установки и запоминания значения регистра приращений. Таким образом, адресное пространство расширялось до четырех блоков (страниц) оперативной памяти.

В общем случае **страничная организация памяти** предполагает разбиение оперативной памяти на равные по размеру страницы (блоки). Пусть имеется **N** страниц, каждая из которых имеет длину **L**. На этой основе реализуется присоединенная адресация (Рис. II.22), которая использует исполнительный адрес **Аисп**, состоящий из двух компонент:

Аисп = <номер страницы>⊕<номер строки>.

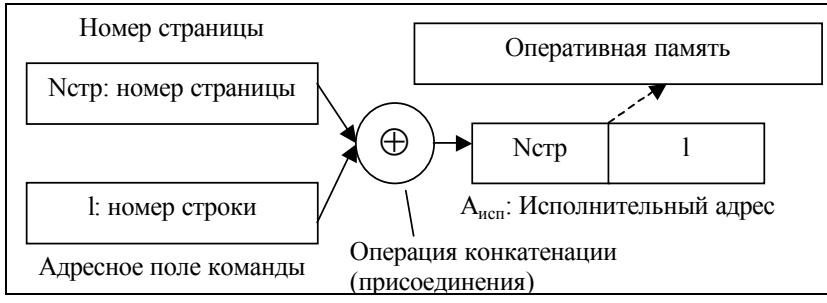
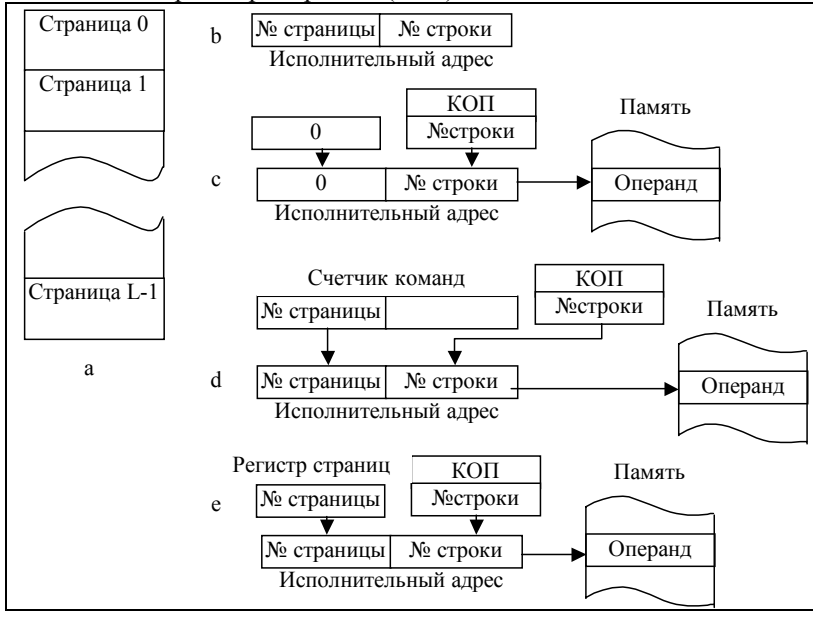


Рис. II.22. Вычисление исполнительного адреса при страничной адресации оперативной памяти

При этом в адресном поле команды записывается только номер строки. Тогда как в качестве номера страницы может использоваться номер базовой страницы, номер текущей страницы или номер, хранящийся на специальном регистре страниц (II.23).



II.23 Разновидности присоединенной адресации:

На II.23 изображено: a – деление адресного пространства; b – исполнительный адрес; c – использование базовой страницы; d – использование текущей страницы; e - использование регистра страниц.

Очевидно, что размер адресуемого пространства оперативной памяти не зависит от длины поля адреса команды, который устанавливает только длину страницы, но определяется числом страниц, на которое «нарезана» оперативная память.

В силу того, что исполнительный адрес вычисляется посредством функции конкатенации (присоединения), такая адресация называется **присоединенной адресацией**.

Здесь и далее также используется понятие **исполнительного адреса**, который хранится на специальном регистре и является физическим адресом доступа к регистру оперативной памяти. Исполнительный адрес состоит из нескольких компонент и вычисляется при каждом обращении к оперативной памяти.

4.7.2. Сегментная организация памяти и адресация с индексированием

Страничная организация оперативной памяти и связанная с ней присоединенная адресация может использоваться при мультипрограммном режиме работы вычислительной системы. В этом случае несколько задач (программ вместе с данными) располагаются в оперативной памяти одновременно, и каждой задаче выделяется целое число страниц. Очевидно, что оперативная память расходуется нерационально: задаче, занимающей всего несколько строк в странице, отводится страница целиком.

Этого недостатка лишен способ адресации, который использует **сегментную организацию** оперативной памяти. По сути дела, это модификация страничной организации памяти, при которой страницы могут иметь различную длину. В этом случае задаче (программе, вместе с данными) выделяется всего один сегмент, но достаточной длины (Рис. II.24).

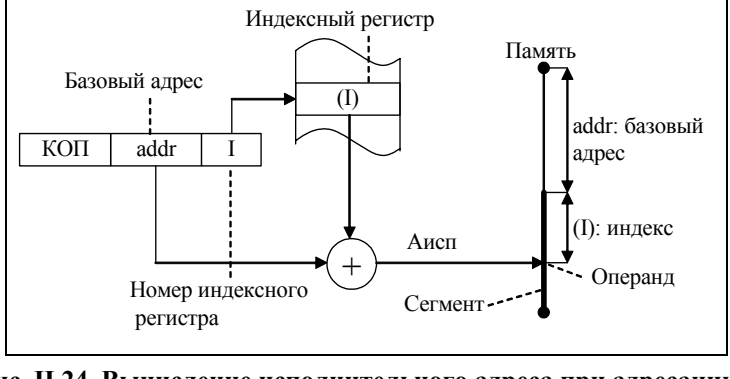


Рис. II.24. Вычисление исполнительного адреса при адресации с индексированием

При **адресации с индексированием** два компонента адреса объединяются путем сложения. Этот способ является удобным средством для организации доступа к массивам и таблицам. Как показано на Рис. II.24, составной частью команды является базовый адрес **addr**, который всегда является физическим адресом оперативной памяти. Индекс (**I)** хранится в специальном индексном регистре **I**, который либо существует отдельно, либо является одним из регистров **РОН**. При вычислении исполнительного адреса индекс прибавляется к базовому адресу:

$$\textbf{Аисп = addr+(I).} \qquad \textbf{(II.10)}$$

На Рис. II.25представлена процедура переписи элементов из Массива1 в Массив2, использующая адресацию с индексированием и автоинкрементную адресацию.

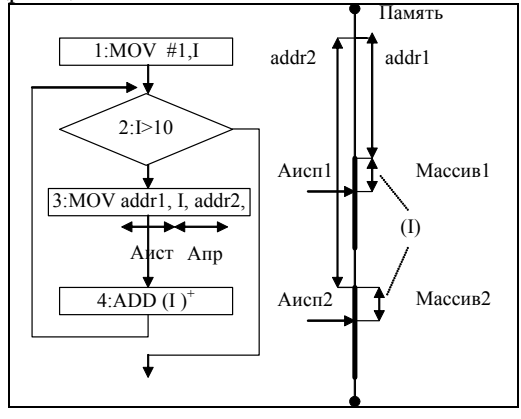


Рис. II.25. Перепись элементов массива с использованием адресации по индексу.

Здесь addr1 – базовый адрес первого массива; addr2 –базовый адрес второго массива; I – индексный регистр. Команды: 1- занесение непосредственной константы (единицы) на индексный регистр; 2- условие окончания переписи; 3 –пересылка элемента из массива в массив; 4 – увеличение индексного регистра на единицу (автоинкрементная адресация).

4.7.3. Адресация по базе

Способ **адресации по базе** также использует сегментную организацию памяти и похож на адресацию с индексированием, по этой причине их часто путают.

При способе адресации с индексированием команда содержит базовый адрес, а в индексном регистре находится смещение. Базовый адрес всегда является физическим адресом оперативной памяти. Смещение может быть как положительным, так и отрицательным и указывает положение операнда относительно базового адреса.

Еще раз подчеркнем, что базовым адресом является физический адрес оперативной памяти, что делает команду "длинной". Поэтому, индексная адресация не преследует цель эффективной адресации памяти большого объема, но позволяет эффективно программировать процедуры работы с массивами и списками.

При адресации по базе ситуация обратная – команда содержит смещение, а в специальном базовом регистре **B** размещен базовый адрес **(B)** (Рис. II.26). Как и в случае индексного регистра, базовый регистр либо существует отдельно, либо является одним из регистров **РОН**. При этом команда получается "короткой", что позволяет эффективно работать с оперативной памятью большого объема.

Исполнительный адрес в случае адресации по базе вычисляется по формуле:

$$\textbf{Аисп} = (B)+D. \qquad \textbf{(II.11)}$$

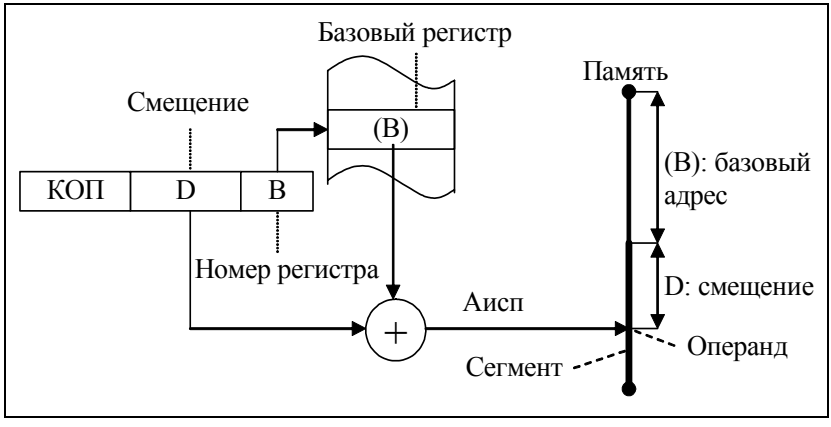


Рис. II.26. Вычисление исполнительного адреса в случае адресации по базе.

Во втором адресе команды указывается регистр **B**, содержимое которого **(B)** прибавляется к смещению при вычислении исполнительного адреса.

4.7.4. Адресация по базе с индексированием

Здесь также используется сегментная организация памяти. Адрес **(B)** начала сегмента в оперативной памяти называется базой задачи (точнее, базовым адресом задачи) и хранится на **регистре базы B**.

Все адреса в программе задачи записываются так, как бы программа размещается в оперативной памяти, начиная с нулевого адреса (относительная адресация). Тогда исполнительный адрес текущей команды вычисляется по следующей формуле:

$$\textbf{Аисп.команды} = (B)+\textbf{счетчик команд}. \qquad \textbf{(II.12)}$$

Адресация команд по базе с индексированием позволяет реализовать перемещаемость программ в оперативной памяти. Это необходимо при мультипрограммном режиме работы вычислительной системы, поскольку при различных сеансах исполнения программы супервизор загружает соответствующую ей задачу в различные области памяти из "соображений" наличия свободной памяти в данный момент. При загрузке задачи супервизор передает на регистр базы базовый адрес задачи, и базовая адресация обеспечивает правильную последовательность исполнения команд вне зависимости от местоположения программы (Рис. II.27).

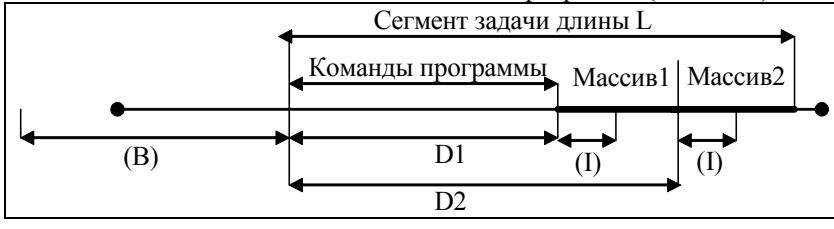


Рис. II.27. Задача, загруженная в оперативную память

Адресация компоненты задачи по базе (например, массива данных) определяется **смещением D** и хранится в поле адреса команды. Таким образом, исполнительный адрес начала компоненты (массива) определяется по формуле:

$$\textbf{Аисп.компоненты} = (B) + D. \qquad \textbf{(II.13)}$$

Местоположение элемента компонента определяется индексом **I**, который хранится на **индексном регистре**:

$$\textbf{Аисп.элемент} = (B) + D + (I). \qquad \textbf{(II.14)}$$

Таким образом, исполнительный адрес Аисп **образуется путем арифметического сложения смещения** D **из поля адреса команды, содержимого (**B**) регистра базы и содержимого (**I**) индексного регистра (**

Рис. II.28).

Как сказано выше, содержимое регистра базы адресует начало сегмента программы в оперативной памяти. Для 32 разрядного регистра базы **РС** это значение может изменяться от 0 до 2^{32}. Длина сегмента определяется смещением, т.е. размером адресного поля команды.

Примечание. Присоединенная и базово–индексная адресации были изобретены до появления косвенной регистровой адресации. Последняя позволяет существенным образом расширить адресуемое пространство оперативной памяти, используя сравнительно короткое поле адреса в команде. Для 16 разрядных машин размер адресуемого пространства q = 2^{16}=64 килобайта, для 32-х разрядных машин -q=2^{32}=2^2*2^{30}=4 терабайта.

Если для 16-ти разрядных машин размер адресуемого пространства оперативной памяти был явно недостаточен, то для 32-х разрядных машин (персональных) этого размера хватает с избытком (по крайней мере, пока).

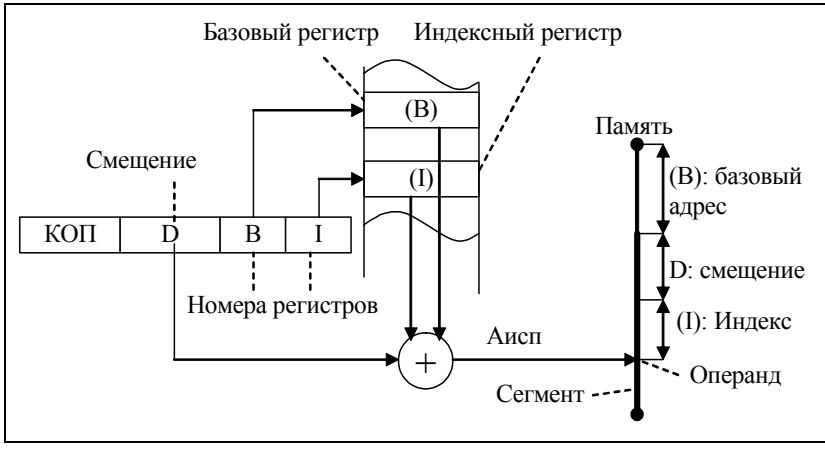


Рис. II.28. Вычисление исполнительного адреса в случае адресации по базе с индексированием

Может сложиться мнение, что в архитектуре современных вычислительных машин от присоединенной и относительной адресации можно оказаться. Но имеются проблемы, которые эффективно решаются с применением этих методов адресации: организация виртуальной и кэш памяти, организация мультипрограммного режима работы.

4.7.5. Относительная адресация

Адресацию, при реализации которой исполнительный адрес вычисляется как сумма фиксированного смещения в команде и текущего значения счетчика команд, называют **относительной адресацией**. При этом смещение **D** может быть как положительным, так и отрицательным:

Аисп = (счетчик команд) +D.

При короткой относительной адресации используется смещение небольшой величины для указания адреса команд, находящихся недалеко от текущей команды. Наиболее широкое распространение короткая относительная адресация получила в командах перехода. (Статистика показывает, что 80% команд перехода находятся в пределах 127 байт от местоположения текущей команды.)

Длинная относительная адресация (смещение большой величины) используется как в командах перехода, так и в общих командах обработки данных.

Относительный способ адресации, так же как и базовый способ адресации, позволяет создавать перемещаемые программы.

4.8. Специальные виды памяти

4.8.1. Организация виртуальной памяти

Память вычислительной машины является многоуровневой. Рассмотрим два основных уровня: оперативная и внешняя (дисковая) память. Оперативная память более быстродействующая по сравнению с внешней памятью, но имеет гораздо меньший объем. В силу специфики функционирования, обмен данными между оперативной и внешней памятью осуществляется блоками. В результате появляется проблема программирования обработки "по частям" больших объемов данных, размещаемых на внешних устройствах.

Привлекательна идея организации абстрактного адресного пространства памяти большого объема, относительно которого записывается программа обработки данных. При этом подразумевается, что существует система управления абстрактной памятью, которая берет на себя все трудности организации обработки в двухуровневой памяти. Такая абстрактная память получила название **виртуальной памяти**. Рассмотрим принципы ее организации.

В оперативной памяти выделяется какое-то число страниц фиксированной длины. Внешняя память (или часть ее) нарезается на страницы такой же длины (Рис. II.29). Предполагается, что число страниц внешней памяти значительно больше числа страниц оперативной памяти.

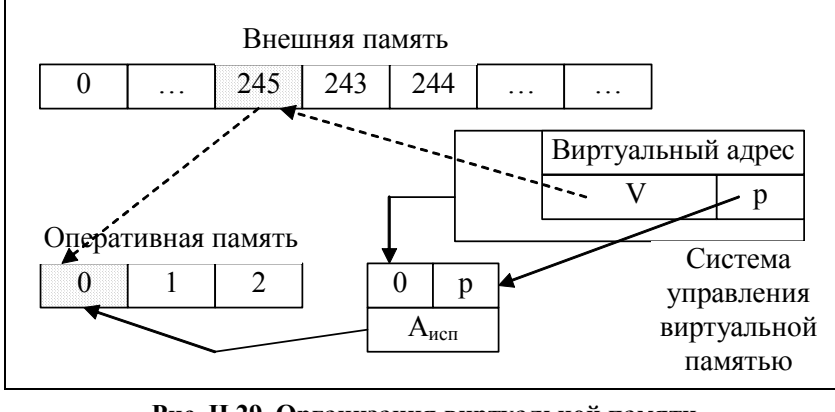


Рис. II.29. Организация виртуальной памяти

Вводится понятие виртуального (абстрактного) адреса, который состоит из двух компонент, как в присоединенной адресации:

Виртуальный адрес = V⊕p,

где **V** - номер страницы внешней памяти, **p** - номер строки в странице.

Когда в поле адреса команды встречается виртуальный адрес **Vp**, система управления виртуальной памятью (СУВП) обрабатывает его посредством процедуры (Рис. II.30):

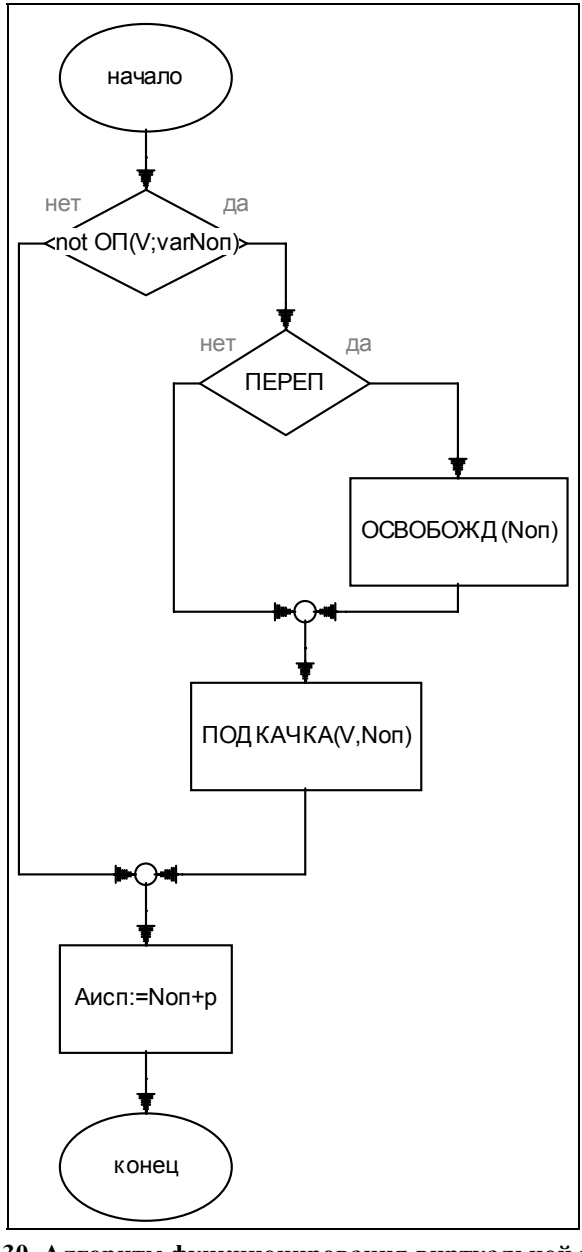


Рис. II.30. Алгоритм функционирования виртуальной памяти

Здесь используются логические функции.

Function *Страница_ОП*(V; var $N_{оп}$): Boolean;

выдает значение **True**, если страница внешней памяти **V** переписана в страницу $N_{оп}$ оперативной памяти, и выдает значение **False** в противном случае.

Function *Переполнение*: Boolean;

выдает значение **True**, если нет свободных страниц в оперативной памяти.

Также используются вспомогательные процедуры.

Procedure Освобождение($N_{оп}$);

освобождение страницы оперативной памяти в соответствии с каким-либо правилом.

Простейшее правило - освобождать страницу, к которой дольше всего не было обращения. При этом различаются два варианта освобождения страницы. Если в освобождаемую страницу не было записи, то номер страницы заносится в список пустых страниц, иначе страница из оперативной памяти переписывается на свое место во внешней памяти и только после этого ее номер заносится в список свободных страниц.

Procedure *Подкачка*(V;$N_{оп}$);

перепись содержимого страницы внешней памяти **V** в страницу оперативной памяти $N_{оп}$.

Таким образом, система управления виртуальной памятью обеспечивает работу непосредственно со страницами оперативной памяти, подкачивая в нее необходимые страницы из внешней памяти в случае необходимости.

Как уже говорилось, это обеспечивает эффективность программирования с использованием адресов большого адресного пространства виртуальной памяти. Кроме того, использование виртуальной памяти позволяет конструировать информационные системы обработки данных любого объема (в пределах размера виртуальной памяти). Причем, небольшой объем данных, который может разместиться в оперативной памяти, обрабатывается быстро, а данные большего объема обрабатываются медленнее, но все же обрабатываются.

Быстродействие обработки данных с использованием виртуальной памяти зависит от частоты смены страниц в оперативной памяти (малая скорость обмена между внешней памятью и оперативной памятью). В большинстве случаев вероятность того, что программа, затребовав доступ к странице $N_{оп}$, следующий доступ затребует к той же странице, достаточно велика (гипотеза компактного доступа к данным). Поэтому виртуальная память обеспечивает не только эффективность программирования в виртуальных адресах, но также быстродействие обмена данными в системе внешняя память – оперативная память.

4.8.2. Организация кэш памяти

Своеобразным вариантом виртуальной памяти служит распространенная в современных компьютерах система обмена данными между центральным процессором и оперативной памятью с использованием **кэш памяти**.

Как говорилось ранее, основное время выполнения команды составляет время обмена данными между оперативной памятью и центральным процессором. Кэш-память служит быстродействующим посредником между ними. Являясь относительно дорогостоящим устройством, кэш-память имеет небольшой объем по сравнению с оперативной памятью и хранит данные, к которым центральный процессор обращается достаточно часто.

Рис. II.31. Взаимодействие центрального процессора и оперативной памяти с использованием посредника - кэш памяти

Здесь уместна следующая аналогия. Центральный процессор – научный работник, оперативная память – книжный шкаф, кэш-память – рабочий стол. В процессе постоянной работы с литературой на письменном столе располагаются книги, к которым научный работник обращается достаточно часто. Но если потребовалась книга, которой нет на письменном столе, она всегда может быть перемещена из книжного шкафа на письменный стол (либо на свободной место, либо вместо книги, которая в данный момент не нужна).

Самой простой организацией обладает кэш-память с прямым отображением страниц оперативной памяти на строки кэш памяти. В этом случае в адресе оперативной памяти выделяется три поля: номер тома (тег), номер страницы в томе и смещение в странице (Рис. II.32).

Пусть, например, имеется оперативная память размером $2^{10} = 4096$ байт. Исполнительный адрес в этом случае содержит 10 разрядов Пусть также кэш-память состоит из восьми строк, каждая из которых может хранить восемь страниц оперативной памяти.

В этом случае исполнительный адрес подразделяется на три поля: тег (том), страница в теге, смещение в странице. Всего имеется 16 тэгов (томов), в каждом томе содержится 8 страниц длиной в 8 строк каждая. Таким образом, всего $2^{10} = 1024$ байт. Для любого тега страница **I** отображается в строку **I** кэш-памяти. Таким образом, в строку **I** кэш-памяти отображается 16 страниц оперативной памяти (по одной из каждого тэга). Для того, чтобы определить, страница какого тома хранится в строке **I** кэш-памяти, используется дополнительное поле строки кэш памяти, хранящее номер тома.

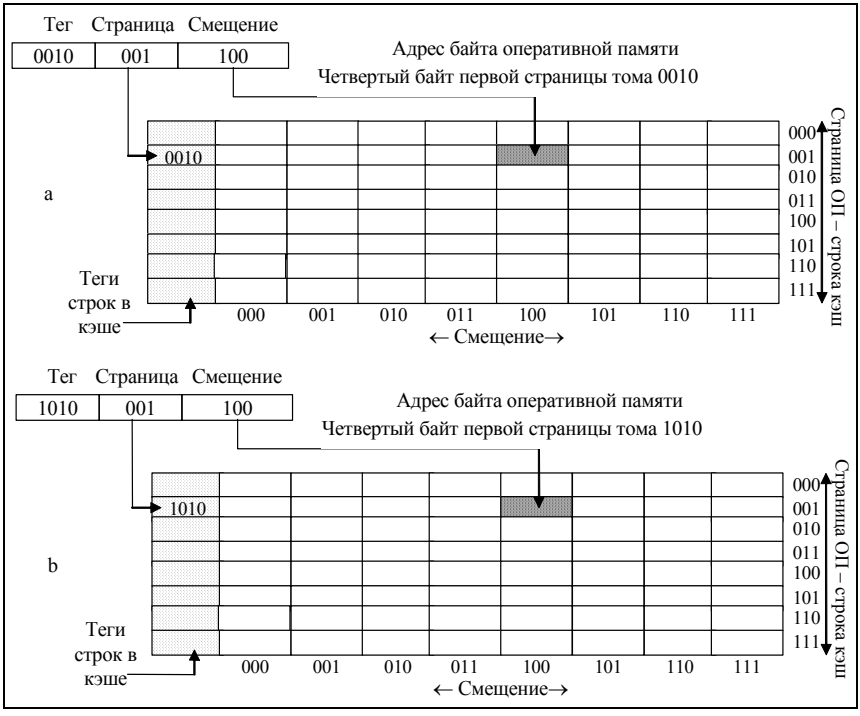


Рис. II.32. Кэш память с прямым отображением

Пусть, в какой-то момент времени программа требует доступ к байту оперативной памяти с адресом 0010 001001.Система управления памятью сначала проверяет гипотезу, что этот байт хранится в строке 001 кэш-памяти. Для этого сравнивается тэг адреса с содержимым поля тэга в строке 001 кэш памяти. Если сравнение проходит, реализуется доступ к элементу 100 (смещение) строки 001 кэш памяти. На Рис. II.32a) изображен этот случай. Если же, например, требуется доступ по адресу 1001 001 100, то такое сравнение не проходит и в строку 001 кэш памяти переписывается страница 001 из тома (тэга) 1001 оперативной памяти (Рис. II.32b).

Преимуществом организации кэш-памяти с прямым отображением является достаточно высокое быстродействие, поскольку требуется всего лишь одна операция сравнения для (тегов). Недостатком является множественное отображение страниц оперативной памяти на одну и ту же строку кэш памяти. Например, если две страницы оперативной памяти с одинаковыми номерами, но из разных тэгов , используются одинаково часто, происходит постоянная смена содержимого одной и той же строки кэш памяти, что существенно увеличивает время доступа к данным.

Кроме прямого отображения, используются другие способы организации кэш-памяти, которые имеют как свои недостатки, так и свои преимущества. Но назначение кэш памяти не зависит от способа ее организации.

В силу того, что центральный процессор работает не непосредственно с оперативной памятью, а использует посредника в виде кэш памяти, возникает проблема поддержания соответствия одних и тех же единиц данных, хранящихся в посреднике и в оперативной памяти (проблема целостности данных).

Эта проблема возникает при замещении страницы оперативной памяти, хранящейся в строке **I** кэш-памяти, новой страницей с тем же номером, но из другого тега. Смена содержимого строки **I** кэш-памяти реализуется следующим образом:

- если в строку кэш-памяти I записи не производилось, в нее можно переписывать новую страницу оперативной памяти **I** из тэга **T2**;
- в противном случае, перед переписью новой страницы из тега **T2** в строку **I** кэш-памяти, страница оперативной памяти **I** из тега **T1** заменяется содержимым строки **I** кэш-памяти.

Выше мы обсуждали способ повышения быстродействия компьютера за счет использования кэш-памяти для данных. Что касается команд, в программах часто встречаются участки, многократно выполняемые в цикле. При этом центральный процессор многократно считывает из оперативной памяти одни и те же команды. Если использовать кэш-память для промежуточного запоминания команд и если эта память достаточного объема, чтобы хранить все команды одного цикла, вычисления будут выполняться с большей скоростью. Более того, для каждого типа компьютеров можно разрабатывать оптимизирующие компиляторы, которые анализируют заданные программистом циклы и разбивают их на циклы меньшего размера, размещающиеся в кэш памяти команд целиком.

Более того, используются кэш-памяти для дисковой памяти, что позволяет существенным образом снять проблему замедления вычислений при работе с внешней памятью.

4.9. Контрольные вопросы

7. Расширение размеров адресного пространства за счет уменьшения числа адресов команды.
8. Схема выполнения трехпараметрических действий двухадресными и одноадресными командами.
9. Однокомпонентные способы адресации.
10. Присоединенная адресация.
11. Адресация по базе.
12. Стековая адресация.
13. Виртуальная память.
14. Кэш-память как разновидность виртуальной памяти.

5. Структурная организация вычислительной машины

Рассмотрение фундаментальных принципов автоматической обработки данных позволило выявить формы представления данных и номенклатуру действий, необходимых для решения широкого круга задач. Чтобы реализовать заданный набор действий, необходима соответствующая структура – определенный набор устройств, объединенных посредством линий связи в одно целое. Выделяются три типа устройств, образующих вычислительную машину: центральный процессор, оперативная память, устройства ввода – вывода (**УВВ**). К устройствам ввода – вывода (терминальным устройствам) относятся: устройства ввода (клавиатура, сканер и т.д.); устройства вывода (монитор, принтер и т.д.); устройства долговременного хранения данных (магнитные диски и лазерные диски и т.д.).

Общим для внешних устройств является:

- обмен данными между оперативной памятью и УВВ;
- необходимость преобразования форматов "центра" – центральный процессор в совокупности с оперативной памятью, к различным специфическим форматам устройств ввода - вывода.

Внешнее устройство обычно состоит из механического и электронного компонента. Электронный компонент называется устройством управления вводом – выводом **УУВВ** (контроллером или адаптером). Механический компонент представляет собственно устройство. Некоторые контроллеры могут управлять несколькими устройствами.

В 1952 году была запущена в эксплуатацию вычислительная машина, функционирующая в соответствии с принципами фон Неймана. Этот проект до сих пор используется в большинстве современных компьютеров. Выпускающиеся с тех пор вычислительные машины достаточно условно можно разделить на четыре группы: суперкомпьютеры, большие компьютеры, миникомпьютеры и персональные компьютеры. Показатели вычислительной мощности и объема запоминающих устройств для каждого класса постоянно увеличивались, но соотношение "большой – средний – малый" остается до сих пор. Первые вычислительные машины следует отнести к большим компьютерам, других просто не было (М – 220, БЭСМ – 4). К 70-м годам прошлого века, параллельно с большими вычислительными машинами (IBM-370, ЕС – ЭВМ, БЭСМ-6) появилась концепция средних или мини компьютеров (PDP-11, СМ, ЭЛЕКТРОНИКА). В 1981 году началась эра персональных компьютеров (IBM PC, ЕС 18**, ЭЛЕКТРОНИКА 88).

В 1974 году был создан первый векторный суперкомпьютер CRAY –1, первый суперскалярный компьютер был создан в 1990 году.

Первый отечественный суперкомпьютер (многопроцессорный вычислительный комплекс Эльбрус-1 со скоростью выполнения операций 15 млн. операций/сек) был разработан в 1980 году. Следующая версия (Эльбрус-2, быстродействие 125 млн. операций/сек)– в 1985 году. Эти вычислительные комплексы выпускались в ограниченном количестве, последняя версия (Эльбрус 3.1) реализована в виде четырех экземпляров в 1990 – 95 годах.

Со времени появления первых вычислительных машин организация ввода-вывода претерпела значительные изменения. Произошло это отчасти потому, что вычислительные машины начали использоваться для решения задач, требующих большого объема памяти, В связи с этим, в состав вычислительной машины, наряду с оперативной памятью, стали подключать в качестве устройств ввода – вывода внешние запоминающие устройства большой емкости. Кроме того, на организацию управления УВВ повлияло большое различие в скорости работы УВВ и внутренним быстродействием компьютера.

Замечание. По сути дела, теперь следует говорить не об отдельной вычислительной машине, а о **вычислительной системе**, состоящей из многих специализированных процессоров ввода – вывода и одного или нескольких центральных процессоров. Также изменилась семантика некоторых терминов. Так, например, под интерфейсом ранее понимались устройства, которые сейчас принято называть шинами. А интерфейс в настоящем понимании – контроллер, который управляет действиями УВВ в соответствии с командами процессора, преобразует данные из внутреннего представления в форматы конкретных внешних устройств и выполняет обратное преобразование.

Ниже рассматриваются возможные схемы подключения устройств ввода-вывода к центральному процессору и оперативной памяти.

5.1. Вводом-выводом управляет центральный процессор (прямое управление)

При использовании **прямого управления** в состав устройства управления компьютера, наряду с устройством автоматического исполнения программы (**УУАВП**), входит также устройство управления вводом – выводом **УУВВ** (Рис. II.33).

Прямое управление подразумевает наличие специальных команд для программирования ввода и вывода. Существенно, что такие команды выполняются непосредственно центральным процессором. Это не вызывает вопросов с точки зрения логики программирования, но приводит к чрезвычайно неэффективной работе вычислительной системы.

Дело в том, что для реализации ввода – вывода использовались электромеханические устройства: читающее устройство с перфокарт; алфавитно-цифровое печатающее устройство; накопители на магнитных лентах, барабанах и дисках.

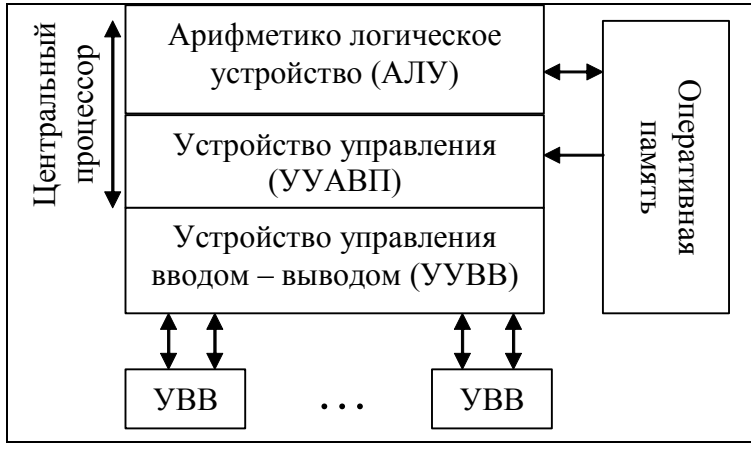


Рис. II.33. Прямое управление устройствами ввода – вывода компьютеров первого поколения

Но эти электромеханические устройства работали неизмеримо медленнее электронных устройств центрального процессора. (Пока молоточек с литерой устройства вывода бьет по бумаге, центральный процессор простаивает. Но за это время он способен выполнить большое число команд обработки и управления.) В результате, в случае программ, содержащих значительное число команд ввода-вывода, центральный процессор в основном простаивает. Накопители на магнитных носителях, а также клавиатуры и мониторы, которые добавились впоследствии, работают гораздо быстрее, но проблема по-прежнему остается.

Для решения этой проблемы конструкторы предложили выполнять устройство управления вводом – выводом в виде отдельного специализированного процессора (или несколько специализированных процессоров), реализующего только команды ввода и вывода. Главная идея этого нововведения состоит в следующем: выполняя команду ввода–вывода центральный процессор не занимается передачей данных между оперативной памятью и внешними устройствами, а лишь запускает УУВВ и продолжает выполнять следующие команды программы. ***Устройство ввода - вывода выполняет работу по пересылке и преобразованию данных, функционируя параллельно с центральным процессором***.

5.2. Буферизация данных от ввода – вывода

Первым шагом в этом направлении являлась организация **буфера ввода – вывода**, реализованного в виде либо одного буферного регистра, либо массива буферных регистров с собственным управлением.

Выполнение команды вывода-вывода центральным процессором заключается в том, что он передает в устройство управления буфером адрес ячейки оперативной памяти и сигнал запуска. После этого буфер ввода – вывода и центральный процессор работают параллельно: первый – занимается реализацией вывода, второй – продолжает выполнять следующие команды за командой вывода.

5.3. Структурная организация больших компьютеров

С целью еще большей степени параллелизма функционирования центрального процессора и УВВ был разработан так называемый **канал** данных. Канал данных представляет собой специализированный процессор, созданный для выполнения только операций ввода–вывода (Рис. II.34). Когда в центральном процессоре выполняется команда ввода–вывода, то сам процессор операцию ввода – вывода не выполняет, а только инициирует ее выполнение в канале и передает каналу управление этой операцией. Если в системе имеется несколько каналов, то одновременно могут выполняться несколько операций ввода–вывода.

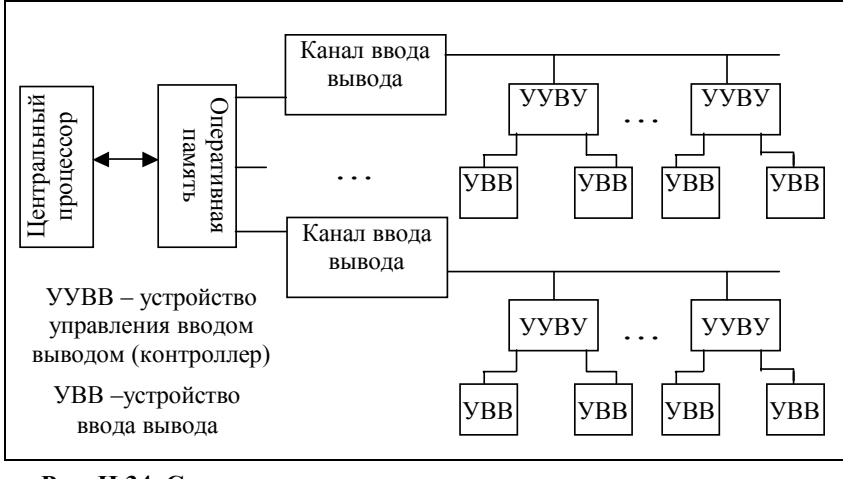


Рис. II.34. Структурная организация с каналами ввода – вывода

При такой **канальной организации ввода–вывода** в систему команд вычислительной машины входят специальные команды ввода–вывода, которые задают как начало области оперативной памяти, так и адрес устройства ввода вывода.

Как уже говорилось, канал является специализированным процессором со своей системой команд. Последовательность таких команд образует программу канала, которая и выполняется при реализации операций ввода – вывода.

При выполнении команды вывода канал последовательно выбирает слова из оперативной памяти и переносит их на устройство вывода. При выполнении команды ввода канал последовательно записывает поступающие с устройства ввода слова в оперативную память.

Канал, который может выполнять одновременно с центральным процессором только одну операцию ввода–вывода, принято называть **селекторным каналом.** Такой канал обеспечивает большую скорость передачи данных и используется для работы с высокоскоростными УВВ, прежде всего с магнитными дисками.

Мультиплексный канал может работать одновременно с большим количеством низкоскоростных **УВВ**, таких как принтеры, сканеры и т.д.

5.4. Структурная организация миникомпьютеров и персональных компьютеров

5.4.1. Порты ввода – вывода

Порт ввода – вывода (специфический регистр ввода – вывода) является частью интерфейса (контроллера) устройства ввода – вывода и представляет собой группу разрядов, значение которых доступно центральному процессору во время исполнения операций ввода – вывода.

На Рис. II.35изображена структура порта, используемого для ввода с клавиатуры. Он состоит из двух регистров: регистр данных **KBDATA** и управляющий регистр **KBCS**. На регистр данных заносится код нажатой клавиши, который считывается центральным процессором при взаимодействии с управляющим регистром.

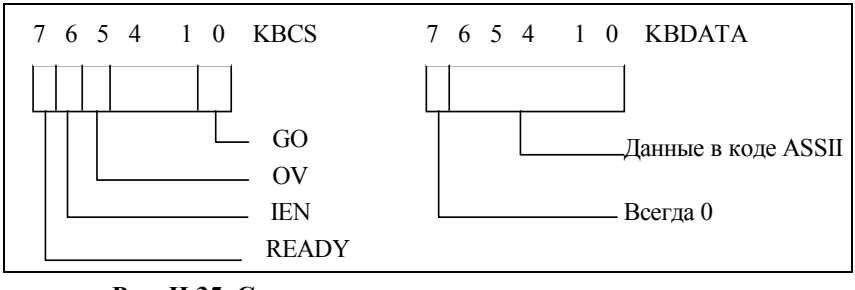


Рис. II.35. Структура порта для ввода с клавиатуры.

В разряд **GO** центральный процессор заносит "1" в случае готовности принять символ с клавиатуры. При этом сбрасывается разряд **READY** и активизируется операция ввода. Для клавиатуры активизация операции состоит в сбросе разрядов регистра данных и в ожидании нажатия клавиши. При нажатии клавиши соответствующий ей код заносится на регистр данных и разряд **READY** устанавливается в "1".

Разряд **OV** указывает на состояние переполнения и устанавливается в "1", когда с клавиатуры поступают два символа подряд без промежуточного чтения процессором порта **KBDATA**.

5.4.2. Раздельные шины данных

В этом случае оперативная память подключается к центральному процессору посредством шины памяти, а подсистема ввода – вывода подключается к центральному процессору посредством шины ввода – вывода (Рис. II.36). Как шина памяти, так и шина ввода-вывода включают в себя: линии адреса, линии данных, линии управления.

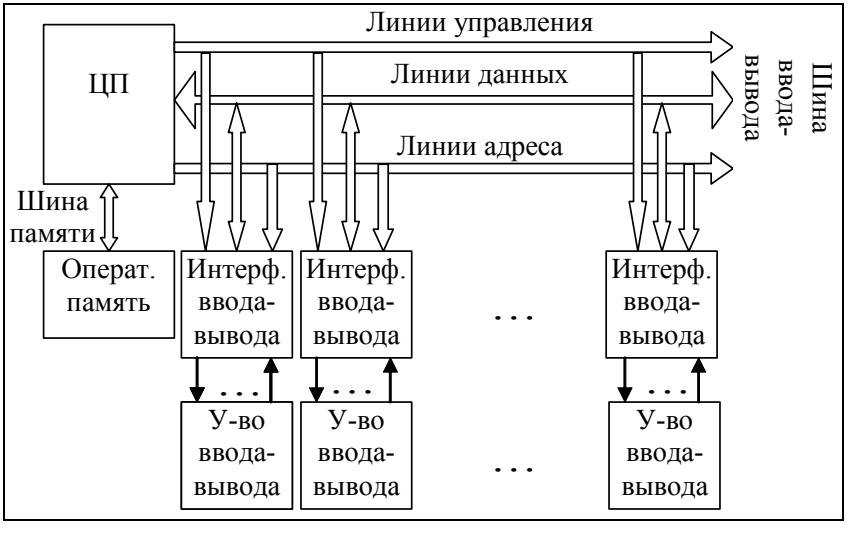


Рис. II.36. Структурная организация с шиной памяти и шиной ввода – вывода

Так как основная оперативная память и порты ввода-вывода (в составе интерфейсов ввода – вывода) подключены к разным шинам, адресные пространства для команд ввода – вывода и для остальных команд различны.

Адресные линии шины ввода – вывода позволяют выбирать конкретный порт, тогда как адресные линии шины памяти – конкретную ячейку оперативной памяти. Выбор типа шины определяется типом выполняемой команды: то ли это команда ввода – вывода, то ли любая другая команда. В случае такой организации доступ к портам осуществляется посредством специальных команд ввода – вывода, аналогичных командам загрузки **LOAD** и сохранения **STORY**:

INB R, p_n; **REG[R] := IPORT[p_n];**
OUTB R, p_n; **OPORT[p_n] := REG[R].**

Здесь, **IPORT** – массив входных портов, **OPORT** – массив выходных портов, **REG** – массив регистров общего назначения, p_n – адрес порта, **R**– номер регистра общего назначения.

5.4.3. Единая шина данных.

В этом случае, как устройства ввода – вывода, так и оперативная память подключены к единой шине данных (Рис. II.37). Говоря другими словами, это ввод – вывод, организованный по аналогии обращения к памяти.

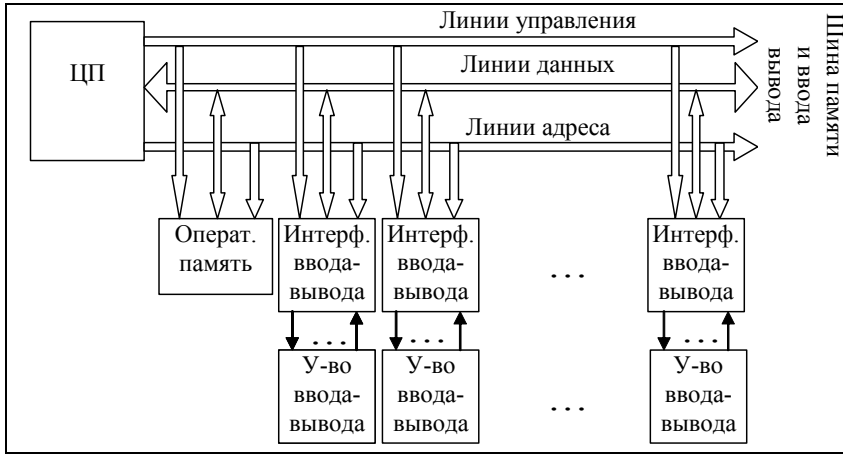


Рис. II.37. Структурная организация с единой шиной

Каждый порт ввода–вывода имеет адрес в пространстве адресов оперативной памяти. Порт ввода допускает выполнение любой команды, которая осуществляет чтение по его адресу, порт вывода–любой команды, которая производит запись по его адресу.

MOV addr_mem, addr_mem - команда пересылки в оперативной памяти;

MOV addr_mem, r_p – команда вывода;

MOV r_p, addr_mem – команда ввода.

В зависимости от того, какое устройство подсоединено к порту с адресом r_p, ввод или вывод будет осуществляться на это устройство.

5.4.4. Драйверы устройств

До сих пор мы говорили о командах ввода–вывода на уровне символьного языка программирования. На самом деле реализация каждой из этих команд на языке двоичных кодов представляет собой достаточно сложную подпрограмму, включающую в себя как обычные команды вычислительной машины, так и специфические команды управления вводом – выводом. И сложность ее не столько в размерах, сколько в необходимости хорошо представлять всю технологию реализации ввода – вывода на уровне аппаратуры.

В силу этого любая операционная система включает в себя встроенные программы управления вводом – выводом, к которым можно обращаться, не вникая в специфику их функционирования. Причем эти программы управляют вводом – выводом на двух уровнях. Уровень, обслуживающий шину данных, не зависит от типа устройств, подключенных к этой шине. Эти программы образуют систему управления вводом–выводом **BIOS**, которая в настоящее время, как правило, размещается в постоянном запоминающем устройстве **(ПЗУ**).

Весь зависимый от устройства код оформляется в виде **драйвера** конкретного устройства. Каждый драйвер управляет устройствами одного типа или, может быть, одного класса. В операционной системе только драйвер устройства знает о конкретных особенностях какого-либо устройства. Например, только драйвер диска имеет дело с дорожками, секторами, цилиндрами, временем установления головки и другими факторами, обеспечивающими правильную работу диска.

Драйвер устройства принимает запрос от операционной системы и решает, как его выполнить. Типичным запросом является чтение **n** блоков данных. Если драйвер был свободен во время поступления запроса, то он начинает выполнять запрос немедленно. Если же он был занят обслуживанием другого запроса, то вновь поступивший запрос присоединяется к очереди уже имеющихся запросов, и он будет выполнен, когда наступит его очередь.

5.5. Контрольные вопросы

1. Прямое управление вводом-выводом.
2. Целесообразность параллельной работы центрального процессора и устройств ввода-вывода.
3. Использование канала ввода-вывода.
4. Понятие порта ввода-вывода.
5. Архитектура с раздельными шинами данных.
6. Архитектура с общей шиной данных.

6. Параллельная работа центрального процессора и устройств ввода-вывода

"Прервем программу, чтобы сделать важное сообщение…" знакомая всем фраза определяет основную функцию системы прерываний: извещение программы о возникновении в устройствах вычислительной машины какой-либо заранее предусмотренной ситуации. Прежде всего, необходимость прерываний возникает при управлении вводом–выводом.

Предположим, что выполняемая программа содержит цикл, включающий в себя команду вывода "печать строки". Выполняя эту команду, центральный процессор заполняет буфер вывода символами строки и запускает устройство вывода (**ВУ**). Пока **ВУ** осуществляет вывод буфера, центральный процессор может продолжать выполнение основной программы и готовить для печати следующую строку. ***Центральный процессор и устройство вывода работают параллельно.*** Но подготовка следующей строки для печати, как правило, выполняется быстрее, чем реализуется вывод предыдущей строки. Т.е. следующая команда "печать строки" не может быть выполнена до завершения вывода предыдущей строки. И центральный процессор простаивает, т.е. быстродействие центрального процессора все равно ограничивается медленным электромеханическим устройством.

Стоимость центрального процессора гораздо выше стоимости внешнего устройства. Поэтому проблема работы центрального процессора без простоев чрезвычайно актуальна. Это возможно реализовать путем включения в состав вычислительной машины нескольких внешних устройств и организации параллельной работы этих устройств и центрального процессора. Управление вводом–выводом в этом случае осуществляется **механизмом прерываний**.

Рассмотрим подробнее работу механизма прерываний в случае наличия в программе цикла вывода нескольких строк (Рис. II.38).

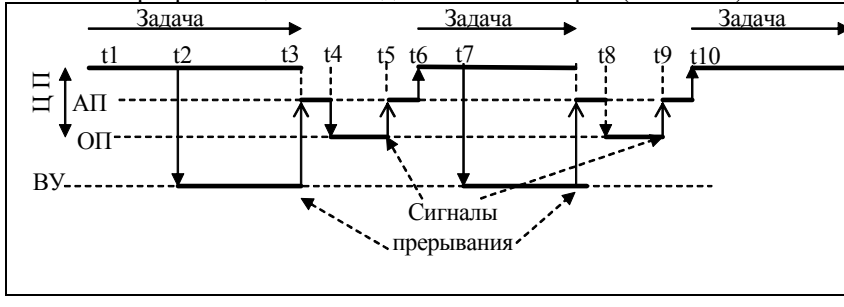


Рис. II.38. Прерывание задачи внешним устройством: нет простоя процессора (АП – анализ прерывания, ОП – обработка прерывания, ВУ – внешнее устройство)

Пусть задача пользователя выполняется в промежутке времени **t1** – **t2**. (Задача - программа пользователя, загруженная в оперативную память и поставленная под контроль центрального процессора.) К моменту **t2** она подготавливает строку текста в буферной памяти и дает сигнал запуска устройства вывода. После этого центральный процессор и внешнее устройство (**ВУ**) функционируют параллельно в промежутке времени **t2** – **t3:** устройство вывода разгружает буфер вывода (выводит строку), а центральный процессор продолжает вычисления и готовит к выводу следующую строку. В момент **t3** внешнее устройство (**ВУ**) заканчивает вывод строки и вырабатывает **сигнал прерывания.** Получив этот сигнал, центральный процессор прекращает выполнение задачи пользователя и запускает системную задачу **АП - анализ прерываний** (**t3** – **t4**). Последняя запускает задачу **обработка прерывания** (**ОП**), которую выполняет центральный процессор (**t4** – **t5**). Задача обработки прерывания производит необходимые действия для завершения вывода строки и в момент своего окончания t5 выдает сигнал программного прерывания, который анализируется анализатором прерываний (**t5** – **t6**) и инициирует продолжение задачи пользователя (**t6**). В момент **t8** задача пользователя выдает команду на вывод следующей строки, и все повторяется.

Прерывание – реакция центрального процессора на некоторое событие, приводящая к прекращению выполнения текущей задачи с целью выполнения какой – либо другой, более важной, задачи. События, требующие прерывания, могут произойти как внутри компьютера (сбои в работе отдельных устройств, попытка деления на нуль, коллизии в устройствах ввода–вывода и т.д.), так и вне его (нажатие клавиши при вводе символа, запрос на совместную работу с другим компьютером и т.д.). Во всех случаях моменты возникновения событий заранее неизвестны и не могут быть учтены при программировании.

В зависимости от характера прерывания и результатов его обработки, выполнение прерванной задачи через некоторое время может продолжиться или будет прекращено.

6.1. Контрольные вопросы

1. Необходимость параллельной работы центрального процессора и устройств ввода-вывода.
2. Прерывания как средство управления вводом-выводом.

7. Мультипрограммный режим работы вычислительной системы

7.1. Разделение времени между задачами

В большинстве случаев имеет место ситуация, когда после запуска внешнего устройства задача пользователя не может продолжаться до завершения его работы. Например, следующая строка для вывода подготавливается центральным процессором гораздо быстрее реализации вывода предыдущей строки (Рис. II.39). В этом случае центральный процессор простаивает в ожидании завершения работы внешнего устройства.

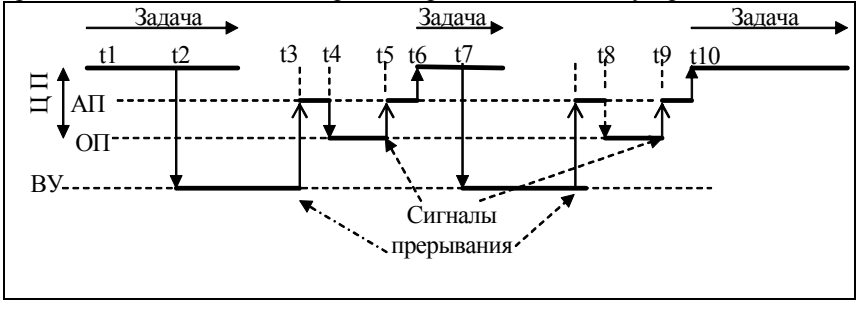


Рис. II.39. Прерывание от внешнего устройства: простой процессора(АП – анализ прерывания, ОП – обработка прерывания, ВУ – внешнее устройство)

Для полной загрузки центрального процессора с целью наибольшей эффективности его использования применяется **мультипрограммный режим** функционирования вычислительной системы (Рис. II.40).

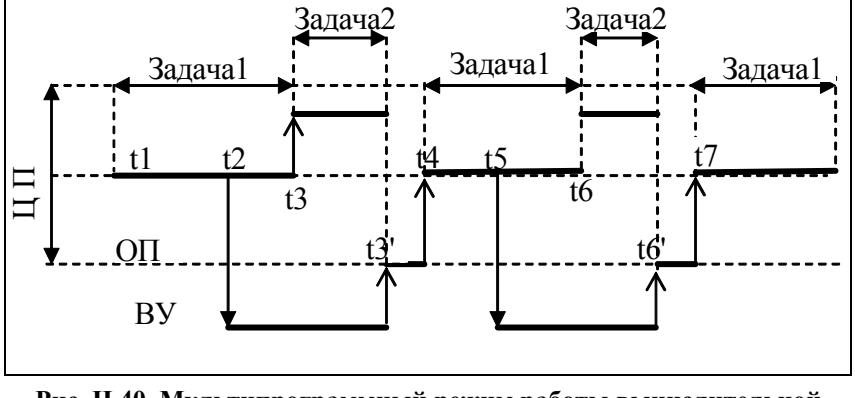


Рис. II.40. Мультипрограммный режим работы вычислительной системы (ОП – обработка прерываний, ВУ – внешнее устройство)

В этом случае в оперативной памяти одновременно присутствуют несколько задач пользователя, например, **Задача1** и **Задача2.**

Когда, отдав приказ на вывод (**t2**) и подготовив вывод следующий строки, **Задача1** не может продолжать работу, она выдает сигнал прерывания, который запускает на выполнение **Задачу2** (**t3**). (Для простоты рисунка работа анализатора прерываний здесь и далее не представлена). Теперь сигнал прерывания от внешнего устройства прерывает **Задачу2** в момент времени **t3'** и запускает **Задачу1**на продолжение. Последняя выдает приказ на вывод в момент **t4** и т.д.

Примечание. Обычно прерывания допускаются только между командами, а не во время выполнения команды.

7.2. Прерывания и их обработка

В случае лишь одного прерывающего устройства или события дела обстоят сравнительно просто. Однако при наличии многих таких устройств или событий сразу возникает вопрос приоритетности обслуживания. Каждое прерывающее устройство должно как-то идентифицировать себя. Эта идентификация должна быть подробной: группа, к которой относится устройство; указатель конкретного устройства; признак прерывающего события. Так, например, каждому принтеру может быть определено несколько прерываний: "работа окончена"; "нет бумаги"; "разорвана бумага"; "неисправность в электронике"; "ненормальная температура" и т.д.

Существенно, что вызывающие прерывания события в вычислительной системе возникают независимо друг от друга. И в каждый момент времени может возникнуть несколько различных прерываний. Для исключения коллизий, каждому типу прерываний присвоен определенный приоритет и в каждый момент времени выполняется задача обработки прерывания с наивысшим приоритетом. Отсюда следует, что выполняемая задача обработки прерываний может прерываться с целью запуска более приоритетной задачи. Например, всем дискам может быть присвоен приоритет одного уровня, а всем устройствам вывода – другого уровня, тогда программа обработки прерывания от принтера может быть прервана запросом от контроллера магнитного диска, нуждающегося в обслуживании.

Практически такая приоритетность реализуется аппаратно-программным механизмом прерываний, в состав которого входит регистр прерываний и регистр маски прерываний (Рис. II.41). **Регистр прерываний** являет собой двоичный регистр, каждый разряд которого соответствует какому-либо прерыванию. Когда от аппаратуры прерываний приходит сигнал прерывания, соответствующий разряд регистра прерываний устанавливается в "1". После окончания обработки прерывания этот разряд устанавливается в "0".

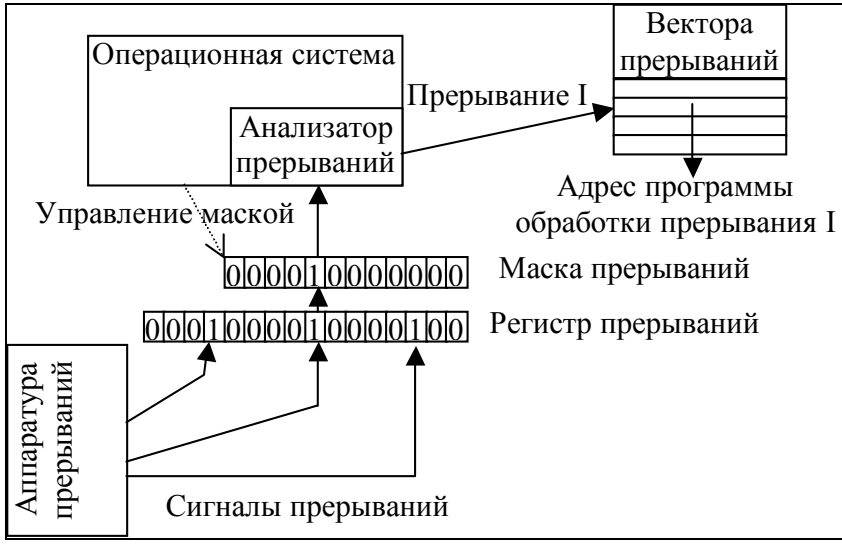


Рис. II.41. Маскирование и обработка прерываний.

В силу асинхронности прерываний, в общем случае, на регистре прерываний в каждый момент времени находится несколько единиц. И встает вопрос, какое прерывание обрабатывать непосредственно, а обработку которого можно отложить. Для сортировки поступивших прерываний используется двоичный регистр **маски прерывания**, каждый бит которого соответствует определенному биту регистра прерываний. Если бит регистра прерываний установлен в единицу и также в единицу установлен соответствующий бит маски, – сигнал прерывания проходит на анализатор, который запускает программу обработки этого прерывания.

Установкой в "1" и "0" регистров маски управляет операционная система. Образно говоря, регистр маски представляет собой своеобразное решето с управляемыми ячейками.

Обычно выделяются пять типов прерываний:

1. Прерывания от схем аппаратного контроля вычислительной машины возникает в случае неисправности оборудования.
2. Внешние прерывания дают возможность центральному процессору реагировать на сигналы от таких источников, как таймер, кнопка прерывания на клавиатуре, внешние датчики и т.д.
3. Прерывания от ввода–вывода позволяют центральному процессору реагировать на события, возникающие в контроллерах и устройствах ввода–вывода.
4. Программные прерывания происходят в тех случаях, когда в исполняемой программе возникает какой – либо особый случай (переполнение разрядной сетки, деление на нуль и т.д.).

5. Прерывания при обращении к операционной системе возникают, когда в исполняемой программе встречается команда перехода в "защищенный режим".

7.3. Слово состояния программы и цикл прерываний

При прерывании программы $\mathbf{P_I}$, и запуска вместо нее программы $\mathbf{P_J}$, для обеспечения возможности продолжения нормальной работы $\mathbf{P_I}$, необходимо запоминать данные, которые может испортить программа $\mathbf{P_J}$. К числу таких данных относятся: содержимое счетчика команд – адрес команды, содержимое регистра прерываний и регистра маски прерываний, флаги управления условными переходами, ключ защиты памяти. Все эти данные сводятся в одно **слово состояния программы** и в момент прерывания запоминаются в стековой памяти (Рис. II.42). В той же памяти запоминаются значения регистров. Дисциплина обслуживания стековой памяти позволяет продолжить задачу $\mathbf{P_I}$ с точки прерывания путем вызова из стека ее слова состояния.

Адрес команды	Длина команды	Регистр прерываний	Регистр маски	Флаги условий	Ключ защиты памяти

Рис. II.42. Формат слова состояния программы (PSW)

В каждый момент времени задача находится в одном из трех состояний (Рис. I.1). Состояние исполнения после прерывания сменяется состоянием обработки прерывания. После окончания обработки прерывания задача переходит в состояние готовности, ожидая, когда операционная система разрешит центральному процессору продолжить ее исполнение.

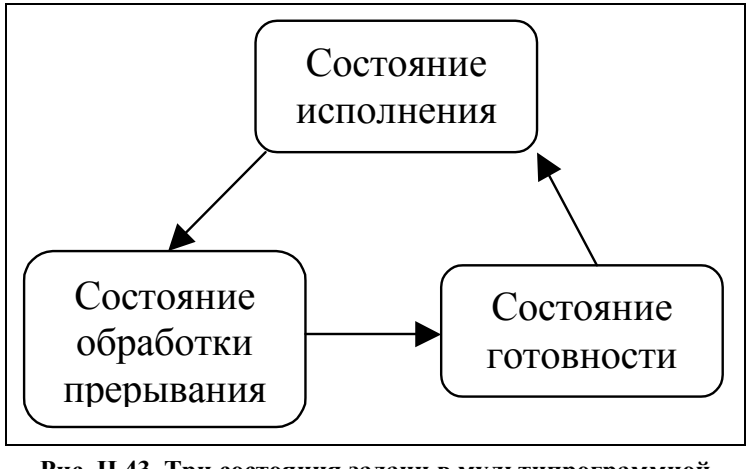


Рис. II.43. Три состояния задачи в мультипрограммной вычислительной системе

Цикл прерывания

1. Сигнал прерывания принимается блоком управления прерываниями.
2. Используемая в данный момент времени команда доводится до завершения в установленном порядке.
3. Состояние счетчика команд и регистров центрального процессора запоминается в стеке.
4. Реализуется переход на программу обработки соответствующего прерывания.
5. При завершении программы обработки прерывания управление передается либо на прерывающую задачу (если она готова продолжаться), либо другой программе, готовой к продолжению. При этом восстанавливаются значение счетчика команд и содержание регистров продолжаемой программы.

Мультипрограммная работа компьютера обеспечивается операционной системой. Основной составляющей операционной системы является управляющая программа - **супервизор**. Выполняя свои функции, супервизор контролирует состояние и управляет работой всех устройств: центрального процессора, оперативной памяти, внешних запоминающих устройств и устройств ввода–вывода.

Управление мультипрограммным режимом заключается в закреплении устройств и областей памяти за задачами, инициировании работы устройств, а также в освобождении устройств и областей памяти по окончании их использования программами.

Другой управляющей программой является **планировщик**, функции которого – ввод заданий и обеспечение их ресурсами с целью создания задач.

Не надо путать понятие однопрограммной или многопрограммной операционной системы и мультипрограммного режима работы вычислительной системы. Однопрограммная операционная система способна выполнять только одно задание пользователя. Напротив, многопрограммная операционная система способна одновременно выполнять несколько заданий, возможно, различных пользователей. Но даже однопрограммная операционная система использует мультипрограммный режим работы вычислительной системы: время центрального процессора делится между операционной системой и задачей пользователя, которая выполняется под управлением операционной системы.

7.4. Контрольные вопросы

1. Стратегия распределения времени центрального процессора между несколькими задачами.
2. Понятие прерывания и обработки прерывания.
3. Регистр прерываний и маска прерываний.
4. Слово состояния задачи и цикл прерывания.

8. Компьютеры параллельного действия

Существенная особенность архитектуры компьютеров, основанных на принципах фон Неймана, – последовательное исполнение команд программы. Борьба за быстродействие в этом случае заключается в разработке все более и более быстродействующих электронных схем. Эти схемы состоят из блоков, объединенных линиями связи, скорость передачи сигналов по которым существенным образом влияет на быстродействие системы в целом. Постоянная тенденция к микроминиатюризации уменьшает длину линий связи, т.е. время передачи сигналов, что также повышает быстродействие вычислительной системы.

В 80-х годах XX столетия появилось мнение, что теоретическое быстродействие последовательных компьютеров фон Неймана, определяемое скоростью передачи электронных сигналов по линиям связи (скорость света), практически достигнуто. Этот барьер назывался где-то в районе одного миллиона операций центрального процессора в секунду.

Значительные усилия конструкторов сконцентрировались на решении проблемы существенного увеличения быстродействия компьютеров за счет организации параллельной обработки данных. Здесь можно назвать два основных подхода.

Один подход рассматривает компьютеры параллельного действия как набор микросхем, которые соединены друг с другом определенным образом. При другом подходе возникает вопрос, какие процессы выполняются параллельно. Здесь имеется два крайних варианта.

Для решения задач на уровне крупных структурных единиц используются вычислительные системы со слабой связью – множество процессоров, которые взаимодействуют по схемам с низкой скоростью передачи данных.

Для решения задач на уровне мелких структурных единиц. используются системы с тесной связью – компоненты аппаратуры меньше, расположены ближе друг к другу и взаимодействуют через специальные коммуникационные сети с высокой пропускной способностью.

8.1. Параллелизм на уровне одной команды

Как уже неоднократно говорилось, основным барьером увеличения быстродействия компьютера является время доступа к оперативной памяти (как при выборке операнда, так и при чтении команды). Для снижения этого барьера в современных компьютерных архитектурах используется своеобразная **"подкачка" команд** в сверхбыстродействующий **буфер выборки с упреждением**.

Смысл этого усовершенствования в следующем: пока исполняется текущая команда (хранящаяся на регистре команд) из оперативной памяти в этот буфер – разновидность кэш-памяти, читаются команды, следующие за текущей. Таким образом, когда требуется следующая команда, она читается на регистр команд с минимальной затратой времени непосредственно из буфера. (Эта идея использовалась еще в нашей отечественной ЭВМ БЭСМ-6.)

Процесс выбора с упреждением позволяет разделить выполнение команды на два этапа: вызов и исполнение. Идея **конвейера** развивает эту идею: процесс выполнения команды подразделяется на несколько этапов, каждый из которых называется стадией и выполняется своим аппаратным блоком, причем, эти блоки функционируют параллельно. На Рис. II.44 вверху изображен конвейер пятистадийной команды и включающий в себя пять блоков аппаратуры.

Рис. II.44. Конвейерная обработка команд

На том же рисунке внизу приведена временная диаграмма работы конвейера команд. Порядковые номера команд приведены на пересечении строк и столбцов. Столбец **I** отображает команды, которые обрабатываются конвейером в момент - такт $\mathbf{t_J}$: чтение команды из памяти, дешифровка кода операции, чтение операндов из памяти, выполнение операции, запись результата. Строка **J** отображает номера команд, которые обрабатываются блоком $\mathbf{C_J}$ в каждый такт времени.

Из диаграммы видно, что блок $\mathbf{C_1}$ последовательно выбирает из памяти команды **1..9**, блок $\mathbf{C_2}$, со сдвигом на один такт, дешифрует коды операций команд **1..8**, блок $\mathbf{C_3}$, со сдвигом еще на один такт, читает операнды из памяти и т.д.

При конвейерной обработке суммарное время выполнения одной команды не уменьшается. Но за то же время выполняется **n** команд, где **n** – число стадий выполнения команды.

Сделав шаг в одну сторону и убедившись, что это хорошо, хочется сделать еще один шаг в ту же сторону. Точно так образуется **суперскалярная архитектура** компьютера, использующая несколько конвейеров, подробное описание которой выходит за рамки настоящего учебного пособия.

Параллелизм на уровне команд помогает в какой-то степени, но конвейеры и суперскалярная архитектура увеличивают скорость обработки данных всего лишь в 5 – 10 раз. Чтобы увеличить производительность на несколько десятичных порядков, нужно разрабатывать многопроцессорные системы.

8.2. Параллелизм на уровне данных

Достаточно часто при научно-технических расчетах необходимо выполнять операции над векторами и матрицами очень большого размера. Практически всегда такие данные можно разбить на сегменты, число которых определяется числом процессоров, и одновременно выполнять обработку всех сегментов данных. Такие процессоры имеют общую память и называются **мультипроцессорами**. Характерным примером является операция сложения двух векторов (Рис. II.45).

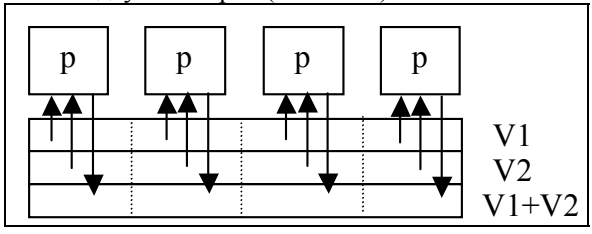


Рис. II.45. Сложение векторов, реализованное на четырех параллельных процессорах.

Для матричных операций также существуют блочные алгоритмы их выполнения.

Компьютеры, состоящие из нескольких процессоров и выполняющие одни и те же вычисления над разными сегментами одного набора данных в одно и то же время, называются **векторными компьютерами.**

8.3. Клонирование задачи в виде потока управления

Другая специфика обработки данных – запросы на выполнение работ, каждая из которых выполняется по одному и тому же алгоритму, поступают независимо. Такой подход оправдан для реализации быстрого обслуживания абонентов в системах коллективного пользования. (Системы резервирования на авиалиниях, независимые прогоны моделирующих программ с использованием нескольких наборов параметров и т.д.).

Поток управления – последовательность, в которой команды выполняются динамически, т.е. во время выполнения задачи. Последовательный поток управления изменяется командой вызова процедуры. Прерывание – это изменение в потоке управления, вызванное не самой программой, а возникшем событием в системе.

Напомним, что задача (процесс) **t(w)** определяется как динамическая единица работы **w**, которая образуется в момент начала работы и исчезает в момент ее окончания. Пусть **w -** некоторая вычислительная работа. Описанием работы служит программа **p(w)**.

При начале работы **w** по запросу одного пользователя запускается задача **t(w**), которая порождает свой клон в виде потока управления **s(p(w)), d1))** по обработке данных **d1**. При поступлении запроса на выполнение той же работы **w** с другим набором данных **d2** (другой пользователь) задача **t(w)** порождает еще один клон в виде потока управления **s(p(w),d2)**, выполняющего задачу **t(w)** для второго пользователя по обработке данных **d2** и т.д.

Допускается множество **клонов** – **потоков управления одной задачи**, и каждый клон выполняется на своем процессоре. Выигрыш во времени очевиден.

8.4. Параллелизм внутри одного потока управления

При выполнении потока управления на одном процессоре могут возникать ситуации, когда его дальнейшее продолжение невозможно до наступления вполне определенного события в вычислительной системе, например – до завершения операции ввода данных. В случае мультипрограммирования единственный процессор переключается на выполнение другой задачи. В случае многопроцессорной системы запуск новой задачи или другого потока управления далеко не всегда целесообразен, и для устранения простоя процессора в некоторых случаях можно использовать параллелизм на уровне одного потока управления.

Если программа выполнения работы такая, что в ней можно выделить несколько участков, которые можно выполнять независимо друг от друга (либо всегда, либо при некоторых условиях), то для каждого такого участка создается частичный поток управления – **нить** (Рис. II.46). Все нити выполняются на одном процессоре в режиме традиционного мультипрограммирования.

Очевидно, что возникает сложная проблема синхронизации выполнения множества потоков и множества нитей, которую решает операционная система.

Еще раз подчеркнем, что в рассматриваемом случае потоки существуют независимо и данными не обмениваются.

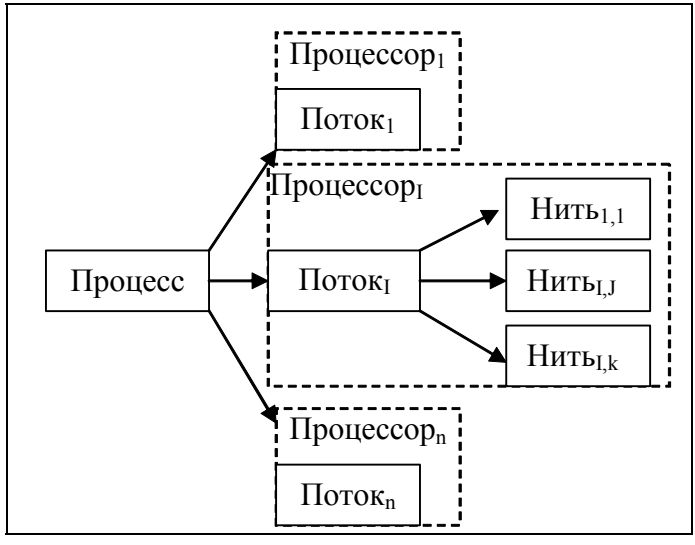


Рис. II.46. Процесс порождает потоки, каждый поток порождает нити

8.5. Параллелизм на уровне процессов

Во многих случаях одну суперработу по обработке данных целесообразно представить в виде нескольких слабо связанных по данным работ, каждая из которых может выполняться на своем процессоре, обмениваясь данными с другими работами. Обмен данными между процессорами происходит по сети межсоединений, и вычислительная система превращается в **мультикомпьютер** (Рис. II.47)**.**

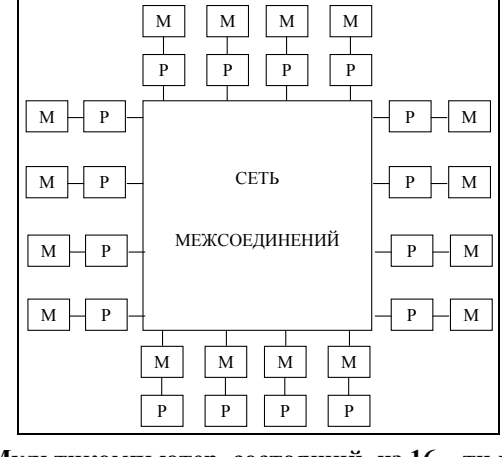


Рис. II.47. Мультикомпьютер, состоящий из 16 – ти процессоров, каждый из которых имеет собственную память

Скорость обмена данными в сети межсоединений относительно небольшая, в силу чего частота обмена данными между процессами не должна быть чрезмерной (Слабая связанность процессов).

Ключевое отличие мультипроцессора от мультикомпьютера состоит в том, что каждый процессор в мультикомпьютере имеет собственную память, к которой никакой другой процессор не может обращаться непосредственно. Поскольку процессоры в мультикомпьютере не могут взаимодействовать друг с другом путем чтения из общей памяти, они посылают друг другу сообщения, используя сеть межсоединений. При этом основной проблемой становится правильное разделение данных и разумное их размещение, тогда как в мультипроцессоре размещение сегментов данных не влияет на правильность выполнения задачи.

Таким образом, мультикомпьютер программировать гораздо сложнее, чем мультипроцессор. Зачем же создавать такие мультикомпьютеры? Дело в том, что гораздо проще множество процессоров с локальными блоками памяти соединить сетью межсоединений, чем реализовать общую память, разделяемую множеством процессоров.

8.6. Контрольные вопросы

1. Параллелизм как принципиальное средство увеличения быстродействия вычислительной системы.
2. Конвейерная обработка команд.
3. Параллелизм в случае векторных вычислений.
4. Понятие потока управления и его роль в организации параллельных вычислений.
5. Нити и параллелизм внутри одного потока управления.
6. Параллелизм на уровне процессов.

9. Автоматизация программирования

Аппаратура компьютера проектируется из соображений оптимальности технических решений и эффективности обработки данных. В силу этого, как самые первые, так и современные компьютеры, требуют представления данных и команд программы в виде двоичных кодов, т.е. последовательностей нулей и единиц. Очевидно, что программирование в двоичных кодах является весьма кропотливым и малоэффективным занятием. В силу этого, развитие вычислительных технологий, практически с самого первого дня изобретения компьютеров, идет по двум направлениям: совершенствование аппаратуры вычислительных устройств; совершенствование средств автоматизации разработки и исполнения программ.

Представление данных и команд в памяти компьютера одинаково (двоичная последовательность), поэтому иногда мы будем говорить об информации, с которой имеет дело компьютер, подразумевая под этим как данные, так и программу.

9.1. От двоичных кодов к языкам символьного кодирования

9.1.1. Проблема ввода-вывода данных

В конце концов, способ представления данных в оперативной памяти компьютера программиста не волнует. Но ему не безразлично, в каком виде он должен вводить исходные данные и получать результаты вычислений.

Нашлись "реформаторы", предложившие перевести школьное обучение на двоичную систему представления чисел. Их аргументация была проста и привлекательна:

- с точки зрения теории вычислений совершенно безразлично, в какой системе кодировать числа;
- мы, старики, худо - бедно перемучаемся, зато наши дети будут с компьютером "на ты".

Однако, если подумать, то эту проблему можно решить без революции в образовании. Надо просто учесть, что перевод чисел из десятичной системы в двоичную систему и обратно реализуется по формальным алгоритмам. Следовательно, можно сделать программы перевода из десятичной системы в двоичную систему и обратно, которые будет выполнять компьютер и которые позволят вводить и выводить числа в десятичной системе счисления.

Уже здесь появляется специфика двухуровневого представления данных и программ в системе человек - компьютер:

- на внешнем уровне необходимо представлять данные и программы в виде, максимально удобном для человеческого восприятия (графика, тексты, таблицы и т.д.);

- такое представление данных и программ, как правило, неэффективно с точки зрения ее обработки компьютером;
- на внутреннем уровне данные и программы представляются в виде эффективном для их обработки (двоичные коды, списковые структуры, структуры данных);
- как следствие, необходимы специфические программы – трансляторы , реализующие автоматическое преобразование представления данных и программ с человеческого уровня на машинный уровень и обратно.

Замечание. Программы трансляторы рассматривают и обрабатывают программы вычислений как данные.

9.1.2. Ассемблеры

Здесь и далее, говоря о языке, мы будем понимать **процедурный язык программирования**, т.е. формальную знаковую систему, позволяющую описывать последовательность шагов любого алгоритма обработки данных.

Для компьютера каждого типа на стадии разработки его архитектуры фиксируется система машинных (двоичных) двоичных команд. Каждая такая команда представляет собой определенным образом форматированную двоичную последовательность – предписание для компьютера на выполнение вполне определенного действия. Система команд вычислительной машины называется также **машинным (двоичным) языком программирования**. Машинная(двоичная) программа состоит из последовательности машинных команд, и только такая машинная программа непосредственно интерпретируется, т.е. исполняется аппаратурой компьютера.

"Ручное" кодирование машинной программы чрезвычайно трудоемко и связано с появлением большого числа потенциальных ошибок. Поэтому проблема создания средств автоматизации программирования в виде языков более высокого уровня, чем машинный язык, возникла на первых же шагах развития вычислительной техники.

В самом простейшем варианте решение было очевидным - обозначить поля команды символами, понятными человеку, и получить в результате предписание на выполнение действия в виде символьной строки.

Например, команду сложения можно обозначить следующим образом:

ADD 1001B,1010B,1011B; (II.15)

где **B** - признак двоичного числа.

Такая команда предписывает вычислительной машине следующие действия: прочитать содержимое ячеек оперативной памяти с двоичными адресами 1001 и 1010, произвести над их содержимым операцию арифметического сложения и записать результат в ячейку оперативной памяти с двоичным адресом 1011.

Для более короткой записи команды адреса лучше представлять в шестнадцатеричном коде:

ADD 9H,AH,BH; (II.16)

где **H** - признак шестнадцатеричного кода.

Каждая символьная строка, обозначающая машинную команду, называется **мнемокомандой**, а совокупность таких команд образует **язык автокод** (язык автоматического кодирования программы).

Дальнейшим развитием языка автокод является введение в его описательные возможности средств управления полуавтоматическим распределением оперативной памяти. Суть такого распределения памяти состоит в следующем:

- ячейки и массивы ячеек можно обозначать символьными именами (адресами);
- в адресных полях команды можно использовать эти символьные имена (адреса);
- имеется псевдокоманда MEM, позволяющая установить соответствие символьного адреса и физического адреса;
- имеется псевдокоманда CONST, позволяющая присвоить ячейке памяти с заданным символьным адресом конкретное символьное значение.

Таким образом, предписание ADD 9H,AH,BH; (II.16) можно записать в следующем виде:

MEM Маша 9H; /*симв. адрес */ (II.17)

MEM Коля AH; /*симв. адрес */

MEM Любовь BH; /*симв. адрес */

CONST Маша,12; /*ввод операнда*/

CONST Коля,15; /*ввод операнда*/

ADD Маша, Коля, Любовь. /*действие*/

Толчком к дальнейшему развитию языка автокод послужило увеличение размеров программ. Большую программу можно быстрее и качественнее сконструировать, разбив ее на некоторое количество более или менее независимых сегментов. Это дает возможность разрабатывать и отлаживать сложную программу "по частям", а также использовать при разработке новой программы уже апробированные сегменты ранее разработанных программ. Разумеется, между сегментами допускаются формальные перекрестные ссылки, фактический вид которых не известен во время написания секций.

Более развитый язык автокод, кроме множества мнемокоманд, содержит также **псевдокоманды**, которые обеспечивают полуавтоматическое распределение памяти, связывание в единую программу совокупности сегментов, ввод исходных данных и другие полезные действия. Псевдокоманды не имеют аналога в системе машинных команд и реализуются программным путем в ассемблирующей системе (см. ниже).

Таким образом, язык автокод позволяет использовать при программировании все возможности машинного языка. Его основу составляют термины мнемонических обозначений операций и их операндов (ядро языка), а также других выполняемых операций (псевдокоманды). Это позволяет программисту ссылаться на внутренние регистры машины и адреса оперативной памяти, используя их символьное обозначение. Существенно, что автокоды позволяют описывать различные формы представления данных, обеспечивают автономное преобразование их, позволяют независимо транслировать отдельные программные модули и соединять их в единую программу, а также управлять распределением адресов памяти и документацией программы.

Очевидно, что для каждого семейства компьютеров существует свой уникальный язык автокод, определяемый системой машинных команд этого семейства.

Программа, записанная на языке автокод, более понятна человеку, чем машинная программа, но компьютер такую программу интерпретировать (выполнять) не может. Поэтому нужен переводчик, преобразующий (транслирующий) текст автокодной программы в текст машиной программы. На первых порах в качестве таких переводчиков выступали девочки - лаборанты, которые знали совокупность формальных правил (алгоритм) преобразования символьных текстов в двоичные. Но достаточно быстро были созданы **ассемблеры** (программные ассемблирующие системы), которые не менее успешно справлялись с этой рутинной работой.

Прежде всего, ассемблер реализует функции трансляции (перевода) с языка автокод на язык машинных команд. В качестве входного набора данных для ассемблера выступает программа, записанная на языке автокод, которая переводится в перемещаемую программу на машинном языке. Обычно таким языком является язык загрузчика. Различные секции автокодной программы транслируются независимо друг от друга, в результате чего получается множество объектных модулей, каждый из которых представлен на языке загрузчика.

После трансляции всех модулей работает программа загрузки, которая связывает множество объектных модулей в готовую машинную программу, загружает ее в оперативную память и запускает на выполнение.

Язык автокод, включающий в себя средства сборки программ из секций, принято называть также **языком ассемблера**.

Подводя итог, можно сказать что ассемблером (англ. assembler, от assemble - собирать, монтировать) называется компонента системного программного обеспечения, переводящая программу, написанную на автокоде (символьный модуль) в машинную программу (загрузочный модуль). Программа на автокоде представляется в командах, отражающих архитектуру конкретного типа компьютера. Помимо трансляции (перевода) важнейшей функцией ассемблера является сборка или связывание многих объектных модулей в единый загрузочный модуль.

Замечание. Часто можно слышать от того или иного программиста, что он "программирует на ассемблере". Более правильно говорить о программировании на автокоде или на языке ассемблера, тогда как ассемблером называется программа, "понимающая" этот язык.

9.1.3. Программы - загрузчики

Назначение загрузчика определяется его именем. Он загружает программу в оперативную память машины. Различаются три типа загрузчиков.: двоичные, настраивающие и связывающие.

Двоичный загрузчик загружает машинную программу, записанную в двоичном виде относительно фиксированных адресов, с какого-либо внешнего устройства.

Настраивающий загрузчик получает программу в перемещаемой двоичной форме (записанную в относительных адресах) вместе с информацией о настройке ее по месту и размещает ее в памяти, предварительно настроив относительные адреса.

Связывающий загрузчик получает набор объектных модулей, записанных в двоично-символьной форме, и загружает их в оперативную память как единую программу, заполнив перекрестные ссылки между ними и настроив их по месту.

Связывание объектных модулей в готовую программу может быть выделена в отдельную операцию, которая выполняется программой – редактор связей. Очевидно, в этом случае загрузчик работает после редактора связей и реализует только размещение программы в оперативной памяти.

9.1.4. Макросредства

Часто случается, что одна и та же группа команд с небольшими изменениями встречается в нескольких местах программы. Для того, чтобы при составлении программы многократно не переписывать один и тот же (или, почти один и тот же) текст, был изобретен аппарат **макросов** – макрокоманд, определяемых пользователем.

Макроопределение служит для конструирования пользователем своей собственной макрокоманды, более крупной, по сравнению с машинной. Макроопределение состоит из заголовка и тела.

Заголовок макроопределения представляет собой предложение вида:

<имя макро>:Макро,<список формальных параметров>.

Макро - это псевдокоманда ассемблера, сообщающая ему, что данное предложение является заголовком макроопределения. Имя макро используется для последующих ссылок на это макроопределение, а список формальных параметров определяет каналы связи макроопределения с основной программой.

Тело макроопределения, которое следует за заголовком, представляет собой фрагмент обычной автокодной программы - именно он и определяет существо вводимой макрооперации. Следом за телом следует псевдокоманда ***Конм*** конца макроопределения.

Приведем пример макроопределения, которое вводит в употребление команду нахождения наибольшего из двух значений: z=max(x,y):

Max2: ***макро*****, X,Y,Z; заголовок макроопределения;** **(II.18)**

Пересылка X,Z; тело1;
Вычитание X,Y; тело2;
Передача_управления КОНМ; тело3;
Пересылка Y,Z; тело4;

Конм; признак конца макроопределения

В теле макрокоманды программируется априорная пересылка в ячейку результата Z содержимого ячейки X. Затем производится сравнение содержимого X и содержимого Y, и если Y>X, то реализуется пересылка содержимого ячейки Y в ячейку результата Z. Существенно, что обработка данных в теле макроопределения определяется вне зависимости от его использования относительно формальных параметров.

Теперь программист может использовать это макро в своей программе, указав его имя и список фактических параметров (макровызов).

Так, с использованием макроопределения (II.18) для вычисления максимума двух чисел можно записать такую макропрограмму:

.....
L:MAX2,A,B,C; **(II.19)**
.....
M: MAX2, U,V,W;
.....

Предполагается, что перед запуском ассемблера будет использоваться **макрогенератор**, который в процессе просмотра макропрограммы каждый раз, когда встретит макровызов, будет осуществлять **макроподстановку**.

Т.е. в процессе макрогенерации каждый макровызов будет заменен телом соответствующего макроопределения. Причем, имена формальных параметров будут заменены именами фактических параметров.

В частности, приведенный выше фрагмент макропрограммы **L:MAX2,A,B,C;** **(II.19)**

) после макроподстановки будет иметь вид обычной автокодной программы:

```
.....                                    (II.20)
L:Пересылка A,C;
  Вычитание A,B;
  Перед.упр. L1;
  Пересылка B,C;
  L1:.....
   M:Пересылка U,W;
  Вычитание U,V;
  Перед.упр. M1;
  Пересылка V,W;
M1:.....
```

После работы макрогенератора запускается обычный ассемблер, который формирует объектный модуль.

Возможна другая схема обработки макропрограммы, когда макроассемблер совмещает в себе функции макрогенератора и обычного ассемблера.

Таким образом, аппарат макросов позволяет пользователю расширить базовый язык автокодирования за счет введения проблемно - ориентированных макрокоманд и, тем самым, сделать его более удобным для применения.

Примечание. В современных ассемблерах используется также механизм подпрограмм, но о нем подробнее в следующем разделе.

9.2. Языки программирования высокого уровня

Создание машино-ориентированных языков программирования - автокодов позволило существенным образом повысить эффективность программирования, избегая процедуры полностью ручного распределения оперативной памяти, а также необходимости записывать программы в виде последовательностей нулей и единиц.

Но, как уже говорилось, машино-ориентированный язык (автокод) создается для каждого типа компьютера и для его использования необходимо знать архитектуру и специфику программирования именно этого типа компьютера. Более того, в силу элементарности машинных команд программирование даже простого математического вычисления сводилось к выписыванию многих автокодных команд.

Можно сказать, что между потенциальными проблемными пользователями компьютера, которые привыкли оперировать математическими формулами (физиками, биологами, экономистами и т.д. и т.п.), существовал весьма высокий языковый барьер.

Более того, программа, записанная на автокоде для одного типа машин, не могла выполняться на другом типе машин. Все это существенным образом затрудняло внедрение вычислительной техники в науку и технику, не говоря уже о прикладных сферах человеческой деятельности.

Дальнейшее развитие средств автоматизации программирования связано с созданием **машинонезависимых языков программирования высокого уровня**. Первым таким языком был язык ФОРТРАН -входной язык формульного транслятора, созданного специалистами фирмы IBM. Как следует из названия, его создатели, не задумываясь об очень высоких теориях, поставили своей целью расширить язык автокода до языка, позволяющего достаточно просто программировать формульные вычисления, и также сделать этот язык машинонезависимым.

Однако, идеи, заложенные в ФОРТРАН, оказались настолько плодотворными, что, за редкими исключениями, воплотились во всех дальнейших высокоуровневых языках программирования. Перечислим эти основные идеи.

9.2.1. Переменная и тип переменной

Прежде всего, было введено понятие **переменной** и понятие типа переменной. С одной стороны, переменная - это символьное обозначение области оперативной памяти, как в автокоде. Но "привязывание" к каждой переменной определенного типа означает определение правила интерпретации текущего значения и области допустимых значений этой переменной. Это позволяет решить, по крайней мере, две проблемы автоматизации программирования:

- полностью автоматическое распределение памяти для хранения значений переменных;
- автоматический контроль корректности использования операндов в операциях.

В первом случае, декларация типа переменной дает транслятору достаточно информации о размере области памяти для хранения значения.

Во втором случае, знание типа переменной позволяет избежать таких ошибок, как сложение числа и символьной строки.

Правда, на первых порах введение типов в Фортране преследовало более скромные и практические задачи - экономию оперативной памяти. В первой версии языка допускалось всего два типа переменных: целые и вещественные. Был введен даже специальный оператор эквивалентности, позволявший на различных этапах выполнения программы в одной и той же области памяти хранить значения различных переменных.

Специального оператора декларации типа переменной не было. Было принято, казавшееся сначала простым, но в дальнейшем приводившее к многим неприятностям решение: если имя переменной начинается с символов I,J,K,L,M,N - переменная целого типа, в противном случае - переменная вещественного типа.

Достаточно быстро выявились неудобство и недостаточность такого способа типизации:

- возникла необходимость использования других типов, прежде всего логического;
- определение типа по первому символу противоречило одному из основных правил программирования - назначать переменным содержательные, осмысленные имена.

Поэтому с целью описания типа переменной были введены специальные предложения - декларации типа переменной (integer, real, logical и т.д.).

Особо следует отметить структурную переменную типа массив, которая также впервые определена в Фортране.

Введение типизации переменных позволило реализовать полностью автоматическое управление распределением оперативной памяти для хранения их значений. Чуть позже появилось разделение языков программирования на слабо типизированные языки и сильно типизированные языки. Если первые допускают определение типа некоторых переменных по умолчанию (Фортран), то вторые требуют обязательной декларации типа переменной. Обязательность определения переменных в сильно типизированных языках контролируется транслятором.

Появилось также подразделение языков программирования на статические и динамические языки.

Статический язык программирования (Паскаль) предполагает автоматическое распределение памяти во время трансляции до выполнения программы. Это касается, прежде всего, массивов. Трансляторы для таких языков устроены достаточно просто, но эффективность получаемой двоичной программы (время работы, используемые ресурсы памяти) близка к автокодной программе, реализующей тот же алгоритм. В основном по этой причине долгое время не могли "списать" такой древний язык программирования, как Фортран, эффективность программ которого была практически равна программам ручного изготовления.

Динамический язык программирования (Бейсик) допускает использование динамических массивов, размер которых определяется и меняется в процессе выполнения программы. Более того, переменные могут создаваться и уничтожаться во время ее исполнения. Удобство для программиста очевидное, но сложность трансляторов и малоэффективность двоичных программ не способствовало подавляющему распространению динамических языков.

Как всегда истина - в золотой середине. Язык строится как статический (ПАСКАЛЬ, СИ), но в него включается возможность динамического управления квантами оперативной памяти, что позволяет в полуавтоматическом режиме создавать динамические массивы.

9.2.2. Операторы обработки данных

Операторы обработки данных подразделяют на три класса: операторы ввода данных, операторы вывода данных, операторы присваивания.

Оператор ввода позволяет устанавливать извне (с помощью устройств ввода) значение входной переменной. Аналогично, оператор вывода позволяет выводить на внешнее устройство значение выходной переменной. Тут возникает проблема преобразования внешнего представления единицы данных во внутреннее представление и обратно. В Фортране впервые были использованы своеобразные шаблоны - форматы, которые позволяли программировать эти преобразования. Наряду с другими способами, форматный ввод-вывод сохранился во многих современных языках.

Оператор присваивания можно назвать основным оператором языка программирования высокого уровня и утверждать, что остальные операторы только "обслуживают" его. На самом деле, только оператор присваивания позволяет устанавливать значение переменной левой части в зависимости от значений переменных правой части. По сути дела, ***оператор присваивания является императивом (предписанием) для вычисления функции F, заданной арифметическим выражением F'***:

$$y = F'(x_1, x_2, \ldots, x_n) \qquad \text{(II.21)}$$

Благодаря оператору присваивания, проблемный специалист может записывать предписания на вычисление математических формул в почти привычном для него виде.

Замечание. Не слишком удачным оказался выбор знака равенства для обозначения операции присваивания. Например, запись x = x+1 неопытный программист интерпретирует как: " икс равняется икс плюс единица", что с математической точки зрения звучит совершенно нелепо. В некоторых языках программирования (Паскаль) используется специальный знак присваивания. Например, запись x:=x+1, интерпретируется однозначно: "новое значение x устанавливается равным старому значению x увеличенному на единицу". К сожалению, не все современные языки используют в качестве знака операции присваивания символ отличный от знака равенства.

Можно говорить, что оператор присваивания обозначает последовательность двоичных команд (или мнемокоманд автокода), исполнение которых обеспечивает вычисление функции **F** в заданной точке. Например, оператор присваивания **y = a+b*c-d** обозначает следующую последовательность трехадресных команд автокода:

умножение a,b,u; сложение a,u,v; вычитание v,d,y.

Точно так же, операторы ввода-вывода языка программирования высокого уровня обозначают последовательности двоичных команд, реализующие задаваемые действия по вводу-выводу.

9.2.3. Операторы безусловного и условного перехода

Программу, представляющую собой последовательность операторов обработки данных и только обработки данных, принято называть линейной программой. Однако, линейных программ практически не бывает. Дело в том, что в любой программе конкретный ход вычислительного процесса (порядок исполнения операторов обработки данных) зависит от конкретного набора исходных данных. Представьте, например, программу вычисления корней квадратного уравнения. Кроме того, сокращенная запись периодической последовательности вычислений в виде цикла требует также средств изменения линейного порядка выполнения команд программы.

Для изменения порядка выполнения операторов присваивания был введен **оператор безусловного перехода**:

GO TO N.

Предполагалось, что некоторые операторы могут быть пронумерованы (иметь метки) и **N** - одна из меток.

Оператор условного перехода появился в ФОРТРАНЕ, как бы сейчас сказали, в укороченной форме:

IF (<условие>) **THEN** <альтернатива да>,

где <альтернатива да> - произвольный оператор, чаще всего **GO TO**.

В качестве условия используется логическое выражение, которое представляет собой специальную запись логической (булевской) функции.

По сути дела, оператор условного перехода также обозначает некоторую последовательность двоичных команд, реализующих различные вычисления в зависимости от значения логического выражения.

Достоинством такого решения была возможность задавать достаточно сложные условия перехода по сравнению с условием равенства - неравенства нулю для машинной команды условного перехода.

Недостатков укороченного оператора условного перехода было, по крайней мере, два. Во-первых, альтернатива состояла только из одного оператора. Этот недостаток был исправлен уже в АЛГОЛе, где определили составной оператор как любую последовательность операторов, заключенную в операторные скобки:

IF(условие) **THEN begin** S_1;…; S_n **end**;

Второй недостаток связан с несимметрией оператора условного перехода, и этот недостаток был устранен введением в последующих языках программирования "структурного" оператора условного перехода:

IF(условие) **THEN** <альтернатива да>**ELSE**<альтернатива нет>;

Также в последующих языках, кроме условного оператора двоичного выбора, был введен оператор множественного выбора **case**. Последний является обозначением системы вложенных операторов **IF**, определяющих множественный выбор.

9.2.4. Операторы цикла

Практически в любой программе существуют циклические конструкции. Для их реализации достаточно операторов присваивания и операторов перехода. Например, цикл, реализующий повторение **M** раз операторов **S1,...,Sk**, может быть записан в виде:

```
I=1;                                    (II.22)
N1:IF(I>M) THEH GO TO N;
S1;
....
Sk;
I=I+1;
GO TO N1;
N: .....
```

В языке Фортран был введен **оператор цикла**, который, по сути дела, являлся условным обозначением вышеприведенной циклической конструкции:

```
DO N, I=1,M;                            (II.23)
 S1;
.....
Sk;
N: CONTINUE;
```

Так как в Фортране не было понятия составного оператора, тело цикла ограничивалось специальным оператором **CONTINUE**, на который имелась ссылка в заголовке цикла.

Использование такого цикла упрощало программирование обработки последовательности данных без запоминания и обработки последовательности данных с запоминанием (использование переменной типа массив). Как выяснилось, такие часто встречающиеся циклы не являются единственными. В дальнейшем, в языках структурного программирования был определен универсальный цикл **wile** (цикл ПОКА), который в качестве условия окончания использует логическое выражение. Была даже попытка сделать его единственным возможным циклом. Но практика заставила вернуть в языки программирования фортрановский цикл под названием **For** и добавить цикл **repeat** (цикл ДО).

9.2.5. Подпрограммы

В разделе "Макрокоманды" рассматривался аппарат макросов, обеспечивающий ассемблирование (сборку) программы с использованием заранее изготовленных шаблонов - макроопределений. Это уже были зачатки некоторой технологии программирования, позволяющей:

- не переписывать многократно повторяющиеся в программе последовательности команд, заменяя их макрокомандами;

- собирать большую программу из подпрограмм, каждая из которых может разрабатываться и отлаживаться независимо от других подпрограмм.

Последнее свойство макросов оказалось весьма полезным, когда появилась необходимость разрабатывать большие программы, и появилось понимание того, что один человек за приемлемое время такую программу разработать не в состоянии. Необходима технология программирования, обеспечивающая коллективную (корпоративную) разработку больших программ.

Наряду с достоинствами макрос, как подпрограмма, имел существенный недостаток: макроподстановка конкретного макроопределения производилась столько раз, сколько раз встречалась в программе соответствующая ему макрокоманда (статическая подстановка). Это приводило к большим размерам загрузочного модуля, требующего для своего размещения большого объема оперативной памяти. Особенно это было неприемлемо для компьютеров первых поколений с небольшим ресурсом оперативной памяти.

Поэтому теоретики и практики программирования изобрели механизм **подпрограммы**. Вместо статической подстановки, размножающей текст макроопределения, было предложено использовать динамическую подстановку. Суть ее состоит в том, что подпрограмма существует в единственном экземпляре и при каждом к ней обращении в тексте основной программы происходит динамическая подстановка. При этом управление передается на начало подпрограммы и обеспечивается возврат в основную программу после окончания работы подпрограммы.

Концепция подпрограммы (подпрограммы-процедуры и подпрограммы-функции) достаточно хорошо известна и здесь подробно не рассматривается.

Отметим только, что подпрограммы в языках программирования высокого уровня бывают двух типов. **Подпрограммы функции** предоставляют для программиста средства конструирования собственных стандартных функций, которые затем можно использовать в выражениях (арифметических и логических). Аналогично, **подпрограммы процедуры** являются средством конструирования векторных операторов присваивания.

Подпрограммы экономят оперативную память, но увеличивают время работы подпрограммы за счет дополнительных операций обращения и возврата. При работе на современных компьютерах эти факторы играют незначительную роль. Поэтому в современном программировании используется смешанная тактика использования макросов и подпрограмм, диктуемая эффективностью процесса программирования. Например, используемый в языке программирования Си оператор #Define является аналогом введенного выше оператора ***Макро***

9.3. Трансляция программ

Мы уже говорили, что представление данных и команд в компьютере целесообразно делать многоуровневым: с одной стороны, - внешнее представление, удобное для человека, с другой стороны, - внутреннее представление, эффективное для обработки компьютером. Для чисел- это очевидная дилемма десятичной и двоичной системы кодирования, для языков общения человека с компьютером - иерархия языков программирования.

Мы также знаем, что компьютер может исполнять программу, представленную в двоичных кодах, загруженную в оперативную память и настроенную по месту загрузки. Такие программы вручную не изготавливаются. Они получаются в результате работы системных программ - трансляторов из текстов программ, записанных на символьных языках различного уровня.

Транслятором принято называть системную программу, преобразующую текст программы пользователя в семантически тождественный текст, записанный на языке более низкого уровня.

Сказанное выше можно представить в виде следующей диаграммы (Рис. II.48).

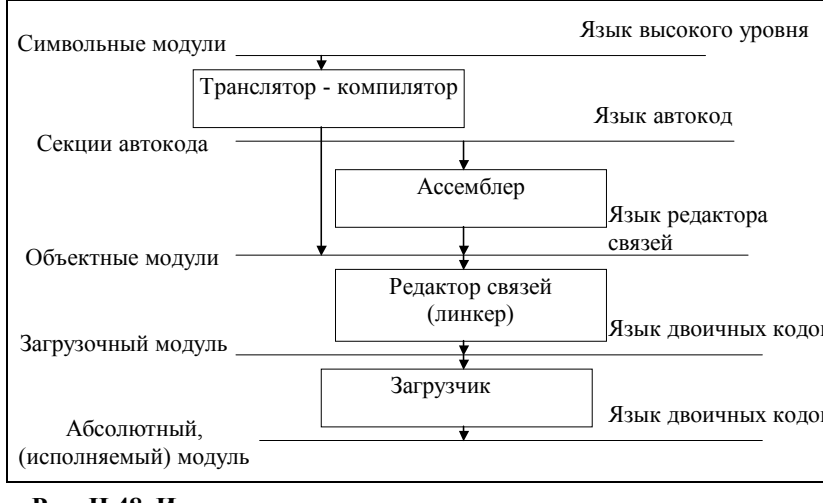


Рис. II.48. Иерархия языков программирования и трансляторов

Начнем с автокода - ассемблера. Каждая секция программы, записанная на языке автокод, преобразуется транслятором ассемблирующего типа в объектный модуль, записанный на двоично-символьном языке, который "понимает" редактор связей. Этот редактор связей из нескольких объектных модулей формирует единый загрузочный модуль - перемещаемую двоичную программу.

Эта программа записана в относительных адресах, как бы предназначенная для загрузки с нулевого адреса оперативной памяти. Загрузчик находит свободное место в оперативной памяти, в которое размещает загрузочный модуль, настраивая его по месторасположению. В результате получается абсолютный модуль, который может выполняться.

Если программа состоит из модулей, написанных на языке высокого уровня, вместо ассемблера она обрабатывается компилятором. В результате получаются объектные модули, которые преобразуются в загруженную двоичную программу вышеописанным образом.

Обратим внимание, что объектный модули, полученные из символьных модулей языка высокого уровня и символьных секций автокода, записаны на одном и том же языке редактора связей, т.е. неотличимы друг от друга. Это важное свойство позволяет при программировании на языке высокого уровня использовать ассемблерные вставки. Редактора связей не интересует предыстория объектных модулей.

Технологическая схема компиляции изображена на (Рис. II.49).

Рис. II.49. Технологическая схема компиляции

Три компьютера (K1, K2, K3) выполняют системные программы компилятора, редактора связей и загрузчика.

Чаще всего, в качестве этих компьютеров используется один и тот же реальный компьютер. В результате получается абсолютный модуль программы пользователя, который исполняется на компьютере K0, реализуя преобразование исходных данных **d** в результирующие данные **r**.

Трансляторы подразделяются на два класса: компиляторы и интерпретаторы. Выше мы описали работу транслятора компилирующего типа- **компилятора**.

Транслятор интерпретирующего типа - **интерпретатор**, по сути дела, представляет собой виртуальную машину "понимающую" язык программирования высокого уровня (Рис. II.50). Такая виртуальная машина состоит из аппаратуры компьютера и транслятора интерпретирующего типа, расширяющего возможности аппаратуры до "понимания" языка программирования высокого уровня.

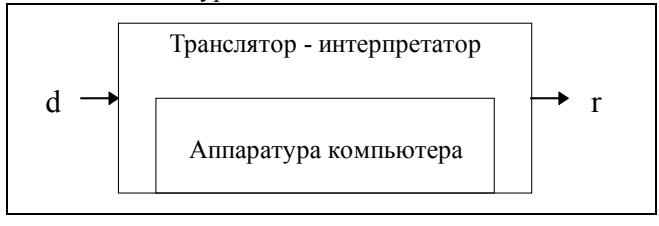


Рис. II.50. Транслятор - интерпретатор как виртуальная вычислительная машина

При разработке небольших программ интерпретатор имеет преимущество перед компилятором в смысле простоты использования. Основные его недостатки, проявляющиеся при конструировании больших программ: программа работает дольше, не реализуется сборка больших программ из модулей, разработанных различными программистами.

Компилятор свободен от названных выше недостатков интерпретатора. Также преимуществом компиляции является возможность многократного исполнения загрузочного модуля без повторной его трансляции. Последнее соображение особо существенно в случае больших программ, для которых время, затрачиваемое на трансляцию и редактирование, является существенным.

9.4. Преобразование кодов при работе с компьютером

Еще раз остановимся на многоуровневом представлении данных и команд в компьютере (Рис. II.51).

На внешнем уровне, для представления информации применяются коды, которые мы называем графическими и которые используют в качестве своих элементов графические образы в виде букв, цифр, символов, изображений

Графические коды можно разбить на три группы: **P**- коды программ, **L** - код символьных строк, **N** - коды чисел.

Устройства и процедуры ввода преобразуют графические коды в двоично-символьные коды и размещают последние в оперативной памяти. Под двоично-символьным кодом мы понимаем двоичную последовательность, состоящую из двоичных кодов графических образов (букв, цифр, символов, изображений).

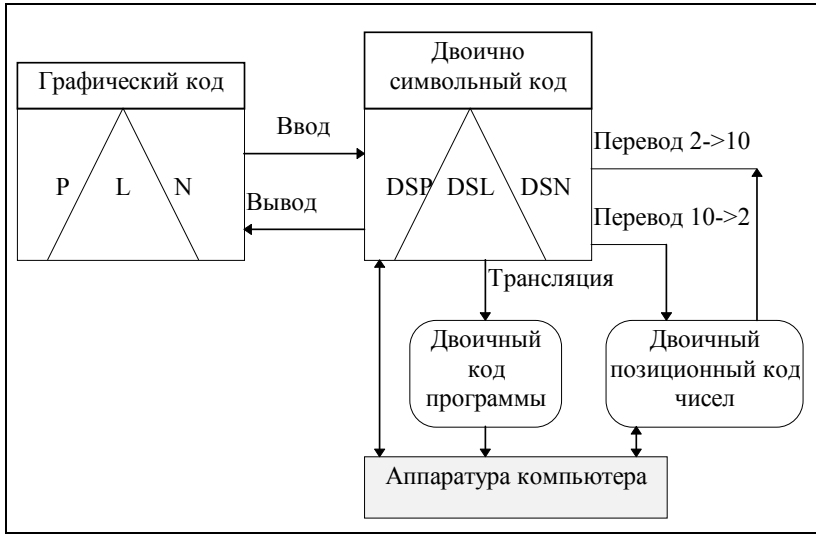


Рис. II.51. Преобразование данных в компьютере

Таким образом, в оперативной памяти располагаются двоично-символьные коды: DSP - программ, DSL - символьных строк, DSN - чисел.

Двоично-символьные коды символьных строк могут непосредственно обрабатываться (операции сравнения и конкатенации строк). Для арифметических операций над числами необходимо их представление в позиционной двоичной системе счисления. Поэтому двоично-символьные коды чисел непосредственно обрабатываться не могут. Их надо преобразовать в двоичные позиционные коды. Для такого преобразования используется процедура перевода чисел из десятичной системы счисления в двоичную систему.

Точно также представленную в двоично-символьном виде программу компьютер "не понимает" и выполнить не может. Необходим перевод программы из двоичного - символьного представления в двоичное представление. Такой перевод осуществляется программой транслятором.

После получения численных результатов они должны быть представлены в виде двоично-символьных кодов, что реализуется процедурой перевода из двоичной системы счисления в десятичную систему.

Двоично-символьные коды строк и чисел выводятся из оперативной памяти аппаратурой и процедурами вывода.

На Рис. II.52 приведены примеры различных кодов данных:

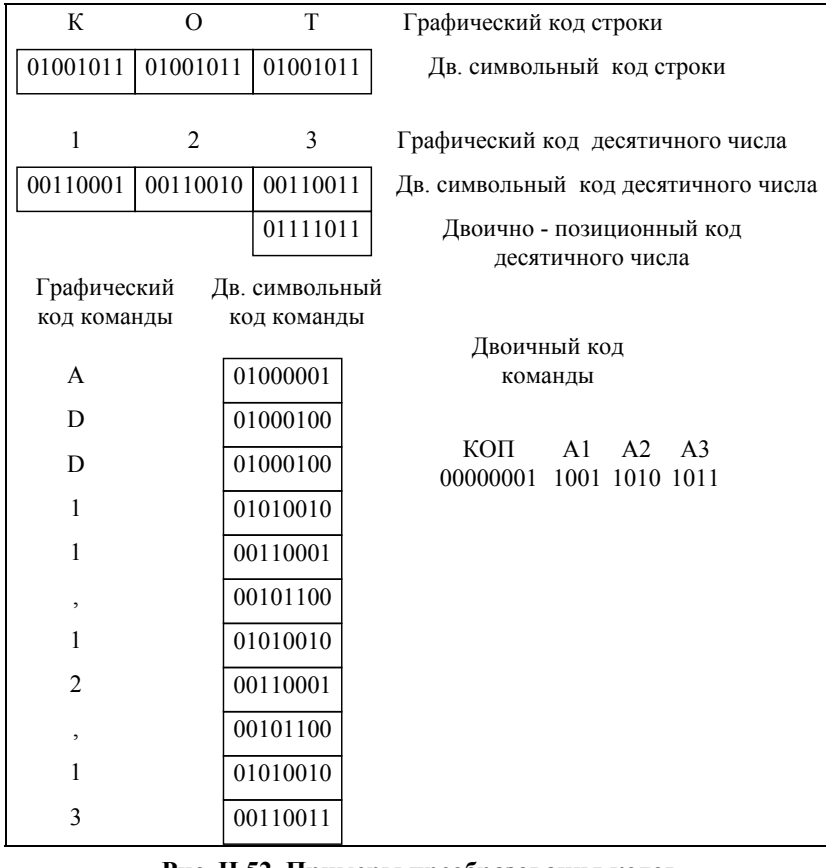


Рис. II.52. Примеры преобразования кодов

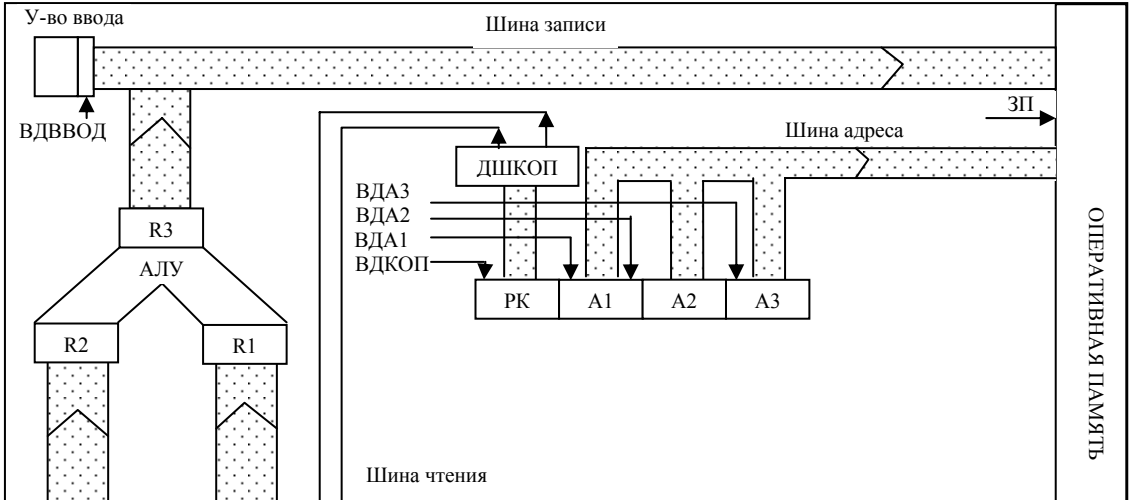
9.5. Контрольные вопросы

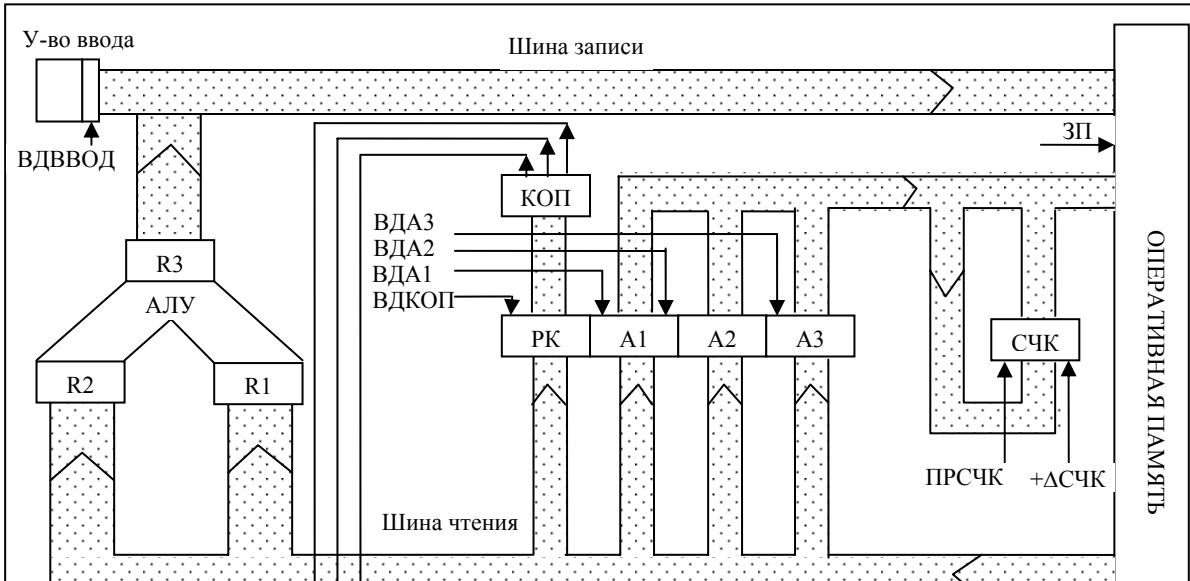
1. Два уровня представления данных и программ в компьютерах.
2. Языки символьного кодирования и ассемблирования программ.
3. Программы загрузчики.
4. Понятие подпрограммы.
5. Макросы как средство статической подстановки подпрограмм.
6. Переменная в императивных языках программирования высокого уровня.
7. Оператор присваивания как императив вычисления функции.
8. Операторы обработки данных.
9. Операторы изменения хода вычислительного процесса.
10. Подпрограммы процедуры и подпрограммы функции.
11. Трансляция и компиляция.

Литература

1. Я. Чу. Организация ЭВМ и микропрограммирование. М.: Мир. 1975.
2. М.А. Карцев. Архитектура цифровых вычислительных машин. М.: Наука, гл. ред. ф-м литературы, 1978.
3. С.А. Майоров, Г.И. Новиков. Структура электронных вычислительных машин. Л.: Машиностроение, 1979.
4. Уокерли. Архитектура и программирование микро – ЭВМ. М., Мир. 1984.
5. П. Нортон. Программно – аппаратная реализация IBM PC. М., Наука: 1992.
6. Э. Таненбаум. Архитектура компьютера. М.: Санкт-Петербург, Питер, 2002.
7. Т. Пратт, М. Зелковиц. Языки программирования. Разработка и реализация. М.: Санкт-Петербург, Питер, 2002.

Приложения


У-во ввода
Шина записи
ЗП
ВДВВОД
Шина адреса
ДШКОП
ВДА3
ВДА2
ВДА1
ВДКОП
R3
АЛУ
R2
R1
РК
А1
А2
А3
Шина чтения
ОПЕРАТИВНАЯ ПАМЯТЬ


У-во ввода
Шина записи
ВДВВОД
ЗП
КОП
ВДА3
ВДА2
ВДА1
ВДКОП
R3
АЛУ
R2
R1
РК
А1
А2
А3
СЧК
ПРСЧК
+ΔСЧК
ОПЕРАТИВНАЯ ПАМЯТЬ
Шина чтения

Алфавитный указатель

А

Абсолютная адресация 110
Автодекрементная адресация 114
Автоинкрементная адресация 113
Адрес доступа. 27
Адрес команды 71
Адрес регистра 27
Адресация компоненты задачи по базе с индексированием 119
Адресация по базе 117
Адресация по базе с индексированием 119
Адресация с индексированием 117
Адресная часть команды 99
Адресное пространством 27
Алгебра целых чисел без знака 11
Алгоритм 70
Алгоритм главного цикла 73
Алгоритм исполнения операции записи 28
Алгоритм исполнения операции чтения 28
Алгоритмы перевода 17
Алфавит цифр 14
Анализ прерываний 136
Арифметико-логическое устройство 61
Арифметические команды 66
Ассемблер 151
Атрибут 5

Б

Байт 29, 36
Библиотеки стандартных программ 93
Бит 29
Буфер 142
Буферизация ввода – вывода 129

В

Векторный компьютер 144
Вещественное число 40
Виртуальная машина 95
Виртуальная память 121
Внешние запоминающие устройства 88
Внешние наборы данных 7
Выполнение программы 80
Вычисление арифметического выражения 108
Вычислительная система 128

Г

Групповой обмен данными 89
Групповой признак 5

Д

Данные 5
Двоичное запоминающее устройство 26
Двоичный вектор памяти 57
Двоичный запоминающий регистр. 25
Двоичный запоминающий элемент 22
Двоичный носитель слова 24
Двоичный носитель текста 26
Двоичный символ 22
Двоичный файл 90
Двухадресная команда 101
Двухуровневое представление данных 8
Действие обработки данных 98
Дефиниция 5
Дешифратор 28
Диалоговый режим 96
Дизъюнктивная нормальная форма 58
Дизъюнкция 58
Длина команды 71
Доступный регистр 27
Драйвер 134

Е

Единая шина данных 133
Единица данных 6

З

Загрузчик 152
Задача 95
Запись 14, 88
Запись числа 14

Знака числа 61

И

Иерархическая структура памяти компьютера 91
Индексно-последовательный файл 90
Индексный регистр 119
Индивидуальный признак 5
Интерактивный режим 96
Интерпретатор 163
Инфиксная форма арифметического выражения 107
Информация 4
Исполнитель 70
Исполнительный адрес 113, 116

К

Канальная организация ввода–вывода 130
Каноническое арифметическое выражение 14
Клон задачи 145
Команда 80, 98
Команда 98
Команда безусловной передачи управления: 74
Команда ввода 68
Команда вывода 68
Команда обработки данных 63
Команда обработки данных 67
Команда преобразования данных 70
Команда условной передачи управления 75
Команды управления 74
Компилятор 162
Конвейер 143
Конструкция автоматической цифровой вычислительной машины 77
Концептуальный класс 4
Конъюнкция 58
Косвенная регистровая адресация 112
Кэш память. 124

Л

Линия связи. 22
Логические команды 66

М

Макро 153
Макрогенератор 153
Макроопределение 152
Макроподстановка 153
Макрос 152
Максимально представимо число 34
Максимально представимое целое число 38
Машинные наборы данных 7
Механизм прерываний 135
Микрокоманда 68, 80
Микрооперация 67, 80, 83
Микропрограмма 83
Микропрограммирование 83
Минимально представимое целое число 38
Мнемокоманда 150
Модель вычислительного процесса 74
Мультикомпьютер 146
Мультиплексный канал 131
Мультипрограммный режим 137
Мультипроцессор 144

Н

Набор данных 5
Набором данных 6
Нерезидентная часть операционной системы 94
Нить 145
Носитель данных 22

О

Обработка прерывания 136
Одноадресная команда 103
Одноразрядный сумматор 62
Оператор безусловного перехода 158
Оператор условного перехода 158
Оператор цикла 159
Операторы обработки данных 157
Операционная система 93
Операционная часть команды 98
Операция записи слова в регистр памяти 27
Операция включения в стек 106
Операция записи 22
Операция записи в ДЗЭ 22
Операция записи в регистр 25

Операция исключения из стека 106
Операция чтения 22
Операция чтения из ДЗЭ 23
Операция чтения из регистра 25
Операция чтения слова из регистра памяти 27
Определение компьютера 78
Основание 14
Относительная адресация 120
Отрицание 58

П

Пакетный режим 96
Память прямого доступа 29
Параметры действия 98
Переменная 155
Переполнение разрядной сетки 35
Переполнения разрядной сетки 63
Планировщик 141
Подкачка 142
Подпрограммы 160
Подпрограммы процедуры 160
Подпрограммы функции 160
Позиционная система счисления 14
Позиционная система счисления вещественных чисел 40
Позиционная система счисления целых чисел 38
Полный сумматор 62
Порт ввода – вывода 131
Последовательный файл 89
Постоянное запоминающее устройство 83
Постфиксная форма арифметического выражения 107
Поток управления 145
Представление "вещественных" чисел 41
Прерывание 136
Примитивная запись числа 13
Принцип программного управления 71
Принципы фон - Неймана 77
Присоединенная адресация 116
Программа 80
Программа 74
Программная реализация алгоритма 81
Проекция действий на команду 100
Процесс 95
Прямая адресация 100, 110
Прямое управление 128
Прямой, обратный и дополнительный коды числа 63
Псевдокоманды 151

Р

Работа 95
Равнодоступная память 29
Раздельные шины данных 132
Разряд числа 14
Разрядная сетка регистра 26
Разрядная сетка числа 14
Распределение ресурсов 94
Реализация "дружественного" интерфейса 94
Регистр – аккумулятор 103
Регистр базы 118
Регистр команд 67
Регистр прерываний 138
Регистровая адресация 112
Регистры общего назначения 111
Резидентная часть операционной системы 94
Ресурсы 95

С

Сегментная организация оперативной памяти 116
Сигнал прерывания 136
Символ 5
Система команд вычислительной машины 75
Система команд компьютера 74
Слово 29
Слово состояния программы 140
Смещение 119
Состояние ДЗЭ 22
Состояние носителя слова. 25
Способы адресации данных. 99
Средства автоматизации программирования 93
Стековая память 106
Страничная организация памяти 115
Суперскалярная архитектура 144
Сущность 4
Схемотехническая реализация алгоритма 81
Счетчик команд 72

Т

Текстовый файл 90
Тестовая функция "Пустой стек" 106
Тип данных 34, 37
Тип данных 33
Тип данных «вещественное» 40
Типы данных 35
Траектория вычислительного процесса 70
Транслятор 161
Трансляторы 93
Трехадресная команда 72, 100
Три основных функции операционной системы 94

У

Управление данными 94
Устройство микропрограммного управления 83
Устройство управления автоматическим выполнением команды 67
Устройство управления автоматическим выполнением программы 72

Ф

Файл 88
Файл прямого доступа 90
Формат команды 99
Формат с плавающей точкой 41
Формат с фиксированной точкой 41
Формат хранения целого числа без знака 35
Форматы хранения целого числа 39

Ц

Целое число 37
Целое число без знака 11
Цель обработки информации 4
Цикл прерывания 141
Цифра 14

Я

Язык автокод 150
Язык ассемблера 151
Язык программирован статический 156
Язык программирования высокого уровня 155
Язык программирования динамический 156
Язык программирования машинный 149
Язык программирования процедурный 149
Языки программирования 93
Ячейка памяти 33

Краткое содержание

ПРЕДИСЛОВИЕ 3

ГЛАВА I. ФУНДАМЕНТАЛЬНЫЕ ПРИНЦИПЫ АВТОМАТИЧЕСКОЙ ОБРАБОТКИ ДАННЫХ 4

1. Информация и данные 4
2. Представление целых неотрицательных чисел 11
3. Оперативная память 22
4. Типы данных 33
5. Сложение и вычитание целых неотрицательных чисел .. 46
6. Автоматическое исполнение команды обработки данных 56
7. Автоматическое исполнение программы 70

ГЛАВА II. СОВЕРШЕНСТВОВАНИЕ АРХИТЕКТУРЫ ВЫЧИСЛИТЕЛЬНОЙ МАШИНЫ - КОМПЬЮТЕРА 80

1. Организация управления компьютером 80
2. Иерархия памяти компьютера 88
3. Операционные системы и системы программирования . 93
4. Совершенствование адресации оперативной памяти 98
5. Структурная организация вычислительной машины ... 127
6. Параллельная работа центрального процессора и устройств ввода-вывода 135
7. Мультипрограммный режим работы вычислительной системы 137
8. Компьютеры параллельного действия 142
9. Автоматизация программирования 148

ЛИТЕРАТУРА 166

ПРИЛОЖЕНИЯ 167

АЛФАВИТНЫЙ УКАЗАТЕЛЬ 169

Содержание

ПРЕДИСЛОВИЕ 3

ГЛАВА I. ФУНДАМЕНТАЛЬНЫЕ ПРИНЦИПЫ АВТОМАТИЧЕСКОЙ ОБРАБОТКИ ДАННЫХ 4

1. Информация и данные 4
1.1. Контрольные вопросы 10
2. Представление целых неотрицательных чисел 11
2.1. Проблемы представления целых неотрицательных чисел 11
2.2. Примитивная система представления целых без знака 12
2.3. Позиционная система обозначения целых без знака 13
2.4. Перевод записи числа из одной системы счисления в другую 17
2.5. Перевод записи числа из одной системы в другую, если основания систем кратны двум 20
2.6. Контрольные вопросы 21
3. Оперативная память 22
3.1. Двоичный запоминающий элемент 22
3.2. Двоичный запоминающий регистр 24
3.3. Двоичная память 26
3.4. Почему компьютерная память двоичная 30
3.5. Контрольные вопросы 32
4. Типы данных 33
4.1. Понятие типа данных 33
4.2. Тип данных «целое без знака» 34
4.3. Типы данных «символ» и «строка символов» 35
4.4. Тип данных «целое» 37
4.5. Тип данных «вещественное» 40
4.6. Специфика обработки чисел на компьютере 43
4.7. Интерпретация двоичного кода единицы данных 44
4.8. Контрольные вопросы 45
5. Сложение и вычитание целых неотрицательных чисел . 46
5.1. Сложение B-ичных целых без знака (вычисления производятся в десятичной системе) 46
5.2. Вычитание B-ичных целых без знака (вычисления производятся в десятичной системе) 49
5.3. Сложение с использованием таблицы сложения 52
5.4. Контрольные вопросы 55
6. Автоматическое исполнение команды обработки данных 56
6.1. Символьный характер арифметических операций 56
6.2. Сложение целых положительных чисел 57

6.3. Арифметико-логическое устройство и команда обработки данных *60*
6.4. Сложение положительных и отрицательных чисел *63*
6.5. Арифметические команды *66*
6.6. Логические команды *66*
6.7. Команда обработки данных и ее автоматическое исполнение *67*
6.8. Контрольные вопросы *69*
7. Автоматическое исполнение программы 70
7.1. Понятие линейной программы *70*
7.2. Принцип программного управления *70*
7.3. Автоматическое исполнение линейной программы *72*
7.4. Понятие программы и ее автоматическое исполнение *73*
7.5. Система команд вычислительной машины *75*
7.6. Принципы функционирования вычислительной машины - компьютера *76*
7.7. Контрольные вопросы *79*

ГЛАВА II. СОВЕРШЕНСТВОВАНИЕ АРХИТЕКТУРЫ ВЫЧИСЛИТЕЛЬНОЙ МАШИНЫ - КОМПЬЮТЕРА 80

1. Организация управления компьютером 80
1.1. Принципы и алгоритмы управления *80*
1.2. Схемотехническая реализация управления *81*
1.3. Микропрограммная реализация управления *83*
1.4. Контрольные вопросы *87*
2. Иерархия памяти компьютера 88
2.1. Контрольные вопросы *92*
3. Операционные системы и системы программирования. 93
3.1. Контрольные вопросы *97*
4. Совершенствование адресации оперативной памяти 98
4.1. Действия и команды *98*
4.2. Трехадресные команды *99*
4.3. Число адресов команды и размер адресного пространства оперативной памяти *101*
4.4. Команды и байтовая структура оперативной памяти *105*
4.5. Использование стековой памяти *106*
4.6. Однокомпонентные способы адресации *110*
4.7. Многокомпонентные способы адресации *114*
4.8. Специальные виды памяти *121*
4.9. Контрольные вопросы *126*
5. Структурная организация вычислительной машины ... 127
5.1. Вводом-выводом управляет центральный процессор (прямое управление) *128*

5.2. Буферизация данных от ввода – вывода *129*
5.3. Структурная организация больших компьютеров *130*
5.4. Структурная организация миникомпьютеров и персональных компьютеров *131*
5.5. Контрольные вопросы..... *134*
6. Параллельная работа центрального процессора и устройств ввода-вывода 135
6.1. Контрольные вопросы..... *136*
7. Мультипрограммный режим работы вычислительной системы..... 137
7.1. Разделение времени между задачами..... *137*
7.2. Прерывания и их обработка..... *138*
7.3. Слово состояния программы и цикл прерываний *140*
7.4. Контрольные вопросы..... *141*
8. Компьютеры параллельного действия..... 142
8.1. Параллелизм на уровне одной команды *142*
8.2. Параллелизм на уровне данных..... *144*
8.3. Клонирование задачи в виде потока управления..... *144*
8.4. Параллелизм внутри одного потока управления *145*
8.5. Параллелизм на уровне процессов *146*
8.6. Контрольные вопросы..... *147*
9. Автоматизация программирования 148
9.1. От двоичных кодов к языкам символьного кодирования..... *148*
9.2. Языки программирования высокого уровня..... *154*
9.3. Трансляция программ *161*
9.4. Преобразование кодов при работе с компьютером.... *163*
9.5. Контрольные вопросы..... *165*

ЛИТЕРАТУРА..... 166

ПРИЛОЖЕНИЯ..... 167

АЛФАВИТНЫЙ УКАЗАТЕЛЬ 169

Кузин Станислав Григорьевич

Логические основы автоматической обработки данных

Учебное пособие

Научный редактор проф. Ю.Г. Васин
Компьютерный набор С. Г. Кузин
Редактор Е.В. Тамберг

Формат 60x84/1/16.
Бумага офсетная. Печать офсетная.
Уч. изд. л.13.5 Усл. печ.12.2 л. Тираж 500 экз. Заказ 610

Издательство Нижегородского государственного университета
им. Н.И. Лобачевского.
603950, Н. Новгород, просп. Гагарина, 23

Отпечатано с готового оригинал-макета в типографии
Нижегородского госуниверситета им. Н.И. Лобачевского.
Лиц. ПД № 18-0099 от 4.05. 2001.
603000, Н. Новгород, ул. Б. Покровская, 37