
СРЕДНЕЕ
ПРОФЕССИОНАЛЬНОЕ
ОБРАЗОВАНИЕ
КУРС
В.В. Степина
ОСНОВЫ АРХИТЕКТУРЫ, УСТРОЙСТВО И ФУНКЦИОНИРОВАНИЕ ВЫЧИСЛИТЕЛЬНЫХ СИСТЕМ
УЧЕБНИК

СРЕДНЕЕ ПРОФЕССИОНАЛЬНОЕ ОБРАЗОВАНИЕ

В.В. СТЕПИНА

ОСНОВЫ АРХИТЕКТУРЫ, УСТРОЙСТВО И ФУНКЦИОНИРОВАНИЕ ВЫЧИСЛИТЕЛЬНЫХ СИСТЕМ

УЧЕБНИК

Рекомендовано Экспертным советом при ГБОУ УМЦ ПО ДОгМ для использования в образовательном процессе профессиональных образовательных организаций города Москвы в качестве учебника для студентов среднего профессионального образования по специальности 2.09.02.04 «Информационные системы (по отраслям)»

Москва
КУРС
ИНФРА-М
2018

УДК 004.2(075.8)
ББК 32.973-02я73
С79

Рецензенты:
Е.И. Ночка — кандидат физико-математических наук кафедры «Информационные технологии» ГАПОУ Колледж предпринимательства № 11.

Степина В.В.

С79 Основы архитектуры, устройство и функционирование вычислительных систем: Учебник / В.В. Степина. — М.: КУРС: ИНФРА-М, 2018. — 288 с. — (Среднее профессиональное образование)

ISBN 978-5-906923-19-6 (КУРС)
ISBN 978-5-16-012698-2 (ИНФРА-М, print)
ISBN 978-5-16-102994-7 (ИНФРА-М, online)

Учебник создан в соответствии с Федеральным государственным образовательным стандартом по специальности 2.09.02.04 «Информационные системы (по отраслям)».

Рассмотрены цифровые вычислительные системы и их архитектурные особенности, работа основных логических блоков системы, вычисления в многопроцессорных и многоядерных системах. Дана классификация вычислительных платформ. Описаны методы повышения производительности многопроцессорных и многоядерных систем. Значительное внимание уделено Организации памяти в микропроцессорных системах (МПС) и подсистеме прерываний в МПС.

УДК 004.2(075.8)
ББК 32.973-02я73

Соответствует
ральному государственному
образовательному стандарту
3-го поколения

ISBN 978-5-906923-19-6 (КУРС)
ISBN 978-5-16-012698-2 (ИНФРА-М, print)
ISBN 978-5-16-102994-7 (ИНФРА-М, online)

ПРЕДИСЛОВИЕ

Учебник ориентирован на дисциплину «Основы архитектуры, устройство и функционирование вычислительных систем» для специальности 09.02.04 «Информационные системы (по отраслям)».

Учебник построен на фундаментальном утверждении современного классика компьютерной науки Эндрю Танненбаума о том, что компьютер можно рассматривать как иерархию структурных уровней организации.

Это утверждение в равной мере относится как к аппаратной организации, так и к структуре и организации программного обеспечения. Более того, исследование взаимодействия уровней организации компьютеров показывает, что нередко затруднительно провести четкую границу между аппаратной и программной реализациями функциональностей. Однако просматривается определенная закономерность, состоящая в том, что чем ниже расположен уровень, тем в большей степени он реализован средствами аппаратуры (а не средствами программирования).

Появляются новые языки программирования еще более высокого уровня, и каждый такой язык использует своего предшественника как основу, поэтому вычислительную машину можно рассматривать в виде ряда многоуровневой машины, и самый нижний уровень — это уровень физической реализации цифровых логических элементов.

Подход к компьютеру как к многоуровневой иерархической структуре позволяет продолжать заниматься дальнейшей разработкой системы, работая на вышележащем уровне. Однако реализация по крайней мере некоторых функций с использованием возможностей нижележащих уровней позволяет повысить эффективность системы, иногда ценой повышения трудоемкости разработки.

Изучение иерархических уровней (особенно изучение уровня команд процессора) дает возможность приобрести знания, использование которых может помочь как системному, так и прикладному программисту в эффективной разработке программного продукта, позволяя повысить эксплуатационные характеристики программных модулей (уменьшить объем требуемой памяти, увеличить скорость выполнения).

Учебник состоит из семи глав.

Первая глава, предназначена для знакомства с уровнем физической реализации и с некоторыми начальными сведениями о вычислительных системах:

- технические и эксплуатационные характеристики вычислительных машин;
- классическая архитектура вычислительной машины;
- основные характеристики и структура типового микропроцессора;
- иерархия памяти вычислительных систем.

Во *второй главе* рассматривается цифровой логический уровень: арифметические и логические основы вычислительной техники, основные цифровые логические устройства (комбинационные логические устройства и устройства с памятью).

Третья глава посвящена уровню микроархитектуры. На этом уровне в обработке команд участвует арифметико-логическое устройство (АЛУ).

В *четвертой главе* представлен уровень архитектуры набора команд. На этом уровне рассматривается архитектура памяти, взаимодействие с внешними устройствами ввода/вывода, режимы адресации, регистры, машинные команды, обработчики прерываний и исключительных состояний.

В *пятой главе* описывается уровень операционных систем. Приводится краткий обзор операционных систем, управление ресурсами вычислительных систем.

Шестая глава посвящена четвертому уровню, который представляет собой символическую форму одного из языков более низкого уровня — ассемблера. Рассмотрены примеры программ на языке ассемблера для конкретной архитектуры, операционной системы и варианты синтаксиса языка.

В *седьмой главе* рассматриваются параллельные вычислительные системы, производительность параллельных вычислительных систем и тенденции развития вычислительных систем.

Глава 1
ВЫЧИСЛИТЕЛЬНЫЕ СИСТЕМЫ. НАЧАЛЬНЫЕ СВЕДЕНИЯ

Изучение любого вопроса принято начинать с договоренностей о терминологии.В нашем случае определению подлежат такие термины, как «электронная вычислительная машина» (ЭВМ), «вычислительная система» (ВС), «информационная система» (ИС) и понятие «архитектура вычислительных систем».

1.1. Основные понятия и определения

Определение вычислительной машины

Согласно ГОСТ 15971-90 **вычислительная машина** (ВМ) — совокупность технических средств, создающая возможность проведения обработки информации (данных) и получения результата в необходимой форме. Под *техническими средствами* понимают все оборудование, предназначенное для автоматизированной обработки данных. Как правило, в состав ВМ входит и системное программное обеспечение.

Вычислительную машину, основные функциональные устройства которой выполнены на электронных компонентах, называют *электронной вычислительной машиной* (ЭВМ).

В последнее время в отечественной литературе широкое распространение получил англоязычный термин «компьютер» (англ. *computer* — вычислитель). Мы будем использовать эти термины как равноправные. Следует отметить, что в настоящее время активно ведутся разработки компьютеров, работа которых основана на оптических, фотонных, квантовых и других физических принципах. Например, оптические компьютеры в своей работе используют скорость света, а не скорость электричества, что делает их наилучшими проводниками данных. Сверхъестественный мир квантовой механики не подчиняется законам общей классической физики. Квантовый бит (*qubit*) не существует в типичных 0- или 1-бинарных формах сегодняшних компьютеров — квантовый бит может существовать в одной из них или же в обеих системах одновременно. В связи с этим понятие «электронная вычислительная машина», в котором акцентируется, что машина построена на основе электронных устройств, становится более узким, чем понятие «компьютер».

С развитием вычислительной техники появились многопроцессорные системы и сети, объединяющих большое количество отдельных процессоров и вычислительных машин, программные системы, реализующие параллельную обработку данных на многих вычислительных узлах. Появился термин «вычислительные системы».

Определение вычислительной системы

Система (от греч. *systema* — целое, составленное из частей соединение) — это совокупность элементов (объектов), взаимодействующих друг с другом, образующих определенную целостность, единство.

Объект (от лат. *objectum* — предмет) — это термин, используемый для обозначения элементов системы.

Вычислительную систему (ВС) стандарт ISO/IEC 2382/1-93 определяет как одну или несколько вычислительных машин, периферийное оборудование и программное обеспечение которых выполняют обработку данных.

Вычислительная система состоит из связанных между собой средств вычислительной техники, содержащих не менее двух основных процессоров, имеющих общую память и устройства ввода-вывода.

Формально отличие ВС от ВМ выражается в количестве вычислительных средств. Множественность этих средств позволяет реализовать в ВС параллельную обработку.

Таким образом, вычислительная система является результатом интеграции аппаратных средств и программного обеспечения, функционирующих в единой системе и предназначенных для совместного выполнения информационно-вычислительных процессов.

Аппаратное средство (*hardware*) включает в себя все внешние и внутренние физические компоненты компьютерной системы [п. 3.7.2 ГОСТ Р 53394-2009].

Программное обеспечение (*software*) по ГОСТ Р 53394-2009 — это совокупность информации (данных) и программ, которые обрабатываются компьютерной системой.

С технической точки зрения вычислительная система — это комплекс вычислительных средств, объединенных в информационно-вычислительную сеть.

Основной отличительной чертой вычислительных систем по отношению к ЭВМ является наличие в них нескольких вычислителей, реализующих параллельную обработку. Точное различие между вычислительными машинами и вычислительными системами опреде-

лить невозможно, так как вычислительные машины даже с одним процессором обладают разными средствами распараллеливания, а вычислительные системы могут состоять из традиционных вычислительных машин или процессоров.

Определение информационной системы

Необходимо понимать разницу между компьютерами и информационной системой: компьютеры оснащены специальными программными системами, являются технической базой и инструментом для информационных систем.

Информационная система — это организационно упорядоченная совокупность документов (массивов документов) и информационных технологий, в том числе с использованием средств вычислительной техники и связи, реализующих информационные процессы [1, ст. 2] [п. 3.1.7 ГОСТ Р 54089—2010].

Информационная система немыслима без персонала, взаимодействующего с компьютерами и телекоммуникациями.

Информационная система с технической точки зрения — это взаимосвязанная совокупность средств, методов и персонала, используемых для хранения, обработки и выдачи информации в интересах достижения поставленной цели.

Понятие архитектуры вычислительных машин и вычислительных систем

С развитием средств вычислительной техники изменился подход к созданию вычислительных машин. Вместо разработки аппаратуры и средств математического обеспечения стала проектироваться система, состоящая из синтеза аппаратных и программных средств. При этом на главный план выдвинулась концепция взаимодействия аппаратных и программных средств. Так возникло новое понятие — «архитектура ЭВМ».

Архитектура (*architecture*) — это базовая организация системы, воплощенная в ее компонентах, их отношениях между собой и с окружением, а также принципы, определяющие проектирование и развитие системы [ИСО/МЭК 15288:2008, определение 4.5].

Архитектура вычислительной машины (*computer architecture*) — это концептуальная структура вычислительной машины, определяющая проведение обработки информации и включающая методы преобразования информации в данные и принципы взаимодействия технических средств и программного обеспечения [ГОСТ 15971-90, определение 29].

Таким образом, архитектуру вычислительной машины можно представить как множество взаимосвязанных компонентов, включающих элементы различной природы: программное обеспечение (*software*), аппаратное обеспечение (*hardware*), алгоритмическое обеспечение (*brainware*), специальное фирменное обеспечение (*firmware*), создающих возможность проведения обработки информации и получения результата в необходимой форме.

Следует отличать архитектуру вычислительной машины от ее структуры.

Структура — это отношение между элементами системы [ISO/IEC 2382/1-93].

Структура вычислительной машины определяет отношение между ее элементами (множество взаимосвязанных компонентов) на уровне детализации. Элементами детализации могут быть различные функциональные узлы (блоки, устройства и т.д.). Графически описание вычислительной машины на любом уровне детализации представляется в виде структурных схем.

Под *архитектурой вычислительной машины* понимают общее описание принципов организации аппаратно-программных средств и основных их характеристик, определяющих функциональные возможности вычислительной машины.

Архитектура вычислительной системы — совокупность характеристик и параметров, определяющих функционально-логичную и структурно-организованную систему и затрагивающих в основном уровень параллельно работающих вычислителей.

Понятие архитектуры охватывает общее понятие организации системы, включающее такие высокоуровневые аспекты разработки компьютера, как система памяти, структура системной шины, организация ввода/вывода и т.п.

Архитектура определяет принципы действия, информационные связи и взаимное соединение основных логических узлов компьютера: процессора, оперативного ЗУ, внешних ЗУ и периферийных устройств. Общность архитектуры разных компьютеров обеспечивает их совместимость с точки зрения пользователя.

1.2. Многоуровневая организация вычислительных машин

Согласно всемирно известному специалисту в области информационных технологий Эндрю Таненбауму, в основе структурной организации компьютера лежит идея иерархической структуры, в ко-

торой каждый уровень выполняет вполне определенную функцию. Это утверждение в равной мере относится как к аппаратной организации, так и к организации программного обеспечения.

Достоинства такого представления вычислительных машин:

- каждый верхний уровень интерпретируется одним или несколькими нижними уровнями;
- каждый из уровней можно проектировать независимо;
- чем ниже уровень, на котором реализуется программа, тем более высокая производительность достижима;
- модификация нижних уровней не влияет на реализацию верхних.

Понятие семантического разрыва между уровнями

Преобразование операторов языков высокого уровня в машинный код или в микрокоманды требует от транслятора, во-первых, умения распознать операторы и команды различных уровней и, во-вторых, для любого оператора языка высокого уровня генерировать десятки или сотни команд низкого уровня. Это приводит к усложнению транслятора, увеличению трудоемкости его разработки, снижению производительности генерируемых программ. Наличие этих проблем называют семантическим разрывом между уровнями. Основным способом его преодоления является специализация машин, при которой операторы проблемно-ориентированных языков могут непосредственно выполняться аппаратными средствами машины и не требовать трансляции, например аппаратная реализация графических преобразований, аппаратная реализация операций с векторами и матрицами.

Языки, уровни и виртуальные машины

Основой функционирования любой вычислительной машины является ее способность выполнять заданные действия. Аппаратные средства любой вычислительной машины способны выполнять только ограниченный набор сравнительно простых команд. Эти примитивные команды составляют так называемый машинный язык. Говоря о сложности аппаратуры компьютера, машинные команды целесообразно делать как можно проще, но примитивность большинства машинных команд делает их использование неудобным и трудным. Для эффективной работы человека с компьютером разработчики вводят другой набор команд, более удобный для человеческого общения (языки более высокого уровня).

Появляются новые языки программирования еще более высокого уровня, и каждый такой язык использует своего предшественника

как основу, поэтому вычислительную машину рассмотрим в виде ряда многоуровневой машины, структура которой изображена на рис. 1.1. Между языками программирования и существующей виртуальной машиной существует тесная связь. Язык, находящийся в самом низу иерархической структуры компьютера, является примитивным, а тот, что расположен на ее вершине, — самым сложным. Большинство современных вычислительных машин включают 6–7 уровней виртуализации. Нижние уровни, начиная с машинного, более консервативны к изменениям.

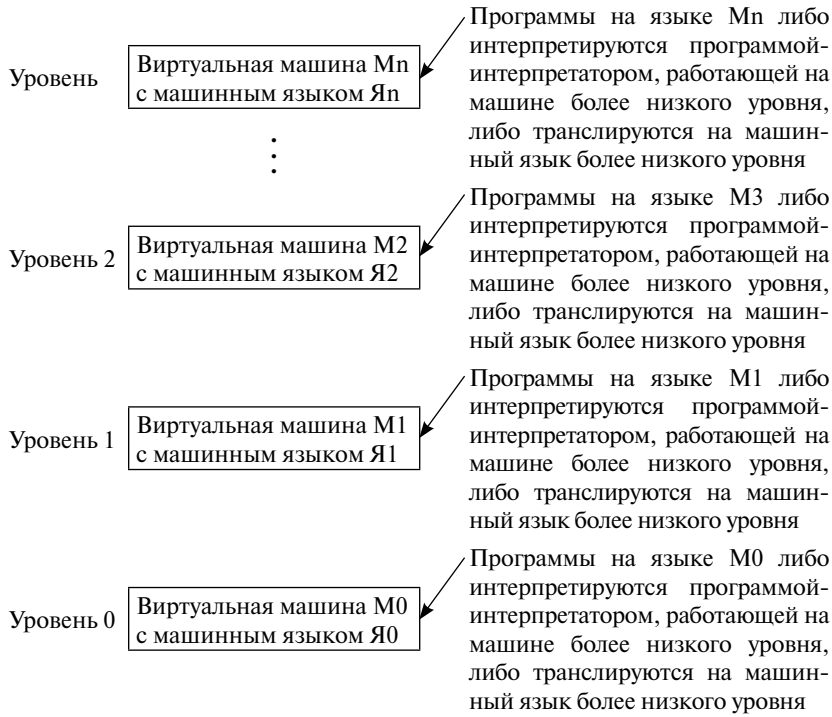


Рис. 1.1. Структура многоуровневой вычислительной машины

Компьютер с *n* уровнями можно рассматривать как *n* разных виртуальных машин, у каждой из которых есть свой машинный язык. Термины «уровень» и «виртуальная машина» мы будем использовать как синонимы. Только программы, написанные на Я0, могут выполняться компьютером без трансляции или интерпретации. Программы, написанные на Я1, Я2, ..., Я*n*, должны проходить через ин-

терпретатор более низкого уровня или транслироваться на язык, соответствующий более низкому уровню.

Современные многоуровневые вычислительные машины

Современные компьютеры можно представить как структуру, состоящую из шести уровней (рис. 1.2).

Рис. 1.2. Структура шестиуровневого компьютера

Способ поддержки каждого уровня указан под ним, в скобках дано название соответствующего программного обеспечения.

Существуют уровни, которые расположены ниже нулевого уровня. Эти уровни не рассматриваются из-за сложности материала, так как они попадают в сферу таких направлений, как элементная база микроэлектроники и наноэлектроники, микросистемная техника, твердотельная электроника.

Уровень 0: Цифровой логический уровень. Цифровой логический уровень представляет собой аппаратное обеспечение компьютера. Объектами цифрового логического уровня являются цифровые логические устройства. Для описания того, как функционируют циф-

ровые логические устройства, применяется математический аппарат алгебры логики, в которой используются логический ноль и логическая единица.

Основу для проектирования сложных цифровых устройств функции составляют базовые логические элементы — это схемы, содержащие электронные ключи (вентили) и выполняющие основные логические операции. Сложные логические функции можно выразить через совокупность конечного числа базисных логических функций, таких как НЕ, И, ИЛИ.

Одним из важнейших элементов цифровой техники является триггер (англ. trigger — защелка, спусковой крючок). Триггер имеет два устойчивых состояния, одно из которых соответствует двоичной единице, а другое — двоичному нулю. Сам триггер не является базовым элементом, так как он собирается из более простых логических схем.

Триггер — это электронная схема, широко применяемая в регистрах компьютера для надежного запоминания двоичной единицы (бита памяти).

Биты памяти, объединенные в группы, например, по 16, 32 или 64, формируют регистры. Каждый регистр может содержать одно двоичное число до определенного предела.

Уровень 1: уровень микроархитектуры. В компьютерной инженерии микроархитектура (англ. microarchitecture), также называемая организацией компьютера, — это способ, которым данная архитектура набора команд (ISA, АНК) реализована в процессоре. Каждая АНК может быть реализована с помощью различных микроархитектур. Реализации могут варьироваться в зависимости от целей конкретной разработки или в результате технологических сдвигов. Архитектура компьютера является комбинацией микроархитектуры, микрокода и АНК.

На этом уровне в обработке команд участвует арифметико-логическое устройство (АЛУ).

АЛУ состоит из регистров, сумматора с соответствующими логическими схемами и элемента управления выполняемым процессом. Оперативная память организована в виде последовательностей, которые группируются в машинные слова. Арифметико-логическое устройство работает в соответствии с сообщаемыми ему именами (кодами) операций, которые при пересылке данных нужно выполнить над переменными, помещаемыми в регистры. Над какими кодами производится операция, куда помещается ее результат — определяется выполняемой командой. Примерами обработки могут служить логические операции («И», «ИЛИ», «Исключающего ИЛИ»

и т.д.), т.е. побитные операции над операндами, а также арифметические операции (сложение, вычитание, умножение, деление и т.д.). Команды содержат от одного до трех операндов.

В операционном устройстве (АЛУ) реализуется заданная последовательность микрокоманд (команд), в микропрограммном устройстве управления (УУ) задается последовательность микрокоманд (команд).

Различают два вида микрокоманд: внешние — такие микрокоманды, которые поступают в АЛУ от внешних источников и вызывают в нем преобразование информации, и внутренние — те, которые генерируются в АЛУ и оказывают влияние на микропрограммное устройство, изменяя таким образом нормальный порядок следования команд.

Микропрограмма — это интерпретатор для команд на уровне 2. Микропрограмма вызывает команды из памяти и выполняет их одну за другой, а результаты вычислений из АЛУ передаются в оперативную память по кодовым шинам записи.

На некоторых машинах работа тракта данных контролируется особой программой, которая называется микропрограммой. На других машинах тракт данных контролируется аппаратными средствами.

Например, при выполнении команды ADD микропрограмма вызывает из памяти операнды команды сложения, помещает их в регистры, АЛУ вычисляет сумму операндов, а затем результат переправляется обратно в память. На компьютере с аппаратным контролем тракта данных происходит такая же процедура, но при этом нет программы, интерпретирующей команды уровня 2.

Машины с различной микроархитектурой могут иметь одинаковую архитектуру набора команд и, таким образом, быть пригодными для выполнения тех же программ. Новые микроархитектуры и/или схемотехнические решения вместе с прогрессом в полупроводниковой промышленности являются тем, что позволяет новым поколениям процессоров достигать более высокой производительности, используя ту же АНК.

Уровень 2: уровень архитектуры набора команд. Архитектура набора команд (англ. instruction set architecture, ISA) — часть архитектуры компьютера, определяющая программируемую часть ядра микропроцессора. На этом уровне определяются реализованные в микропроцессоре конкретного типа:

- архитектура памяти;
- взаимодействие с внешними устройствами ввода/вывода;
- режимы адресации;

- регистры;
- машинные команды;
- различные типы внутренних данных (например, с плавающей запятой, целочисленные типы и т.д.);
- обработчики прерываний и исключительных состояний.

В современных микроархитектурах используется конвейерный тракт. Конвейер содержит такие стадии, как выбор инструкций, декодирование инструкций, исполнение и запись результата. Некоторые архитектуры включают также доступ к памяти. Дизайн конвейера является фундаментальным при разработке микроархитектуры.

Устройства исполнения также являются ключевыми для микроархитектуры. Они включают арифметико-логические устройства, устройства обработки чисел с плавающей точкой, устройства выборки и хранения, прогнозирование ветвления, параллелизм на уровне данных (SIMD). Эти блоки производят операции или вычисления процессора. Выбор числа блоков исполнения, их задержек, пропускной способности и способа соединения памяти с системой также является микроархитектурным решением.

Проектные решения уровня системы, такие как включать или нет периферийные устройства типа контроллеров памяти, могут считаться частью процесса разработки микроархитектуры, поскольку они содержат решения по уровню производительности и способам соединения этих периферийных устройств.

В отличие от архитектурного дизайна, где достижение определенного уровня производительности является главной целью, проектирование микроархитектуры уделяет большее внимание другим ограничениям. Поскольку дизайн микроархитектуры прямо влияет на то, что происходит в системе, внимание должно быть уделено таким проблемам, как площадь/стоимость чипа, потребление энергии, сложность логики, технологичность, простота отладки и тестируемость.

Уровень 3: уровень операционной системы. Третий уровень операционной системы является гибридным. Большинство команд в его языке есть как на третьем, так и на втором уровне — уровне АНК (команды, имеющиеся на одном из уровней, вполне могут быть представлены и на других уровнях). У этого уровня есть некоторые дополнительные особенности: новый набор команд, другая организация памяти, способность выполнять две и более программы одновременно и некоторые другие. При построении уровня 3 возможно больше вариантов, чем при построении уровней 1 и 2.

Новые средства, появившиеся на уровне 3, выполняются интерпретатором, который работает на втором уровне. Этот интерпретатор

был когда-то назван операционной системой. Команды уровня 3, идентичные командам уровня 2, выполняются микропрограммой или аппаратным обеспечением, но не операционной системой. Другими словами, одна часть команд уровня 3 интерпретируется операционной системой, а другая часть — микропрограммой. Вот почему этот уровень считается гибридным.

Между уровнями 3 и 4 есть существенная разница. Нижние три уровня задуманы не для того, чтобы с ними работал обычный программист. Они изначально ориентированы на интерпретаторы и трансляторы, поддерживающие более высокие уровни. Эти трансляторы и интерпретаторы составляются так называемыми системными программистами, которые специализируются на разработке новых виртуальных машин. Уровни с четвертого и выше предназначены для прикладных программистов, решающих конкретные задачи.

Еще одно изменение, появившееся на уровне 4, — механизм поддержки более высоких уровней. Уровни 2 и 3 обычно интерпретируются, а уровни 4, 5 и выше обычно, хотя и не всегда, транслируются.

Другое отличие между уровнями 1, 2, 3 и уровнями 4, 5 и выше — особенность языка. Машинные языки уровней 1, 2 и 3 — цифровые. Программы, написанные на этих языках, состоят из длинных рядов цифр, которые воспринимаются компьютерами, но малопонятны для людей. Начиная с уровня 4 языки содержат слова и сокращения, понятные человеку.

Уровень 4: уровень ассемблера. Уровень 4 представляет собой символическую форму одного из языков более низкого уровня — ассемблера. *Ассемблер* (от англ. assembler — сборщик) — транслятор исходного текста программы, написанной на языке ассемблера, в программу на машинном языке.

Как и сам язык, ассемблеры, как правило, специфичны для конкретной архитектуры, операционной системы и варианта синтаксиса языка. Вместе с тем существуют мультиплатформенные или вовсе универсальные (точнее, ограниченно-универсальные, потому что на языке низкого уровня нельзя написать аппаратно-независимые программы) ассемблеры, которые могут работать на разных платформах и операционных системах. Среди последних можно также выделить группу кросс-ассемблеров, способных собирать машинный код и исполняемые модули (файлы) для других архитектур и операционных систем.

Ассемблирование может быть не первым и не последним этапом на пути получения исполнимого модуля программы. Так, многие компиляторы с языков программирования высокого уровня выдают результат в виде программы на языке ассемблера, которую в даль-

нейшем обрабатывает ассемблер. Также результатом ассемблирования может быть не исполняемый, а объектный модуль, содержащий разрозненные блоки машинного кода и данных программы, из которого (или из нескольких объектных модулей) в дальнейшем с помощью редактора связей может быть получен исполнимый файл.

Уровень 5: уровень языка прикладных программистов. *Прикладная программа* или *приложение* — программа, предназначенная для выполнения определенных задач и рассчитанная на непосредственное взаимодействие с пользователем. В большинстве операционных систем прикладные программы не могут обращаться к ресурсам компьютера напрямую, а взаимодействуют с оборудованием и другими программами посредством операционной системы.

К прикладному программному обеспечению относятся компьютерные программы, написанные для пользователей или самими пользователями для задания компьютеру конкретной работы. Программы обработки заказов или создания списков рассылки — пример прикладного программного обеспечения. Программистов, которые пишут прикладное программное обеспечение, называют прикладными программистами.

Языки, разработанные для прикладных программистов, называются языками высокого уровня. Существуют сотни языков высокого уровня. Наиболее известные среди них — C, C++, Java, LISP и Prolog. Программы, написанные на этих языках, обычно транслируются на уровень 3 или 4. Трансляторы, которые обрабатывают эти программы, называются *компиляторами*. Отметим, что иногда также имеет место интерпретация. Например, программы на языке Java сначала транслируются на язык, напоминающий ISA и называемый байт-кодом Java, который затем интерпретируется.

В некоторых случаях уровень 5 состоит из интерпретатора для конкретной прикладной области, например символической логики. Он предусматривает данные и операции для решения задач в этой области, выраженные при помощи специальной терминологии.

Таким образом, компьютер проектируется как иерархическая структура уровней, которые надстраиваются друг над другом. Каждый уровень представляет собой определенную абстракцию различных объектов и операций. Рассматривая компьютер подобным образом, мы можем не принимать во внимание ненужные нам детали и, таким образом, сделать сложный предмет более простым для понимания.

Набор типов данных, операций и характеристик каждого отдельно взятого уровня называется *архитектурой*. Архитектура связана с программными аспектами. Например, сведения о том, сколько

памяти можно использовать при написании программы,— часть архитектуры. Аспекты реализации (например, технология, применяемая при реализации памяти) не являются частью архитектуры. Изучая методы проектирования программных элементов компьютерной системы, мы изучаем компьютерную архитектуру.

1.3. Классическая архитектура вычислительной машины

Разнообразие современных вычислительных машин очень велико, но все они представляют собой реализацию так называемой фон-неймановской (принстонской) архитектуры, представленной Джорджем фон Нейманом еще в 1945 г. Рассмотрим классическую архитектуру вычислительной машины на примере архитектуры фон Неймана.

Принцип действия вычислительной машины состоит в выполнении программ последовательностей арифметических, логических и других операций, описывающих решение определенной задачи.

Программа (для ЭВМ) — это упорядоченная последовательность команд, подлежащая обработке (стандарт ISO 2382/1-84).

Команда — это описание операции, которую должна выполнить вычислительная машина.

Результат команды вырабатывается по точно определенным для данной команды правилам, заложенным в конструкцию компьютера. Электронные схемы каждого компьютера могут распознавать и выполнять ограниченный набор простых команд.

На рис. 1.3 представлена схема фон-неймановской вычислительной машины, на основе которой уже более полувека создаются современные вычислительные машины.

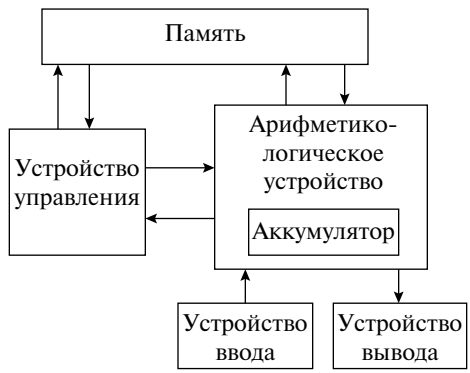


Рис. 1.3. Схема фон-неймановской вычислительной машины

Фон-неймановская архитектура состоит из следующих основных устройств:

- памяти, которая включала 4096 машинных слов разрядностью 40 бит. Машинное слово содержало или команды (две команды по 20 бит), или целое число со знаком на 40 бит (8 бит указывали на тип команды, а остальные 12 бит определяли одно из $2^{12} = 4096$ слов);
- арифметико-логического устройства (АЛУ), внутри которого находится особый внутренний регистр разрядностью 40 бит, так называемый аккумулятор;
- устройства управления (УУ), выполняющего функции управления устройствами;
- устройства ввода информации;
- устройства вывода информации.

Эти устройства соединены каналами связи, по которым передается информация.

В современных вычислительных машинах арифметико-логическое устройство и устройство управления сочетаются в одной микросхеме, которая называется *центральным процессором* (ЦП).

Машинная команда считывает слово из памяти в аккумулятор или сохраняет содержимое аккумулятора в памяти. Машина фон Неймана выполняла арифметические операции только с фиксированной точкой, поскольку фон Нейман был отличным математиком и считал, что плавающую точку можно держать в голове.

В основу построения современных вычислительных машин были положены следующие принципы фон Неймана.

1. *Принцип однородности памяти*. Команды и данные хранятся в одной и той же памяти и внешне в памяти неразличимы. Распознать их можно только по способу использования; т.е. одно и то же значение в ячейке памяти может использоваться и как данные, и как команда, и как адрес в зависимости лишь от способа обращения к нему. Это позволяет производить над командами те же операции, что и над числами, и, соответственно, открывает ряд возможностей. Так, циклически изменяя адресную часть команды, можно обеспечить обращение к последовательным элементам массива данных. Такой прием носит название модификации команд и с позиций современного программирования не приветствуется. Более полезным является другое следствие принципа однородности, когда команды одной программы могут быть получены как результат исполнения другой программы. Эта возможность лежит в основе трансляции — перевода текста программы с языка высокого уровня на язык конкретной вычислительной машины.

2. *Принцип адресности*. Структурно основная память состоит из пронумерованных ячеек, причем процессору в произвольный момент доступна любая ячейка. Двоичные коды команд и данных разделяются на единицы информации, называемые словами, и хранятся в ячейках памяти, а для доступа к ним используются номера соответствующих ячеек — адреса.

3. *Принцип программного управления*. Все вычисления, предусмотренные алгоритмом решения задачи, должны быть представлены в виде программы, состоящей из последовательности управляющих слов — команд. Каждая команда предписывает некоторую операцию из набора операций, реализуемых вычислительной машиной. Команды программы хранятся в последовательных ячейках памяти вычислительной машины и выполняются в естественной последовательности, т.е. в порядке их положения в программе. При необходимости с помощью специальных команд эта последовательность может быть изменена. Решение об изменении порядка выполнения команд программы принимается либо на основании анализа результатов предшествующих вычислений, либо безусловно.

4. *Принцип двоичного кодирования*. Согласно этому принципу вся информация, как данные, так и команды, кодируется двоичными цифрами 0 и 1. Каждый тип информации представляется двоичной последовательностью и имеет свой формат. Последовательность битов в формате, имеющая определенный смысл, называется *полем*. В числовой информации обычно выделяют поле знака и поле значащих разрядов. В формате команды можно выделить два поля: поле кода операции и поле адресов.

Огромным преимуществом фон-неймановской архитектуры является ее простота, поэтому данная концепция легла в основу большинства компьютеров общего назначения. Однако совместное использование шины для памяти программ и памяти данных приводит к так называемому узкому месту архитектуры фон Неймана. Термин «узкое место архитектуры фон Неймана» ввел Джон Бэкус в 1977 г. в своей лекции «Можно ли освободить программирование от стиля фон Неймана?», которую он прочитал при вручении ему Премии Тьюринга.

Фон-неймановская архитектура — не единственный вариант построения ЭВМ, есть и другие, которые не соответствуют указанным принципам (например, потоковые машины). Однако подавляющее большинство современных компьютеров основано именно на указанных принципах, включая и сложные, многопроцессорные комплексы, которые можно рассматривать как объединение фон-неймановских машин.

1.4. Технические и эксплуатационные характеристики вычислительных машин

Производительность вычислительной машины. Этот показатель определяется архитектурой процессора, иерархией внутренней и внешней памяти, пропускной способностью системного интерфейса, системой прерывания, набором периферийных устройств в конкретной конфигурации, совершенством ОС и т.д. Основные единицы оценки производительности:

- абсолютная, определяемая количеством элементарных работ, выполняемых в единицу времени;
- относительная, определяемая для оцениваемой ЭВМ относительно базовой в виде индекса производительности.

Для каждого вида производительности применяются следующие традиционные методы их определения.

Пиковая производительность (быстродействие) определяется средним числом команд типа «регистр—регистр», выполняемых в одну секунду без учета их статистического веса в выбранном классе задач.

Номинальная производительность (быстродействие) определяется средним числом команд, выполняемых подсистемой «процессор—память» с учетом их статистического веса в выбранном классе задач. Она рассчитывается, как правило, по формулам и специальным методикам, предложенным для процессоров определенных архитектур, и измеряется с помощью разработанных для них измерительных программ, реализующих соответствующую эталонную нагрузку.

Для данных типов производительностей используются следующие единицы измерения:

- MIPS (Mega Instruction Per Second) — миллион команд в секунду;
- MFLOPS (Mega Floating Operations Per Second) — миллион операций над числами с плавающей запятой в секунду;
- GFLOPS (Giga Floating Operations Per Second) — миллиард операций над числами с плавающей запятой в секунду и т.д.

Системная производительность измеряется с помощью синтезированных типовых (тестовых) оценочных программ, реализованных на унифицированных языках высокого уровня. Унифицированные тестовые программы используют типичные алгоритмические действия, характерные для реальных применений, и штатные компиляторы ЭВМ. Они рассчитаны на использование базовых технических средств и позволяют измерять производительность для расширенных конфигураций технических средств. Результаты оценки системной производительности ЭВМ конкретной архитектуры приводятся от-

носительно базового образца, в качестве которого используются ЭВМ, являющиеся промышленными стандартами систем ЭВМ различной архитектуры. Результаты оформляются в виде сравнительных таблиц, двумерных графиков и трехмерных изображений.

Эксплуатационная производительность оценивается на основании использования данных о реальной рабочей нагрузке и функционировании ЭВМ при выполнении типовых производственных нагрузок в основных областях применения. Расчеты делаются главным образом на уровне типовых пакетов прикладных программ текстообработки, систем управления базами данных, пакетов автоматизации проектирования, графических пакетов и т.д.

1.5. Процессоры

Центральное процессорное устройство (ЦПУ; англ. central processing unit, CPU; дословно — «центральное обрабатывающее устройство») — это электронный блок либо интегральная схема (микропроцессор), исполняющие машинные инструкции (код программ), главная часть аппаратного обеспечения компьютера или программируемого логического контроллера. Иногда называют микропроцессором (МП) или просто процессором.

Характеристики процессора

Современные процессоры представляют собой сложные, высокотехнологичные устройства и описываются целым рядом показателей. Важнейшими характеристиками процессора являются:

1) **тактовая частота** (clock rate) — это число элементарных операций (тактов), производимых за 1 секунду. Измеряется в герцах (Hz, Гц и их производных по системе СИ — килогерцах, kHz, кГц, мегагерцах, MHz, МГц, гигагерцах, GHz, ГГц).

Тактовая частота центрального микропроцессора определяет производительность как самого процессора, так и в целом всей компьютерной системы. Тем не менее с развитием современных технологий, таких как, например, суперскалярность, и созданием многоядерных процессоров нельзя установить прямо пропорциональной зависимости между тактовой частотой процессора и его производительностью;

2) **разрядность** — максимальное число одновременно обрабатываемых двоичных разрядов. Разрядность МП обозначается *m/n/k/* и включает:

- *m* — разрядность внутренних регистров, определяет принадлежность к тому или иному классу процессоров;

- n — разрядность шины данных, определяет скорость передачи информации;
- k — разрядность шины адреса, определяет размер адресного пространства. Например, МП i8088 характеризуется значениями $m/n/k = 16/8/20$;

3) **количество вычислительных ядер** на одном процессорном кристалле. Многоядерность как способ повышения производительности процессоров используется с относительно недавнего времени, но признана самым перспективным направлением их развития. Для домашних компьютеров уже существуют процессоры с восемью ядрами. Для серверов на рынке есть 12-ядерные предложения (Opteron 6100). Разработаны прототипы процессоров, содержащие около 100 ядер.Эффективность вычислительных ядер разных моделей процессоров отличается. Но в любом случае чем ядер больше, тем процессор производительнее;

4) **количество потоков.**Чем больше потоков — тем лучше. Количество потоков не всегда совпадает с количеством ядер процессора. Так, благодаря технологии Hyper-Threading 4-ядерный процессор Intel Core i7—3820 работает в восемь потоков и во многом опережает 6-ядерных конкурентов;

5) **размер кэш-памяти** 2-го и 3-го уровней, иногда и 4-го уровня. *Кэш* — это очень быстрая внутренняя память процессора, которая используется им как буфер для временного хранения информации, обрабатываемой в конкретный момент времени;

6) **скорость шины процессора** (FSB, HyperTransport или QPI).Через эту шину центральный процессор взаимодействует с материнской платой. Ее скорость (частота) измеряется в мегагерцах, и чем она выше — тем лучше;

7) **технологический процесс.** Чем выше степень интеграции (количество транзисторов на кристалле), тем меньше потребляемая электроэнергия процессора. От техпроцесса во многом зависит еще одна важная характеристика процессора — TDP;

8) **Termal Design Point** (TDP) — показатель, отображающий энергопотребление процессора, а также количество тепла, выделяемого им в процессе работы. Единицы измерения — ватты (Вт). TDP зависит от многих факторов, среди которых главными являются количество ядер, техпроцесс изготовления и частота работы процессора.

Кроме прочих преимуществ, «холодные» процессоры (с TDP до 100 Вт) лучше поддаются разгону, когда пользователь изменяет некоторые настройки системы, вследствие чего увеличивается частота

процессора. Разгон позволяет без дополнительных финансовых вложений увеличить производительность процессора на 15–25 %, но это уже отдельная тема. В то же время проблему с высоким TDP всегда можно решить приобретением эффективной системы охлаждения;

9) **наличие и производительность видеоядра.** Последние технические достижения позволили производителям помимо вычислительных ядер включать в состав процессоров еще и ядра графические. Такие процессоры, кроме решения своих основных задач, могут выполнять роль видеокарты;

10) **тип и максимальная скорость поддерживаемой оперативной памяти.** Эти характеристики процессора необходимо учитывать при выборе оперативной памяти, с которой он будет использоваться. Нет смысла переплачивать за быстрые модули ОЗУ, если процессор не сможет реализовать все их преимущества.

Развитие архитектуры процессора

Развитие вычислительной техники сопровождается совершенствованием центральных процессоров. При проектировании новых моделей процессоров разработчики основывались на принципах совместимости, т.е. разные модели процессоров, различающихся по производительности, должны были «уметь» выполнять одни и те же команды. Чтобы охарактеризовать этот уровень совместимости, компания IBM ввела термин «архитектура». Новое семейство процессоров должно было иметь единую архитектуру, т.е. новая модель процессора разрабатывается на основе какой-либо существующей архитектуры.

Под термином «архитектура процессора» понимают совокупность и способ объединения компонентов процессора, а также его совместимость с определенным набором команд.

Знание этих двух моментов дает возможность грамотно организовать интерфейс аппаратных и программных средств вычислительной системы. Например, с точки зрения программиста, архитектура процессора — это способность процессора выполнять набор машинных кодов, а с точки зрения проектирования компьютерных составляющих архитектура процессора — это отражение основных принципов внутренней организации определенных типов процессоров. Допустим, архитектура Intel Pentium обозначается P5, Pentium II и Pentium III — P6, а не так давно популярных Pentium 4 — NetBurst. Когда компания Intel закрыла P5 для конкурирующих производителей, компания AMD разработала свою архитектуру K7 для Athlon и Athlon XP, а для Athlon 64 — K8.

Но даже процессоры с одинаковой архитектурой могут существенно отличаться друг от друга. Эти различия обусловлены разнообразием процессорных ядер, которые обладают определенным набором характеристик. Наиболее частыми отличиями являются различные частоты системной шины, а также размеры кэша второго уровня и технологические характеристики, по которым изготовлены процессоры. Очень часто смена ядра в процессорах из одного и того же семейства требует также замены процессорного разъема, а это влечет за собой проблемы с совместимостью материнских плат. Но производители постоянно совершенствуют ядра и вносят постоянные, но незначительные изменения в ядре. Такие нововведения называют ревизией ядер и, как правило, обозначают цифробуквенными комбинациями.

Выделяют понятия микроархитектуры и макроархитектуры.

Микроархитектура микропроцессора — это аппаратная организация и логическая структура микропроцессора, регистры, управляющие схемы, арифметико-логические устройства, запоминающие устройства и связывающие их информационные магистрали.

Макроархитектура — это система команд, типы обрабатываемых данных, режимы адресации и принципы работы микропроцессора.

Архитектура современных микропроцессоров представляет собой продукт нескольких независимых групп разработчиков, которые развивают эту архитектуру более 15 лет, добавляя новые возможности к первоначальному набору команд.

В настоящее время доминирующее положение на рынке центральных процессоров занимает семейство процессоров фирмы «Intel» с архитектурой X86.

Это семейство открывают 16-разрядные процессоры 8086 и 8088 с 16/8-битной шиной данных и 20-битной шиной адреса. Второе поколение процессоров представлено процессором 80286, в котором шина адреса была расширена до 24 бит, что позволяло в особом режиме (protected mode — защищенный режим) адресовать до 16 Мбайт физической памяти. Начиная с третьего поколения (Intel 386) архитектура процессоров этого семейства стала 32-битной, а основным режимом работы — защищенный. В новых моделях усовершенствована работа с кешем (Intel 486), появились параллельные конвейеры (Pentium), новые архитектурные блоки (Pentium MMX), появился встроенный кеш второго уровня (P6). Эти изменения сопровождались также добавлением новых возможностей при работе в защищенном режиме: VME (Virtual Mode Extension) у Pentium, PAE (Physical Address Extension) у P6 и др.

В семействе Intel с архитектурой X86 декларируется программная совместимость моделей процессоров сверху вниз. Это значит, что код, написанный для 8086, должен работать и на 80386, и на Pentium 4. С другой стороны, программы, разработанные для более поздних процессоров, могут не работать на более ранних, если в них используются какие-либо специфические особенности новой модели.

Немного истории

Появлением первых микропроцессоров мы обязаны компании Intel Corporation of Santa Clara, которая объявила процессор как «компьютер-на-чипе», это был 4-битный Intel 4004 (ноябрь 1971). В одном кристалле размещалась большая часть компонентов процессора. Фактически процессор был реализован на четырех микросхемах. На базе 4-битного Intel 4004-процессора по заказу японской компании «Busicom» был сделан первый микрокалькулятор.

Процессор был однопрограммным — следующая программы исполнялась только после завершения предыдущей — и неуниверсальным — был предназначен исключительно для вычислительных работ и не мог применяться, например, для обработки текстов.

Почти в то же время появился специализированный микропроцессор 8008, который предназначался для использования в терминале вычислительной машины. Развитием микропроцессоров стало появление универсальных многопрограммных 8-битных Intel 8080 (апрель 1974) и MOS Technology 6502 (сентябрь 1975). Они оба использовались в производстве настольных компьютеров и игровых консолей: первый — Altair 8800, второй — Nintendo NES, Atari 2600, Apple I, Apple II, Commodore 64, Агат и др. В отличие от 4004 эти микропроцессоры использовали раздельные шины адреса и данных, а инструкции и данные хранились в одних и тех же областях памяти. Он имел довольно высокую производительность, позволял адресовать значительный объем памяти, да и спектр его возможностей был существенно шире, чем у предшественников. Это были первые CPU, работающие на основе архитектуры фон Неймана и выполняющие функции арифметико-логического устройства и устройства управления.

Intel 8080 быстро стал общепризнанным стандартом, и многие фирмы начали выпускать его по лицензии. Стали появляться улучшенные версии, например Z80 фирмы «Zilog» или V10 фирмы «NEC», но структура оставалась прежней. Кстати, и сама фирма «Intel» в 1976 г. тоже выпустила модернизированный вариант кри-

сталла 8080 — микропроцессор 8085, который был дополнен несколькими командами и улучшен аппаратно — встроены тактовый генератор и контроллер шины, добавлен простой последовательный порт, увеличена тактовая частота.

Следующим этапом в эволюции центральных процессоров стал выпуск 16-битных Intel 8086/88 (июнь 1978), которые положили начало архитектуре X86 и массовому распространению персональных компьютеров. Процессор обладал весьма высокой по тем временам производительностью и серьезными возможностями, в том числе полной десятичной арифметикой, что позволяло применять его в самых различных областях. В 1980 г. был представлен первый процессор с RISC-архитектурой — IBM 801. По сравнению с CISC-процессорами того времени он имел меньшие размеры и число инструкций, был проще и дешевле в изготовлении. В 1984 г. начали изготовляться первые процессоры VLIW-архитектуры, однако они не получили большого распространения.

В 1978 г. появилось третье поколение микропроцессоров, и опять фирма «Intel» оказалась лидером — кристалл 8086 стал первым микропроцессором, оперирующим 16-разрядными словами данных. Он обладал весьма высокой по тем временам производительностью и серьезными возможностями, в том числе полной десятичной арифметикой, что позволяло применять его в самых различных областях.

В 1979 г. фирма «Intel» упростила процессор 8086, создав микросхему 8088 с 8-разрядной внешней шиной данных.

В 1983 г. «Intel» разработала еще два микропроцессора, представлявших собой усовершенствованные варианты 8086 и 8088, — 80186 и 80188, однако получить широкого распространения они не успели, так как в том же году появился процессор 80286, ставший серьезным шагом вперед. Всего через год на его базе был создан персональный компьютер IBM PC/AT, предоставивший в распоряжение пользователя вычислительные мощности средней ЭВМ. С появлением виртуального режима стало возможным создавать на базе 80286 системы с разделением ресурсов, что раньше было прерогативой больших машин. В микропроцессоре было также реализовано управление памятью.

Дальнейшее развитие процессоров привело к переходу на 32-разрядные модели, позволявшие эффективнее работать с большими числами и адресовать ранее недоступные объемы памяти. В октябре 1985 г. вышел первый 32-битный X86-процессор, Intel 80386, а в 1986 г. появились три новые 32-битные RISC-архитектуры —MIPS, SPARC и PA-RISC, представленные компаниями «MIPS Technolo-

gies», «Sun» и «HP» соответственно. Следующим шагом было появление 64-битных процессоров MIPS R4000 (февраль 1991) и DEC Alpha 21064 (ноябрь 1992). Alpha был также первым CPU, поддерживающим суперскалярность, т.е. возможность исполнять более одной инструкции за такт. Первыми суперскалярными процессорами других архитектур стали Intel Pentium (март 1993), MIPS R8000 (июнь 1994), PA-RISC 7100 (июнь 1994) и 64-битный UltraSPARC (сентябрь 1994).

Следующей вехой в истории центральных процессоров стало динамическое исполнение команд. Заключалось оно в том, что процессор исполнял команды не в том порядке, в котором он их считывал из памяти, а в том, который был более эффективен по времени выполнения, и при этом, конечно же, не нарушал семантики программы. Эта технология была реализована во всех процессорах соответствующих архитектур, начиная с MIPS R10000 (октябрь 1994), PA-RISC 8000 (март 1995), Intel Pentium Pro (ноябрь 1995), Alpha 21264 (декабрь 1998).

После этого появились первые популярные и коммерчески успешные процессоры с архитектурой VLIW —Intel Itanium (октябрь 1999) и Transmeta Crusoe (январь 2000). Затем с некоторым опозданием от других платформ вышли 64-битные расширения для X86, реализованные в процессорах AMD Opteron (технология AMD64, апрель 2003) и Pentium 4 (EM64T, август 2004).

В начале нового тысячелетия развитие центральных процессоров пошло в сторону увеличения количества ядер в одном процессорном корпусе. Практически одновременно вышли двухъядерные CPU всех популярных архитектур: PA-RISC 8800 (февраль 2004), UltraSPARC-IV (февраль 2004), IBM PowerPC G4 (август 2004), MIPS BCM1255 (октябрь 2004), AMD Athlon X2 (апрель 2005), Pentium D (май 2005), Itanium 2 (октябрь 2005), Intel Core 2 Duo (июль 2006). В ноябре 2006 г. появился уже 7-ядерный Cell в составе Sony PlayStation 3. А в серверном процессоре Intel® Xeon® Processor E7-8890 v4 (дата выпуска — июнь 2016) количество ядер составило уже 24.

Классификация микропроцессоров

1. ***По числу БИС в микропроцессорном комплекте*:**

- однокристальные МП;
- многокристальные МП;
- многокристальные секционные МП.

В первую очередь на такое деление повлияли возможности БИС: ограниченное число элементов, выводов корпуса, в то время как МП — довольно сложное устройство, имеющее много логических элементов и требующее большое количество выводов корпуса БИС.

Однокристальный МП получен при реализации всех аппаратных средств МП в виде одной БИС или СБИС. Основные характеристики таких МП зависят от технологии изготовления БИС.

Многокристальные МП получены при разбиении его логической структуры на функционально законченные части и реализации их в виде БИС (рис. 1.4).

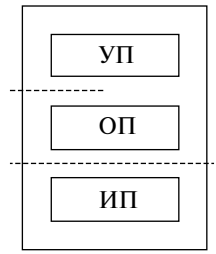


Рис. 1.4. Логические структуры многокристального МП:
ОП — операционный процессор, служит для обработки данных;
УП — управляющий процессор, выполняет функции выборки, декодирования и вычисления адресов операндов, а также генерирует последовательность команд, формирует очередь команд; ИП — интерфейсный процессор, позволяет подключить память к МП

УП, ОП, ИП могут работать автономно (параллельно) и тем самым организовывать конвейер операций.

Многокристальные секционные МП получаются, когда в виде БИС реализуются логические структуры МП при функциональном разбиении ее вертикальными плоскостями (рис. 1.5).

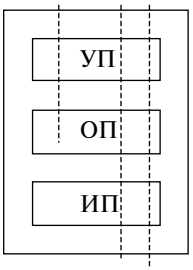


Рис. 1.5. Логические структуры многокристального секционного МП

Пример. Если невозможно реализовать ОП 16 разрядов в одной БИС, его делят на части, реализуемые каждая в своей БИС. Они образуют микропроцессорные секции (например, 4-разрядная секция).

2. ***По назначению***:

- универсальные;
- специальные.

Универсальные МП могут быть применены для решения широкого круга задач. При этом их эффективная производительность мало зависит от проблемной специфики задачи. Как правило, это определяется достаточно широкой универсальной системой команд.

Специальные МП — проблемно ориентированные МП, которые нацелены на ускоренное выполнение определенных функций, что увеличивает эффективную производительность при решении только определенной задачи:

- математические процессоры;
- микроконтроллеры;
- параллельная обработка данных;
- цифровая обработка сигнала — цифровые фильтры и т.д.

Пример. Сравнение входного сигнала одновременно с несколькими эталонами для выделения нужного сигнала.

3. ***По виду обрабатываемых входных сигналов*:**

- цифровые;
- аналоговые.

Требования к аналоговым МП:

- большая разрядность;
- высокая скорость арифметических операций.

4. ***По характеру временной организации работы*:**

- одномагистральные — все устройства имеют одинаковый интерфейс и подключаются к единой информационной магистрали, по которой передаются коды данных, адресов и управляющих сигналов;
- многомагистральные — устройства группами подключаются к своей информационной магистрали, это позволяет осуществить одновременную передачу информационных сигналов по нескольким магистралям. Производительность увеличивается.

5. ***По количеству выполняемых программ*:**

- однопрограммные;
- мультипрограммные.

Мультипрограммные либо могут одновременно выполнять несколько программ, либо имеют средства для поддержки виртуальной мультипрограммности.

Структура типового микропроцессора. Архитектура типичной небольшой вычислительной системы на основе микроЭВМ показана на рис. 1.6. Такая микроЭВМ содержит все пять основных блоков цифровой машины: устройство ввода информации, управляющее устройство (УУ), арифметико-логическое устройство (АЛУ) (входя-

щее в состав микропроцессора), запоминающие устройства (ЗУ) и устройство вывода информации.

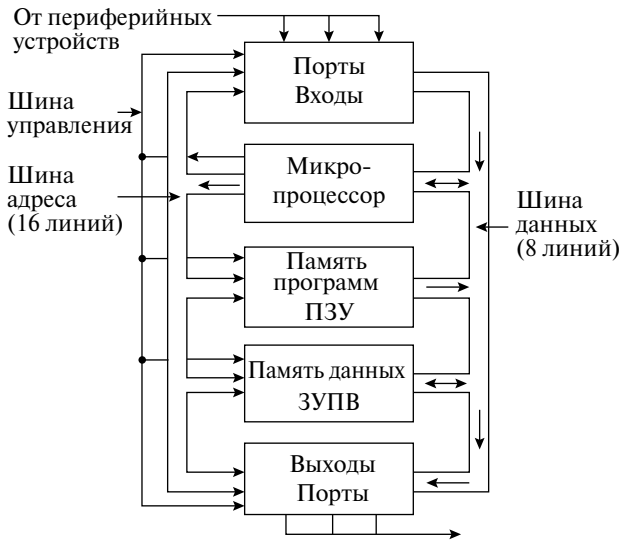


Рис. 1.6. Архитектура типового микропроцессора

Микропроцессор координирует работу всех устройств цифровой системы с помощью шины управления (ШУ). Помимо ШУ, имеется 16-разрядная адресная шина (ША), которая служит для выбора определенной ячейки памяти, порта ввода или порта вывода. По 8-разрядной информационной шине или шине данных (ШД) осуществляется двунаправленная пересылка данных к микропроцессору и от микропроцессора. Важно отметить, что МП может посылать информацию в память микроЭВМ или к одному из портов вывода, а также получать информацию из памяти или от одного из портов ввода.

Постоянное запоминающее устройство (ПЗУ) в микроЭВМ содержит некоторую программу (на практике — программу инициализации ЭВМ). Программы могут быть загружены в запоминающее устройство с произвольной выборкой (ЗУПВ) и из внешнего запоминающего устройства (ВЗУ). Это программы пользователя.

В качестве примера, иллюстрирующего работу микроЭВМ, рассмотрим процедуру, для реализации которой нужно выполнить следующую последовательность элементарных операций.

1. Нажать клавишу с буквой «А» на клавиатуре.
2. Поместить букву «А» в память микроЭВМ.
3. Вывести букву «А» на экран дисплея.

Это типичная процедура ввода-запоминания-вывода, рассмотрение которой дает возможность пояснить принципы использования некоторых устройств, входящих в микроЭВМ.

На рис. 1.7 приведена подробная диаграмма выполнения процедуры ввода-запоминания-вывода. Обратите внимание, что команды уже загружены в первые шесть ячеек памяти. Хранимая программа содержит следующую цепочку команд.

1. Ввести данные из порта ввода 1.
2. Запомнить данные в ячейке памяти 200.
3. Переслать данные в порт вывода 10.

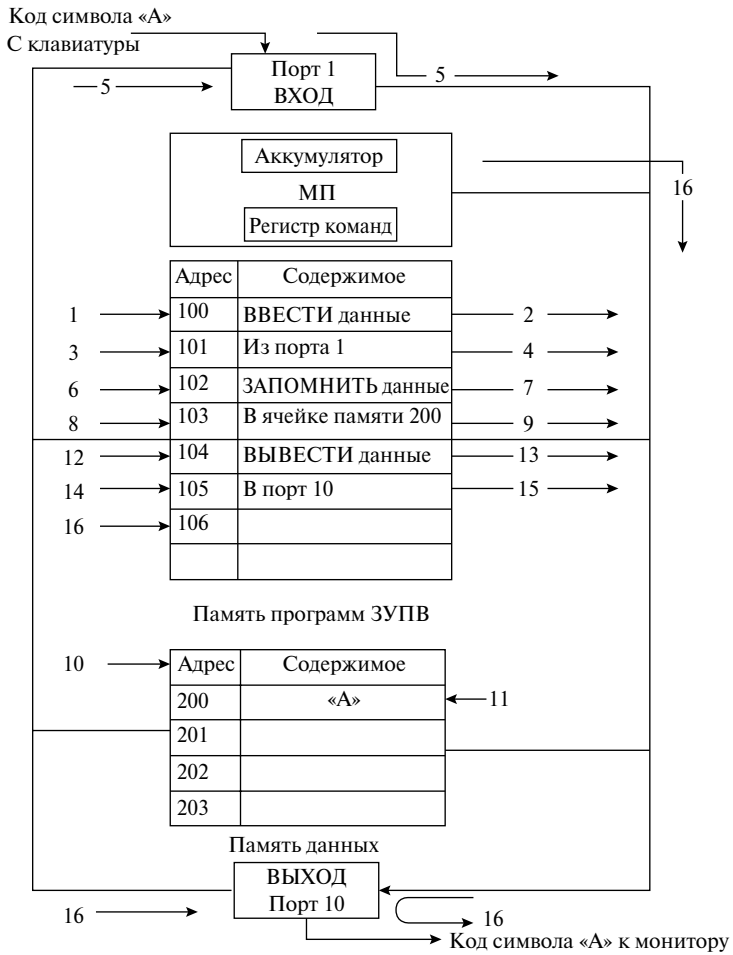


Рис. 1.7. Диаграмма выполнения процедуры ввода-запоминания-вывода

В данной программе всего три команды, хотя на рис. 1.7 может показаться, что в памяти программ записано шесть команд. Это связано с тем, что команда обычно разбивается на части. Первая часть команды 1 в приведенной выше программе — команда ввода данных. Во второй части команды 1 указывается, откуда нужно ввести данные (из порта 1). Первая часть команды, предписывающая конкретное действие, называется *кодом операции* (КОП), а вторая часть — *операндом*. Код операции и операнд размещаются в отдельных ячейках памяти программ. На рис. 1.7 КОП хранится в ячейке 100, а код операнда — в ячейке 101 (порт 1); последний указывает, откуда нужно взять информацию.

В МП на рис. 1.7 выделены еще два новых блока — регистры: аккумулятор и регистр команд.

Рассмотрим прохождение команд и данных внутри микроЭВМ с помощью занумерованных кружков на диаграмме. Напомним, что микропроцессор — это центральный узел, управляющий перемещением всех данных и выполнением операций.

Итак, при выполнении типичной процедуры ввода-запоминания-вывода в микроЭВМ происходит следующая последовательность действий.

1. МП выдает адрес 100 на шину адреса. По шине управления поступает сигнал, устанавливающий память программ (конкретную микросхему) в режим считывания.

2. ЗУ программ пересылает первую команду («Ввести данные») по шине данных, и МП получает это закодированное сообщение. Команда помещается в регистр команд. МП декодирует (интерпретирует) полученную команду и определяет, что для команды нужен операнд.

3. МП выдает адрес 101 на ША; ШУ используется для перевода памяти программ в режим считывания.

4. Из памяти программ на ШД пересылается операнд «Из порта 1». Этот операнд находится в программной памяти в ячейке 101. Код операнда (содержащий адрес порта 1) передается по ШД к МП и направляется в регистр команд. МП теперь декодирует полную команду («Ввести данные из порта 1»).

5. МП, используя ША и ШУ, связывающие его с устройством ввода, открывает порт 1. Цифровой код буквы «А» передается в аккумулятор внутри МП и запоминается. Важно отметить, что при обработке каждой программной команды МП действует согласно микропроцедуре выборки-декодирования-исполнения.

6. МП обращается к ячейке 102 по ША. ШУ используется для перевода памяти программ в режим считывания.

7. Код команды «Запомнить данные» подается на ШД и пересылается в МП, где помещается в регистр команд.

8. МП дешифрирует эту команду и определяет, что для нее нужен операнд. МП обращается к ячейке памяти 103 и приводит в активное состояние вход считывания микросхем памяти программ.

9. Из памяти программ на ШД пересылается код сообщения «В ячейке памяти 200». МП воспринимает этот операнд и помещает его в регистр команд. Полная команда «Запомнить данные в ячейке памяти 200» выбрана из памяти программ и декодирована.

10. Теперь начинается процесс выполнения команды. МП пересылает адрес 200 на ША и активизирует вход записи, относящийся к памяти данных.

11. МП направляет хранящуюся в аккумуляторе информацию в память данных. Код буквы «А» передается по ШД и записывается в ячейку 200 этой памяти. Выполнена вторая команда. Процесс запоминания не разрушает содержимого аккумулятора. В нем по-прежнему находится код буквы «А».

12. МП обращается к ячейке памяти 104 для выбора очередной команды и переводит память программ в режим считывания.

13. Код команды вывода данных пересылается по ШД к МП, который помещает ее в регистр команд, дешифрирует и определяет, что нужен операнд.

14. МП выдает адрес 105 на ША и устанавливает память программ в режим считывания.

15. Из памяти программ по ШД к МП поступает код операнда «В порт 10», который далее помещается в регистр команд.

16. МП дешифрирует полную команду «Вывести данные в порт 10». С помощью ША и ШУ, связывающих его с устройством вывода, МП открывает порт 10, пересылает код буквы «А» (все еще находящийся в аккумуляторе) по ШД. Буква «А» выводится через порт 10 на экран дисплея.

В большинстве микропроцессорных систем (МПС) передача информации осуществляется способом, аналогичным рассмотренному выше. Наиболее существенные различия возможны в блоках ввода и вывода информации.

Подчеркнем еще раз, что именно микропроцессор является ядром системы и осуществляет управление всеми операциями. Его работа представляет последовательную реализацию микропроцедур выборки-дешифрации-исполнения. Однако фактическая последовательность операций в МПС определяется командами, записанными в памяти программ.

Таким образом, в МПС микропроцессор выполняет следующие функции:

- выборку команд программы из основной памяти;
- дешифрацию команд;
- выполнение арифметических, логических и других операций, закодированных в командах;
- управление пересылкой информации между регистрами и основной памятью, между устройствами ввода/вывода;
- отработку сигналов от устройств ввода/вывода, в том числе реализацию прерываний с этих устройств;
- управление и координацию работы основных узлов МП.

Логическая структура микропроцессора

Логическая структура микропроцессора, т.е. конфигурация составляющих микропроцессор логических схем и связей между ними, определяется функциональным назначением. Именно структура задает состав логических блоков микропроцессора и то, как эти блоки должны быть связаны между собой, чтобы полностью отвечать архитектурным требованиям. Срабатывание электронных блоков микропроцессора в определенной последовательности приводит к выполнению заданных архитектурой микропроцессора функций, т.е. к реализации вычислительных алгоритмов. Одни и те же функции можно выполнить в микропроцессорах со структурой, отличающейся набором, количеством и порядком срабатывания логических блоков. Различные структуры микропроцессоров, как правило, обеспечивают их различные возможности, в том числе и различную скорость обработки данных. Логические блоки микропроцессора с развитой архитектурой показаны на рис. 1.8.

При проектировании логической структуры микропроцессоров необходимо рассмотреть:

1) номенклатуру электронных блоков, необходимую и достаточную для реализации архитектурных требований;

2) способы и средства реализации связей между электронными блоками;

3) методы отбора если не оптимальных, то наиболее рациональных вариантов логических структур из возможного числа структур с отличающимся составом блоков и конфигурацией связей между ними.

При проектировании микропроцессора приводятся в соответствие внутренняя сложность кристалла и количество выводов корпуса. Относительный рост числа элементов по мере развития микроэлектронной технологии во много раз превышает относительное

увеличение числа выводов корпуса, поэтому проектирование БИС в виде конечного автомата, а не в виде набора схем, реализующих некоторый набор логических переключательных функций и схем памяти, дает возможность получить функционально законченные блоки и устройства ЭВМ.

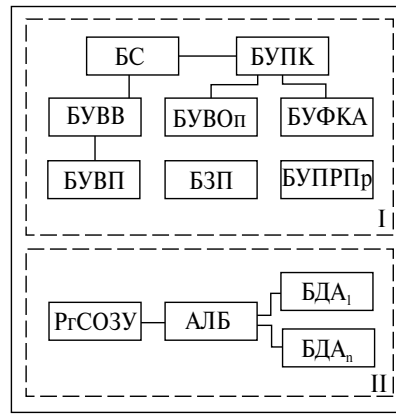


Рис. 1.8. Общая логическая структура микропроцессора:
I — управляющая часть; II — операционная часть; БУПК — блок управления последовательностью команд; БУВОп — блок управления выполнением операций; БУФКА — блок управления формированием кодов адресов; БУВП — блок управления виртуальной памятью; БЗП — блок защиты памяти; БУПРПр — блок управления прерыванием работы процессора; БУВВ — блок управления вводом/выводом; РгСОЗУ — регистровое сверхоперативное запоминающее устройство; АЛБ — арифметико-логический блок; БДА — блок дополнительной арифметики; БС — блок синхронизации

Использование микропроцессорных комплектов БИС позволяет создать микроЭВМ для широких областей применения вследствие программной адаптации микропроцессора к конкретной области применения: изменяя программу работы микропроцессора, изменяют функции информационно-управляющей системы. Поэтому за счет составления программы работы микропроцессоров в конкретных условиях работы определенной системы можно получить оптимальные характеристики последней.

Если уровень только программной «настройки» микропроцессоров не позволит получить эффективную систему, доступен следующий уровень проектирования — микропрограммный. За счет изменения содержимого ПЗУ или программируемой логической мат-

рицы (ПЛМ) можно «настроиться» на более специфичные черты системы обработки информации. В этом случае частично за счет изменения микропрограмм затрагивается аппаратный уровень системы. Технико-экономические последствия здесь связаны лишь с ограниченным вмешательством в технологию изготовления управляющих блоков микроЭВМ.

Изменение аппаратного уровня информационно-управляющей микропроцессорной системы, включающего в себя функциональные БИС комплекта, одновременно с конкретизацией микропрограммного и программного уровней позволяет наилучшим образом удовлетворить требованиям, предъявляемым к системе.

Решение задач управления в конкретной системе чисто аппаратными средствами (аппаратная логика) дает выигрыш в быстродействии, однако приводит к сложностям при модификации системы. Микропроцессорное решение (программная логика) является более медленным, но более гибким решением, позволяющим развивать и модифицировать систему. Изменение технических требований к информационно-управляющей микропроцессорной системе ведет лишь к необходимости перепрограммирования работы микропроцессора. Именно это качество обеспечивает высокую логическую гибкость микропроцессоров, определяет возможность их широкого использования, а значит, и крупносерийного производства.

1.6. Память вычислительных машин

Память (устройство хранения информации, запоминающее устройство) — часть вычислительной машины, физическое устройство или среда для хранения данных, используемая в вычислениях в течение определенного времени.

Память вычислительной машины обеспечивает поддержку одной из функций современногоко мпьютера — способность длительного хранения информации (принципа фон Неймана, заложенного в основу современных вычислительных машин).

Память, как и центральный процессор, является неизменной частью вычислительной машины.

Память в вычислительных устройствах имеет иерархическую структуру и обычно предполагает использование нескольких запоминающих устройств, имеющих различные характеристики.

«Памятью» часто называют один из ее видов — динамическую память с произвольным доступом (DRAM), которая в настоящее время используется в качестве ОЗУ персонального компьютера.

Память вычислительной машины дискретна, состоит из набора последовательных ячеек памяти. Ячейка памяти хранит состояние внешнего воздействия, запись информации. Эти ячейки могут фиксировать самые разнообразные физические воздействия. Они функционально аналогичны обычному электромеханическому переключателю, и информация в них записывается в виде двух четко различимых состояний — «1» и «0» («включено» и «выключено»).

Для записи объема запоминающих устройств в цифровой вычислительной технике и количества памяти, используемой компьютерной программой, применяют термин «количество информации». Единицы количества информации: 1 бит, обозначение — бит, значение — 1; байт, обозначение — Б, значение 1Б = 8 бит (ГОСТ 8.417-2002, табл. А1).

В соответствии с международным стандартом МЭК 60027-2 единицы «бит» и «байт» применяют с приставками СИ (ГОСТ 8.417-2002, табл. 8 и разд. 7) [7]. Исторически сложилась такая ситуация, что с наименованием «байт» некорректно (вместо $1000 = 10^3$ принято $1024 = 2^{10}$) использовали (и используют) приставки СИ: 1 Кбайт = 1024 байт, 1 Мбайт = 1024 Кбайт, 1 Гбайт = 1024 Мбайт и т.д. При этом обозначение Кбайт начинают с прописной буквы в отличие от строчной буквы «к» для обозначения множителя 10^3.

Специальные механизмы обеспечивают доступ (операцию записи и операцию чтения) к состоянию этих ячеек. Процесс доступа происходит под управлением *контроллера памяти* — отдельного специализированного устройства, которое управляет потоком данных от оперативной памяти к процессору.

Контроллер памяти может представлять собой отдельную микросхему или быть интегрирован в микросхему «северный мост» или «микропроцессор».

Также различают операцию стирания памяти — занесение (запись) в ячейки памяти одинаковых значений, обычно 00_{16} или FF_{16}.

Физические основы функционирования. В основе работы запоминающего устройства может лежать любой физический эффект, обеспечивающий приведение системы к двум или более устойчивым состояниям. В современной компьютерной технике часто используются физические свойства полупроводников, когда прохождение тока через полупроводник или его отсутствие трактуются как наличие логических сигналов 0 или 1. Устойчивые состояния, определяемые направлением намагниченности, позволяют использовать для хранения данных разнообразные магнитные материалы. Наличие или отсутствие заряда в конденсаторе также может быть положено в основу

системы хранения. Отражение или рассеяние света от поверхности CD, DVD или Blu-ray-диска также позволяет хранить информацию.

Виды памяти

Различают следующие виды компьютерной памяти.

1. **Процессорная или регистровая память** — это набор быстродействующих регистров, расположенных внутри процессора (непосредственно в АЛУ).

Доступ к значениям, хранящимся в регистрах, как правило, в несколько раз быстрее, чем доступ к ячейкам оперативной памяти (даже если кэш-память содержит нужные данные), но объем оперативной памяти намного превосходит суммарный объем регистров (объем среднего модуля оперативной памяти сегодня составляет 1–4 гигабайт, суммарная «емкость» регистров общего назначения/данных для X86-процессоров, например Intel 80386 и более новых, 8 регистров по 4 байт = 32 байт; в режиме X86 64 — 16 по 8 байт = 128 байт и некоторое количество векторных регистров).

2. **Кэш микропроцессора** — кэш (сверхоперативная память), используемый микропроцессором компьютера для уменьшения среднего времени доступа к компьютерной памяти.

Кэш использует небольшую, очень быструю память (обычно типа SRAM), которая хранит копии часто используемых данных из основной памяти. Если большая часть запросов в память будет обрабатываться кэшем, средняя задержка обращения к памяти будет приближаться к задержкам работы кэша.

Когда процессору нужно обратиться в память для чтения или записи данных, он сначала проверяет, доступна ли их копия в кэше. В случае успеха проверки процессор производит операцию, используя кэш, что значительно быстрее использования более медленной основной памяти.

Большинство современных микропроцессоров для компьютеров и серверов имеют как минимум три независимых кэша: кэш инструкций для ускорения загрузки машинного кода; кэш данных для ускорения чтения и записи данных и буфер ассоциативной трансляции (TLB) для ускорения трансляции виртуальных (логических) адресов в физические как для инструкций, так и для данных. Кэш данных часто реализуется в виде многоуровневого кэша (L1, L2, L3).

Увеличение размера кэш-памяти может положительно влиять на производительность почти всех приложений.

3. **Оперативная память** (англ. *Random Access Memory*, *RAM*, память с произвольным доступом; **ОЗУ** (оперативное запоминающее устрой-

ство) —энергозависимая часть системы компьютерной памяти, в которой во время работы компьютера хранится выполняемый машинный код (программы), а также входные, выходные и промежуточные данные, обрабатываемые процессором.

Обмен данными между процессором и оперативной памятью производится:

- непосредственно (рис. 1.9);
- через сверхбыструю память 0-го уровня — регистры в АЛУ либо при наличии аппаратного кэша процессора — через кэш.
-

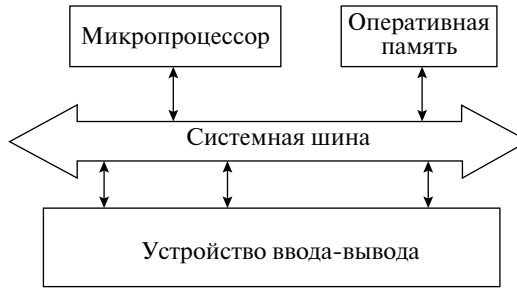


Рис. 1.9. Простейшая схема взаимодействия оперативной памяти с микропроцессором

Содержащиеся в современной полупроводниковой оперативной памяти данные доступны и сохраняются только тогда, когда на модули памяти подается напряжение. Выключение питания оперативной памяти, даже кратковременное, приводит к искажению либо полному разрушению хранимой информации.

В общем случае ОЗУ содержит программы и данные операционной системы и запущенные прикладные программы пользователя и данные этих программ, поэтому от объема оперативной памяти зависит количество задач, которые одновременно может выполнять компьютер под управлением операционной системы.

Оперативное запоминающее устройство, ОЗУ — техническое устройство, реализующее функции оперативной памяти.

ОЗУ может изготавливаться как отдельный внешний модуль или располагаться на одном кристалле с процессором, например в однокристальных ЭВМ или однокристальных микроконтроллерах.

4. **Память долговременного хранения (внешняя память)** — это запоминающие устройства, предназначенные для продолжительного (что не зависит от электропитания) хранения больших объемов информации.

Долговременное хранение в дорогой сверхоперативной и даже оперативной памяти, как правило, невыгодно, поэтому данные такого рода хранятся на накопителях: дисковых, ленточных, флеши т.д.

Наиболее известные запоминающие устройства долговременного хранения — жесткие диски (винчестеры), дискеты (гибкие магнитные диски), CD- или DVD-диски, устройства флеш-памяти.

Для резервного хранения данных создаются библиотеки на съемных носителях, например виртуальная ленточная библиотека или дисковый массив. Емкость съемных носителей большая, но доступ к этим данным занимает значительное время.

Иерархия памяти вычислительных систем

Иерархия памяти — концепция построения взаимосвязи классов разных уровней памяти вычислительных систем на основе иерархической структуры.

Сущность необходимости построения иерархической памяти — необходимость обеспечения вычислительной системы (отдельного компьютера или кластера) достаточным объемом памяти, как оперативной, так и постоянной.

Различные виды памяти образуют иерархию, на различных уровнях которой расположены памяти с отличающимися временем доступа, сложностью, стоимостью и объемом. Возможность построения иерархии памяти вызвана тем, что большинство алгоритмов обращаются в каждый промежуток времени к небольшому набору данных, который может быть помещен в более быструю, но дорогую и поэтому небольшую память. Использование более быстрой памяти увеличивает производительность вычислительного комплекса.

В большинстве современных вычислительных систем используется следующая иерархия памяти.

1. Регистры процессора, организованные в регистровый файл, — наиболее быстрый доступ (порядка 1 такта), но размером лишь в несколько сотен или, редко, тысяч байт.

2. Кэш процессора 1-го уровня (L1) — время доступа порядка нескольких тактов, размером в десятки килобайт.

3. Кэш процессора 2-го уровня (L2) — большее время доступа (от 2 до 10 раз медленнее L1), около полумегабайта или более.

4. Кэш процессора 3-го уровня (L3) — время доступа около сотни тактов, размером в несколько мегабайт (в массовых процессорах используется недавно).

5. ОЗУ системы — время доступа от сотен до, возможно, тысячи тактов, но огромные размеры в несколько гигабайт, вплоть до сотен. Время

доступа к ОЗУ может варьироваться для разных его частей в случае комплексов класса NUMA (с неоднородным доступом в память).

6. Дисковое хранилище — многие миллионы тактов, если данные не были закэшированны или забуферизованы заранее, размеры до нескольких терабайт.

7. Третичная память — задержки до нескольких секунд или минут, но практически неограниченные объемы (ленточные библиотеки).

Получение преимуществ от иерархии памяти требует совместных действий от программиста, аппаратуры и компиляторов (а также базовой поддержки в операционной системе).

Программисты отвечают за организацию передачи данных между дисками и памятью (ОЗУ), используя для этого файловый ввод-вывод; Современные ОС также реализуют это как подкачку страниц.

Аппаратное обеспечение отвечает за организацию передачи данных между памятью и кэшами.

Оптимизирующие *компиляторы* отвечают за генерацию кода, при исполнении которого аппаратура эффективно использует регистры и кэш процессора.

Контрольные вопросы и задания

1. В чем заключается отличие между компьютерами, вычислительной и информационной системами?
2. В чем отличие архитектуры вычислительной машины от ее структуры?
3. Каковы достоинства многоуровневой организации вычислительных систем?
4. Понятие семантического разрыва между уровнями.
5. Опишите структуру многоуровневой вычислительной машины.
6. Опишите принцип работы фон-неймановской вычислительной машины.
7. Перечислите принципы фон Неймана, положенные в основу построения современных вычислительных машин.
8. В чем заключается отличие микроархитектуры процессора от макроархитектуры?
9. Перечислите основные характеристиками процессора.
10. Приведите примеры классификаций микропроцессора.
11. Опишите внутреннюю структуру микропроцессора i8080.
12. Каково назначение устройства управления микропроцессора?
13. Каково назначение регистров микропроцессора?
14. Опишите схему взаимодействия оперативной памяти с микропроцессором.
15. Что лежит в основе работы запоминающего устройства?
16. Каково назначение кэш-памяти? Как влияет объем кэш-памяти на производительность вычислительных машин?

17. Каково назначение оперативной памяти? Как влияет объем оперативной памяти на производительность вычислительных машин?
18. Перечислите наиболее известные запоминающие устройства долговременного хранения.
19. Перечислите уровни памяти вычислительных систем на основе иерархической структуры. Основные преимущества от иерархии памяти.

Глава 2
ЦИФРОВОЙ ЛОГИЧЕСКИЙ УРОВЕНЬ

2.1. Арифметические основы цифровой техники

В цифровых устройствах приходится иметь дело с различными видами информации. Это в чистом виде двоичная информация, такая как включен прибор или выключен, исправно устройство или нет. Информация может быть представлена в виде текстов, и тогда приходится буквы алфавита кодировать при помощи двоичных уровней сигнала. Достаточно часто информация может представлять собой числа. Числа могут быть представлены в различных системах счисления. Форма записи в них чисел существенно различается между собой, поэтому, прежде чем перейти к особенностям представления чисел в цифровой технике, рассмотрим их запись в различных системах счисления.

Системы счисления

Система счисления — это совокупность приемов и правил для представления чисел с помощью цифровых знаков.

Существует множество способов записи чисел цифровыми знаками, но любая применяемая система счисления должна обеспечивать:

- диапазон представления любого числа;
- единственность представления (каждой комбинации символов соответствует только одна величина).

Все системы счисления делятся на позиционные и непозиционные.

В ***непозиционной*** системе счисления значимость цифры в любом месте числа одинакова, т.е. не зависит от позиции расположения. Например, унарная система с одним символом, равным единице. Такая система счислений предназначена для суммарного счета (узелки на «память», зарубки, черточки, счет на пальцах и пр.). Для изображения какого-то числа в этой системе нужно записать число единиц (палочек), равное данному числу. Эта система неэффективна, так как запись числа получается слишком длинной.

Другой пример «почти непозиционной» системы счисления — римская система счета. В римской системе счета используются следующие символы:

I — 1; V — 5; X — 10; L — 50; C — 100; D — 500; M — 1000.

Правила перевода из римской системы счисления в арабскую систему следующие.

Меньшая по величине цифра, стоящая справа от большей цифры, складывается с большей, а меньшая по величине цифра, стоящая слева от большей цифры, вычитается из большей.

Пример перевода из римской системы в арабскую систему счисления:

CCXVVII = 100 + 100 + 10 + 5 + 5 + 1 + 1 = 222;

XIXIV = 10 + (10 – 1) = 19.

Как следует из правила перевода, римская система полностью не является непозиционной. Эта система применяется редко (циферблат, архитектура, история и т.д.).

Позиционные системы счисления — это системы счисления, в которых значение цифры в записи числа N зависит от ее позиции (места). Например, в десятичной системе счисления число 05 обозначает пять единиц, 50 обозначает пять десятков, 500 — пять сотен и т.д.

Основание (базис) системы счисления (q) — это количество используемых знаков или символов для изображения цифр в данной системе счисления.

Возможно бесчисленное множество позиционных систем счисления, так как за основание можно принять любое число и образовать новую систему счисления.

Примеры некоторых позиционных систем счисления и их применение приведены в табл. 2.1.

Таблица 2.1

Примеры позиционных систем счисления

Наименование системы счисления	Основание системы счисления (q)	Используемые цифры	Применение
Двоичная	2	0 и 1	В цифровой вычислительной технике, дискретной математике, программировании и пр.
Троичная	3	Любые три знака: (–, 0, +), (–1, 0, +1), (A, B, C), (X, Y, Z) или три цифры: (1, 2, 3)	В цифровой электронике

Окончание табл. 2.1

Наименование системы счисления	Основание системы счисления (q)	Используемые цифры	Применение
Восьмеричная	8	0, 1, 2, ..., 7	В цифровой вычислительной технике, программировании
Десятичная	10	0, 1, 2, ..., 0	Повсеместное
Шестнадцатеричная	16	0, 1, 2, ..., 9, A, B, C,..., F	В цифровой вычислительной технике, программировании
Шестидесятеричная	60	00, 01, 02, ..., 59	Как единицы измерения времени, измерение углов, координат, долготы и широты

В табл. 2.2 для удобства сопоставления приведены первые 23 числа натурального ряда чисел в различных системах счисления.

Таблица 2.2

Натуральный ряд чисел в различных системах счисления

Десятичная	Двоичная	Восьмеричная	Шестнадцатеричная
0	0000	0	0
1	0001	1	1
2	0010	2	2
3	0011	3	3
4	0100	4	4
5	0101	5	5
6	0110	6	6
7	0111	7	7
8	1000	10	8
9	1001	11	9
10	1010	12	A
11	1011	13	B
12	1100	14	C
13	1101	15	D
14	1110	16	E
15	1111	17	F
16	10000	20	10
17	10001	21	11

Окончание табл. 2.2

Десятичная	Двоичная	Восьмеричная	Шестнадцатеричная
18	10010	22	12
19	10011	23	13
20	10100	24	14
21	10101	25	15
22	10110	26	16
23	10111	27	17

Как видно из табл. 2.2, для записи одного и того же числа в различных системах счисления требуется разное число позиций или разрядов. Например, $14_{10} = 1110_2 = 16_8 = E_{16}$. То есть в десятичной системе счисления число 14 занимает две позиции (два разряда), в двоичной системе счисления — четыре позиции, в шестнадцатеричной системе счисления — одну позицию. Чем меньше основание системы счисления q, тем больше длина числа (длина разрядной сетки).

При заданной длине разрядной сетки ограничивается максимальное по абсолютному значению число, которое можно записать.

Пусть длина разрядной сетки равна положительному числу N, максимальное число равно:

$$A_{(q)\max} = q^N - 1.$$

Например, при $N = 8$:

$$A_{(10)\max} = 10^8 - 1 = 9999999_{(10)};$$
$$A_{(2)\max} = 2^8 - 1 = 256 - 1 = 257_{(10)} = 1111111_{(2)};$$
$$A_{(16)\max} = 16^8 - 1 = 4294967296 - 1 = 4294967295_{(10)} = FFFFFFF_{(16)}.$$

Таким образом, при одинаковой длине разрядной сетки $N = 8$ максимальное по абсолютному значению $A_{(16)\max} > A_{(10)\max} > A_{(2)\max}$, т.е. чем больше q, тем больше $A_{(q)\max}$.

Перевод в позиционных системах счисления

1. **Перевод в десятичную систему счисления.**

Любое число N в позиционной системе счисления можно представить в виде полинома

$$N_q = a_{m-1}q^{m-1} + a_{m-2}q^{m-2} + ... + a_1q^1 + a_0q^0 + a_{-1}q^{-1} + a_{-2}q^{-2} + ... + a_{-k}q^{-k},$$

где q — основание системы счисления; a — элемент числа N_q, принимающий значения 0, 1, 2, ..., (q–1); m — номер разряда целой части, отсчитываемый от нулевого; k — число цифр в дробной части числа.

Например, число $253{,}24_{10}$ в обычной десятичной форме ($q = 10$) можно представить следующим образом:

$$N_{10} = 253{,}24(_{10}) = 2 \cdot 102 + 5 \cdot 101 + 3 \cdot 100 + 2 \cdot 10^{-1} + 4 \cdot 10^{-2}.$$

Пример 2.1. Двоичное число 1101,01(2) перевести в десятичную систему счисления.

В двоичной системе счисления для представления чисел используются две цифры 0 и 1 и двоичное число $1101{,}01_2$ ($q = 2$) можно определить следующим образом:

$$N_2 = 1101{,}01_2 = 1 \cdot 23 + 1 \cdot 2^2 + 0 \cdot 2^1 + 1 \cdot 2^0 + 0 \cdot 2^{-1} + 1 \cdot 2^{-2} = 8 + 4 + \\ + 0 + 1 + 0 + 1/4 = 14{,}25_{10}.$$

Если по правилам десятичной арифметики выполнить действия в правой части приведенного равенства, то получим десятичный эквивалент двоичного числа:

$$1101{,}01_2 = 8 + 4 + 0 + 1 + 0 + 1/4 = 14{,}25_{10}.$$

Пример 2.2. Восьмеричное число $53{,}2_8$ ($q = 8$) перевести в десятичную систему счисления:

$$N_8 = 53{,}2_8 = 5 \cdot 8^1 + 3 \cdot 8^0 + 2 \cdot 8^{-1} + 4 \cdot 8^{-2} = 40 + 3 + 2/8 = 43{,}25_{10}.$$

Пример 2.3. Шестнадцатеричное число $AF7{,}8_{16}$ ($q = 16$) перевести в десятичную систему счисления.

$$N_{16} = AF{,}8_{16} = 10 \cdot 16^2 + 15 \cdot 16^1 + 7 \cdot 16^0 + 8 \cdot 16^{-1} = 2560 + 240 + \\ + 7 + 8/16 = 2807{,}25_{10}.$$

2. **Перевод чисел из десятичной системы счисления в произвольную систему счисления с основанием *q*.**

Правила перевода целой части десятичного числа следующие. Целую часть десятичного числа необходимо последовательно делить на q (основание произвольной системы счисления) до тех пор, пока десятичное число не станет равно нулю. Остатки, полученные при делении и записанные в последовательности, начиная с последнего остатка, являются цифрами числа q-ричной системы счисления.

Правила перевода дробной части десятичного числа следующие. Дробную часть десятичного числа необходимо последовательно умножать на q (основание произвольной системы) и отделять целую часть до тех пор, пока она не станет равной нулю или будет достигнута заданная точность перевода.

Целые части результатов умножения в порядке, соответствующем их получению, составляют число в новой системе.

Пример 2.4. Число $26{,}625_{10}$ перевести в двоичную систему счисления.

Переводим целую часть числа:

- 26 : 2 = 13, остаток равен нулю;
- 13 : 2 = 6, остаток равен единице;
- 6 : 2 = 3, остаток равен нулю;
- 3 : 2 = 1, остаток равен единице;
- 1 : 2 = 0, остаток равен единице.

Десятичное число стало равно нулю, деление закончено. Переписываем все остатки снизу вверх и получаем двоичное число 11010_2.

Переводим дробную часть числа:

- $0{,}625 \cdot 2 = 1{,}250$, целая часть равна единице;
- $0{,}250 \cdot 2 = 0{,}500$, целая часть равна единице;
- $0{,}500 \cdot 2 = 1{,}000$, целая часть равна единице;
- $0{,}000 \cdot 2 = 0{,}000$, целая часть равна нулю.

Целая часть стала равной нулю. Переписываем целые части результатов умножения сверху вниз и получаем двоичное число $0{,}1010_2$.

Пример 2.5. Число $70{,}05_{10}$ перевести в восьмеричную систему счисления с точностью по 4-й разряд.

Переводим целую часть числа:

- 70 : 8 = 8, остаток равен 6;
- 8 : 8 = 1, остаток равен нулю;
- 1 : 8 = 0, остаток равен единице.

Десятичное число стало равно нулю, деление закончено. Переписываем все остатки снизу вверх и получаем восьмеричное число 106_8.

Переводим дробную часть числа:

- $0{,}05 \cdot 8 = 0{,}40$, целая часть равна нулю;
- $0{,}40 \cdot 8 = 3{,}20$, целая часть равна 3;
- $0{,}30 \cdot 8 = 2{,}40$, целая часть равна 2;
- $0{,}40 \cdot 8 = 3{,}20$, целая часть равна 3.

Целая часть стала не равна нулю, получается бесконечный ряд, процесс перевода заканчиваем, так как достигнута заданная точность. Переписываем целые части результатов умножения сверху вниз и получаем восьмеричное число $0{,}0323_{8.}$

Пример 2.6. Число $76{,}05_{10}$ перевести в шестнадцатеричную систему счисления с точностью по 4-й разряд.

Переводим целую часть числа:

- 76 : 16 = 4, остаток равен $12 \rightarrow$ C;
- 4 : 16 = 0, остаток равен 4.

Десятичное число стало равно нулю, деление закончено. Переписываем все остатки снизу вверх и получаем шестнадцатеричное число $4C_{16}$.

Переводим дробную часть числа:

- $0{,}05 \cdot 16 = 0{,}80$, целая часть равна 0;
- $0{,}80 \cdot 16 = 12{,}80$, целая часть равна $12 \rightarrow$ C;
- $0{,}80 \cdot 16 = 12{,}80$, целая часть равна $12 \rightarrow$ C;
- $0{,}80 \cdot 16 = 12{,}80$, целая часть равна $12 \rightarrow$ C.

Целая часть стала не равна нулю, получается бесконечный ряд, процесс перевода заканчиваем, так как достигнута заданная точность. Переписываем целые части результатов умножения сверху вниз и получаем шестнадцатеричное число $0{,}0CCC_{16}$.

Пример 2.7. Число 6610 перевести в произвольную систему счисления, например с основанием $q = 5$.

Переводим целую часть числа:

- $66 : 5 = 13$, остаток равен единице;
- $13 : 5 = 2$, остаток равен 3;
- $2 : 5 = 0$, остаток равен 2.

Десятичное число стало равно нулю, деление закончено. Переписываем все остатки снизу вверх и получаем пятеричное число 231_5.

3. **Перевод из двоичной системы в восьмеричную и шестнадцатеричную.**

Для этого типа операций существует упрощенный алгоритм.

Перевод целой части

Число 2 возводится в ту степень, которая необходима для получения основания системы, в которую требуется перевести. Для восьмеричной системы ($8 = 23$) получаем число 3 (триада), для шестнадцатеричной системы ($16 = 24$) получаем число 4 (тетрада).

Разбиваем переводимое число на количество цифр, равное 3 для восьмеричной системы и равное 4 для шестнадцатеричной системы счисления.

Преобразуем триады по таблице триад восьмеричной системы и тетрады по таблице тетрад для шестнадцатеричной системы счисления.

Таблица 2.3

Таблица триад и тетрад

Триады

0	1	2	3	4	5	6	7
000	001	010	011	100	101	110	111

Тетрады

0	1	2	3	4	5	6	7
0000	0001	0010	0011	0100	0101	0110	0111
8	9	A	B	C	D	E	F
1000	1001	1010	1011	1100	1101	1110	1111

Пример 2.8. Двоичное число 101110_2 перевести в восьмеричную и шестнадцатеричную системы счисления.

- восьмеричная — $101\,110 \rightarrow 56_8$;
- шестнадцатеричная — $0010\,1110 \rightarrow 2E_{16}$.

Перевод дробной части

Алгоритм перевода дробной части из двоичной системы счисления в восьмеричную и шестнадцатеричную системы счисления аналогичен алгоритму целых частей числа, но разбивка на триады и тетрады идет вправо от десятичной запятой, недостающие разряды дополняются нулями справа.

Пример 2.9. Перевести $11101{,}01011_2$ в восьмеричную и шестнадцатеричную системы счисления:

- восьмеричная — $011\,101,\ 010\,110 \rightarrow 35{,}26_8$;
- шестнадцатеричная — $0001\,1101{,}0101\,1000 \rightarrow 1D{,}58_{16}$.

4. **Перевод из восьмеричной и шестнадцатеричной систем в двоичную.**

Для этого типа операций существует упрощенный алгоритм-перевертыш:

Для восьмеричной — преобразуем по таблице в триплеты:

- $0 \rightarrow 000$ $4 \rightarrow 100$;
- $1 \rightarrow 001$ $5 \rightarrow 101$;
- $2 \rightarrow 010$ $6 \rightarrow 110$;
- $3 \rightarrow 011$ $7 \rightarrow 111$.

Для шестнадцатеричной — преобразуем по таблице в квартеты:

- $0 \rightarrow 0000$, $4 \rightarrow 0100$, $8 \rightarrow 1000$, $C \rightarrow 1100$;
- $1 \rightarrow 0001$, $5 \rightarrow 0101$, $9 \rightarrow \rightarrow 1001$, $D \rightarrow 1101$;
- $2 \rightarrow 0010$, $6 \rightarrow 0110$, $A \rightarrow 1010$, $E \rightarrow 1110$;
- $3 \rightarrow 0011$, $7 \rightarrow 0111$, $B \rightarrow 1011$, $F \rightarrow 1111$.

Пример 2.10. Перевести восьмеричное число 243_8 и шестнадцатеричное число $7C_{16}$ в двоичную систему счисления:

- $243_8 \rightarrow 110\,100\,011_2$;
- $7C_{16} \rightarrow 1111\,1100_2$.

Двоичная арифметика

Сложение. Таблица сложения двоичных чисел проста.

- $0 + 0 = 0$;
- $0 + 1 = 1$;
- $1 + 0 = 1$;
- $1 + 1 = 10$;
- $1 + 1 + 1 = 11$.

При сложении двух единиц происходит переполнение разряда и производится перенос в старший разряд. Переполнение разряда

наступает тогда, когда величина числа в нем становится равной или большей основания.

Пример 2.11. Выполнить сложение в двоичной системе счисления:

	1			1	1			Перенос в старший разряд
+	1	1	0	0	0	1	=	49 — первое слагаемое
		1	1	0	1	1	=	27 — второе слагаемое
1	0	0	1	1	0	0	=	76 — сумма

Двоичное вычитание. Рассмотрим правила вычитания меньшего числа из большего. В простейшем случае для каждого разряда правила двоичного вычитания имеют вид:

	0		1		1	?	0		
–	0	–	0	–	1		1		
	0		1		0		1		

Когда производится вычитание (0 – 1), осуществляется заем из более старшего разряда. Знак вопроса означает, что разряд уменьшаемого изменяется в результате займа по правилу следующему. При вычитании (0 – 1) в разряде разности получается единица, разряды уменьшаемого, начиная со следующего, изменяются на противоположные (инвертируются) до первой встречной единицы (включительно). После этого производится вычитание из измененных разрядов уменьшаемого.

Рассмотрим пример вычитания многоразрядных чисел (из большего числа вычитается меньшее).

Пример 2.12. Вычитание в двоичной системе счисления:

	0	1	1	1				Изменение уменьшаемого в результате займа
–	1	1	0	0	0	1	=	49 — уменьшаемое
		1	1	0	1	1	=	27 — вычитаемое
		1	0	1	1	0	=	22 — разность

Умножение. Операция умножения выполняется с использованием таблицы умножения по обычной схеме (применяемой в десятичной системе счисления) с последовательным умножением множимого на очередную цифру множителя.

Пример 2.13. Умножение в двоичной системе счисления:

```
   *1011
    101
   1011
  1011
 110111.
```

Деление. При делении столбиком приходится в качестве промежуточных результатов выполнять действия умножения и вычитания.

Запись десятичных чисел (двоично-десятичный код)

Иногда бывает удобно хранить числа в памяти процессора в десятичном виде (например, для вывода на экран дисплея). Для записи таких чисел используются двоично-десятичные коды. Для записи одного десятичного разряда используется четыре двоичных бита (тетрады). При помощи четырех битов можно закодировать шестнадцать цифр ($2^4 = 16$). Лишние комбинации в двоично-десятичном коде являются запрещенными. Соответствие двоично-десятичного кода и десятичных цифр приведено в табл. 2.4.

Таблица 2.4

Соответствие двоично-десятичного кода и десятичных цифр

Двоично-десятичный код				Десятичный код
0	0	0	0	0
0	0	0	1	1
0	0	1	0	2
0	0	1	1	3
0	1	0	0	4
0	1	0	1	5
0	1	1	0	6
0	1	1	1	7
1	0	0	0	8
1	0	0	1	9

Остальные комбинации двоичного кода в тетраде являются запрещенными.

Пример 2.14. Записать двоично-десятичный код числа 1258_{10}:

$1258_{10} = 0001\,0010\,0101\,1000_2$.

В первой тетраде записана цифра 1, во второй — 2, в третьей — 5, а в последней тетраде записана цифра 8. В данном примере для записи числа 1258 потребовалось четыре тетрады. Количество ячеек памяти микропроцессора зависит от его разрядности. При 16-разрядном процессоре все число уместится в одну ячейку памяти.

Пример 2.15. Записать двоично-десятичный код числа 589_{10}:

$589_{10} = 0000\,0101\,1000\,1001_2$.

В данном примере для записи числа достаточно трех тетрад, но ячейка памяти 16-разрядная. Поэтому старшая тетрада заполняется нулями. Они не изменяют значение цифры.

При записи десятичных чисел часто требуется записывать знак числа и десятичную запятую (в англоязычных странах — точку). Двоично-десятичный код часто применяется для набора телефонного номера или набора кодов телефонных служб. В этом случае кроме десятичных цифр часто применяются символы «*» или «#». Для записи этих символов в двоично-десятичном коде применяются запрещенные комбинации (табл. 2.5).

Таблица 2.5

Соответствие двоично-десятичного кода и дополнительных символов

Двоично-десятичный код				Дополнительный символ
1	0	1	0	* (звездочка)
1	0	1	1	# (решетка)
1	1	0	0	+ (плюс)
1	1	0	1	– (минус)
1	1	1	0	, (десятичная запятая)
1	1	1	1	Символ гашения

Достаточно часто в памяти процессора для хранения одной десятичной цифры выделяется одна ячейка памяти (8-, 16- или 32-разрядная). Это делается для повышения скорости работы программы. Для того чтобы отличить такой способ записи двоично-десятичного числа от стандартного, способ записи десятичного числа, как это показано в примере, называется упакованной формой двоично-десятичного числа.

Пример 2.16. Записать неупакованный двоично-десятичный код числа 1258_{10} для 8-разрядного процессора:

1258 00000001
00000010
00000101
00001000

В первой строке записана цифра 1, во второй — 2, в третьей — 5, а в последней строке записана цифра 8. В данном примере для записи числа 1258 потребовалось четыре строки (ячейки памяти).

Суммирование двоично-десятичных чисел. Суммирование двоично-десятичных чисел можно производить по правилам обычной двоичной арифметики, а затем производить двоично-десятичную коррекцию. Двоично-десятичная коррекция заключается в проверке каждой тетрады на допустимые коды. Если в какой-либо тетраде обнаруживается запрещенная комбинация, то это говорит о переполнении. В этом случае необходимо произвести двоично-десятич-

ную коррекцию. Двоично-десятичная коррекция заключается в дополнительном суммировании числа шесть (число запрещенных комбинаций) с тетрадой, в которой произошло переполнение или произошел перенос в старшую тетраду. Приведем пример:

$$\begin{array}{r} 0\ 0\ 0\ 1 \\ +\ 0\ 0\ 0\ 1 \\ \hline 0\ 0\ 1\ 0 \end{array} \qquad \begin{array}{r} 1\ 0\ 0\ 0 \\ 0\ 0\ 1\ 1 \\ \hline 1\ 0\ 1\ 1 \end{array}$$

Во второй тетраде обнаружилась запрещенная комбинация. Проводим двоично-десятичную коррекцию — суммируем число шесть со второй тетрадой:

$$\begin{array}{r} 0\ 0\ 1\ 0 \\ +\ 0\ 0\ 0\ 0 \\ \hline 0\ 0\ 1\ 1 \end{array} \qquad \begin{array}{r} 1\ 0\ 1\ 1 \\ 0\ 1\ 1\ 0 \\ \hline 0\ 0\ 0\ 1 \end{array}$$

Формы представления в ЭВМ числовых данных

В математике используются две формы записи чисел: естественная (число записывается в естественном натуральном виде) и нормальная (запись числа может быть различной в зависимости от ограничений, накладываемых на форму).

Примеры *естественной формы* записи чисел:

15300 — целое число; 0,000564 — правильная дробь; 6,4540 — неправильная дробь.

Пример нормальной формы записи одного и того же числа 25340 в зависимости от ограничений, накладываемых на нормальную форму:

$25340 = 2{,}534 \cdot 10^4 = 0{,}2534 \cdot 10^5 = 2534000 \cdot 10^{-2}$ и т.д.

В вычислительной технике при естественном представлении чисел устанавливаются длина разрядной сетки, а также фиксированное распределение дробной и целой частей. Поэтому такой способ представления чисел называется с *фиксированной запятой*.

Представление числа в нормальной форме называют представление с *плавающей запятой* (положение запятой меняется).

С числами, представленными в форме с плавающей запятой, работают в основном универсальные ЭВМ, а специализированные ЭВМ — с фиксированной запятой, но целый ряд машин работает с числами в этих двух форматах.

Характер программирования зависит от способа представления чисел. Так, при написании программ для ЭВМ, работающих в системе

с фиксированной запятой, необходимо отслеживание положения запятой, а для выполнения операций с плавающей запятой требуется большее число микроопераций, что снижает быстродействие ЭВМ.

Фиксированная запятая (точка). В современных ЭВМ способ представления чисел с фиксированной запятой в вычислительной технике используется преимущественно для представления целых чисел.

Так как числа бывают положительными и отрицательными, то в разрядной сетке при их машинном представлении один или два разряда (для модифицированных кодов) отводятся под знак числа, а остальные разряды образуют поле числа. В знаковые разряды, которые могут располагаться как в начале, так и в конце числа, записывается информация о знаке числа. Знак «+» кодируется нулем, знак «–» кодируется единицей. Для модифицированных кодов знак «+» кодируется двумя нулями, знак «–» кодируется двумя единицами. Модифицированные коды введены для обнаружения неправильного результата вычислений, т.е. когда результат превышает максимальный размер разрядной сетки и необходим перенос из значащего разряда.

Например, в результате выполнения операций в знаковом разряде число «01» свидетельствует о положительном переполнении разрядной сетки, а число «10» — об отрицательном переполнении разрядной сетки.

Поле числа имеет постоянное число разрядов — n. Диапазон представления целых чисел ограничивается значениями — $(2^n - 1)$ и + $(2^n - 1)$.

Например, в двоичном коде, используя 6-разрядную сетку, число 7 в форме с фиксированной запятой можно представить в виде

$$0.00111_2,$$

где цифра левее точки — это знак числа, а пять цифр правее точки — мантисса числа в прямом коде.

Здесь подразумевается, что запятая фиксирована правее младшего разряда, а точка в изображении числа в данном случае просто разделяет знаковый бит от мантиссы числа. В дальнейшем часто будет использоваться в примерах такой вид представления числа в машинной форме. Можно использовать и другую форму представления числа в машинной форме:

$$[0]00111_2,$$

где знаковый разряд выделяется квадратными скобками.

Количество разрядов в разрядной сетке, отведенное для изображения мантиссы числа, определяет диапазон и точность представле-

ния числа с фиксированной запятой. Максимальное по абсолютной величине двоичное число изображается единицами во всех разрядах, исключая знаковый, т.е. для целого числа

$$|A|\max = (2^{(n-1)} - 1),$$

где n — полная длина разрядной сетки. В случае 16-разрядной сетки

$$|A|\max = (2^{(16-1)} - 1) = 32767_{10},$$

т. е. диапазон представления целых чисел в этом случае будет от $+32767_{10}$ до -32767_{10}.

Для случая, когда запятая фиксируется правее младшего разряда мантиссы, т.е. для целых чисел, числа, у которых модуль больше, чем $(2^{(n-1)} - 1)$ и меньше единицы, не представляются в форме с фиксированной запятой. Числа по абсолютной величине меньше единицы младшего разряда разрядной сетки называются в этом случае машинным нулем. Отрицательный ноль запрещен.

В некоторых случаях, когда можно оперировать только модулями чисел, вся разрядная сетка, включая самый старший разряд, отводится для представления числа, что позволяет расширить диапазон изображения чисел.

Представление отрицательных чисел в формате с фиксированной запятой. В компьютерах в целях упрощения выполнения арифметических операций применяются специальные двоичные коды для представления отрицательных чисел: *обратный* и *дополнительный*. При помощи этих кодов упрощается определение знака результата операции при алгебраическом сложении. Операция вычитания (или алгебраического сложения) сводится к арифметическому сложению операндов, облегчается выработка признаков переполнения разрядной сетки. В результате упрощаются устройства компьютера, выполняющие арифметические операции.

Известно, что одним из способов выполнения операции вычитания является замена знака вычитаемого на противоположный и прибавление его к уменьшаемому:

$$A - B = A + (-B).$$

Этим операцию арифметического вычитания заменяют операцией алгебраического сложения, которую можно выполнить при помощи двоичных сумматоров.

Для машинного представления отрицательных чисел используют коды прямой, дополнительный, обратный. Упрощенное определение этих кодов может быть дано следующим образом. Если число A в обычном двоичном коде — *прямом* двоичном коде изобразить как

$$[A]_{\text{пр}} = 0.a_n\, a_{n-1}\, a_{n-2}...a_1\, a_0,$$

тогда число $-A$ в этом же коде представляется как

$$[-A]_{\text{пр}} = 1.a_n\, a_{n-1}\, a_{n-2}...a_1\, a_0,$$

а в *обратном* (инверсном) коде это число будет иметь вид

$$[-A]_{\text{об}} = 1.a_n\, a_{n-1}\, a_{n-2}...a_1\, a_0,$$

где

$a_i = 1$, если $a_i = 0$,

$a_i = 0$, если $a_i = 1$,

a_i — цифра i-го разряда двоичного числа. Следовательно, при переходе от прямого кода к обратному все цифры разрядов мантиссы числа инвертируются.

Тогда число $-A$ в *дополнительном* коде изображается в виде

$$[-A]_{\text{доп}} = [-A]_{\text{об}} + 1.$$

Таким образом, для получения дополнительного кода отрицательных чисел нужно сначала инвертировать цифровую часть исходного числа, в результате чего получается его обратный код, а затем добавить единицу в младший разряд цифровой части числа.

Дополнительный код некоторого числа получается его заменой на новое число, дополняющее его до числа, равного весу разряда, следующего за самым старшим разрядом разрядной сетки, используемой для представления мантиссы числа в формате с фиксированной запятой. Поэтому такой код числа называется дополнительным.

Представим, что мы имеем только два разряда для представления чисел в десятичной системе счисления. Тогда максимальное число, которое можно изобразить, будет 99, а вес третьего несуществующего старшего разряда будет 102, т.е. 100. В таком случае для числа 20 дополнительным будет число 80, которое дополняет 20 до 100 (100 – – 20 = 80). Следовательно, по определению вычитание

$$50 - 20 = 30$$

можно заменить на сложение:

$$50 + 80 = 1\,30.$$

Здесь старшая единица выходит за пределы выделенной разрядной сетки, в которой остается только число 30, т.е. результат вычитания из 50 числа 20.

А теперь рассмотрим похожий пример для чисел, представленных 4-разрядным двоичным кодом. Найдем дополнительное число для $0010_2 = 2_{10}$. Надо из [1]0000 вычесть [0]0010, получим [0]1110, которое и является дополнительным кодом 2. Разряд, изображенный в квадратных скобках, на самом деле не существует. Но так как у нас 4-разрядная сетка, то выполнить такое вычитание в принципе невозможно, а тем более, мы стараемся избавиться от вычитания. Поэтому дополнительный код числа получают способом, описанным ранее, т.е. сначала получают обратный код числа, а затем прибавляют к нему единицу. Проделав все это с нашим числом (2), нетрудно убедиться, что получится аналогичный ответ.

Подчеркнем, что дополнительный и обратный коды используются только для представления отрицательных двоичных чисел в форме с фиксированной запятой. Положительные числа в этих кодах не меняют своего изображения и представляются, как в прямом коде.

Таким образом, цифровые разряды отрицательного числа в прямом коде остаются неизменными, а в знаковой части записывается единица.

Рассмотрим простые примеры.

Семерка в прямом коде представляется так:

$$[7]_{\text{пр}} = 0.000111_2.$$

Число -7 в прямом коде

$$[-7]_{\text{пр}} = 1.000111_2,$$

а в обратном коде будет иметь вид

$$[-7]_{\text{об}} = 1.111000_2,$$

т.е. единицы заменяются нулями, а нули — единицами. То же число в дополнительном коде будет

$$[-7]_{\text{доп}} = 1.111001_2.$$

Рассмотрим еще раз, как процедура вычитания при помощи представления вычитаемого в дополнительном коде сводится к процедуре сложения. Вычтем из 10 число 7 : $10 - 7 = 3$. Если оба операнда представлены в прямом коде, то процедура вычитания выполняется так:

$$\begin{array}{r} 0.001010 \\ \underline{-1.000111} \\ 0.000011 = 3_{10}. \end{array}$$

А если вычитаемое, т.е. -7, представить в дополнительном коде, то процедура вычитания сводится к процедуре сложения:

$$\begin{array}{l} \quad 0.001010 \\ + \underline{1.111001} \\ 10.000011 = 3_{10}. \end{array}$$

В настоящее время в компьютерах для представления отрицательных чисел в формате с фиксированной запятой обычно используется дополнительный код.

Вещественные числа

Числовые величины, которые могут принимать любые значения (целые и дробные), называются **вещественными числами**.

Вещественные числа в памяти компьютера представляются в форме с плавающей точкой. Форма с плавающей точкой использует представление вещественного числа R в виде произведения мантиссы m на основание системы счисления p в некоторой целой степени n, которую называют порядком:

$$R = m \cdot p^n.$$

Например, число 25,324 можно записать в таком виде: $0.25324 \cdot 10^2$.

Здесь $m = 0.25324$ — мантисса; $n = 2$ — порядок. Порядок указывает, на какое количество позиций и в каком направлении должна «переплыть», т.е. сместиться, десятичная точка в мантиссе. Отсюда название «плавающая точка».

Однако справедливы и следующие равенства:

$$25{,}324 = 2{,}5324 \cdot 10^1 = 0{,}0025324 \cdot 10^4 = 2532{,}4 \cdot 10^{-2} \text{ и т.п.}$$

Получается, что представление числа в форме с плавающей точкой неоднозначно? Чтобы не было неоднозначности, в ЭВМ ***используют нормализованное представление числа в форме с плавающей точкой***. Мантисса в нормализованном представлении должна удовлетворять условию

$$0{,}1_p <= m < 1_p.$$

Иначе говоря, мантисса меньше единицы и первая значащая цифра — не ноль. Значит, для рассмотренного числа нормализованным представлением будет $0.25324 \cdot 10^2$. В разных типах ЭВМ применяются различные варианты представления чисел в форме с плавающей точкой. Для примера рассмотрим один из возможных. Пусть в памяти компьютера вещественное число представляется в форме с плавающей точкой в двоичной системе счисления ($p = 2$) и занимает ячейку размером 4 байт. В ячейке должна содержаться следу-

ющая информация о числе: знак числа, порядок и значащие цифры мантиссы. Вот как эта информация располагается в ячейке:

± машинный порядок	М А Н Т И С С А		
1-й байт	2-й байт	3-й байт	4-й байт

В старшем бите 1-го байта хранится знак числа. В этом разряде 0 обозначает плюс, 1 — минус. Оставшиеся 7 бит первого байта содержат машинный порядок. В следующих трех байтах хранятся значащие цифры мантиссы.

В семи двоичных разрядах помещаются двоичные числа в диапазоне от 0000000 до 1111111. В десятичной системе это соответствует диапазону от 0 до 127, всего 128 значений. Знак порядка в ячейке не хранится. Но порядок, очевидно, может быть как положительным, так и отрицательным. Разумно эти 128 значений разделить поровну между положительными и отрицательными значениями порядка. В таком случае между машинным порядком и истинным (назовем его математическим) устанавливается следующее соответствие:

Машинный порядок	0	1	2	3	...	64	65	...	125	126	127
Математический порядок	–64	–63	–62	–61	...	0	1	...	61	62	63

Если обозначить машинный порядок Mp, а математический — p, то связь между ними выразится такой формулой:

$$Mp = p + 64.$$

Итак, машинный порядок смещен относительно математического на 64 единицы и имеет только положительные значения. При выполнении вычислений с плавающей точкой процессор это смещение учитывает.

Полученная формула записана в десятичной системе. Поскольку $64_{10}=40_{16}$ (проверьте!), то в шестнадцатеричной системе формула примет вид

$$Mp_{16} = p_{16} + 40_{16}.$$

И наконец, в двоичной системе:

$$Mp_2 = p_2 + 100\,0000_2.$$

Теперь мы можем записать внутреннее представление числа 25,324 в форме с плавающей точкой.

1. Переведем его в двоичную систему счисления с 24 значащими цифрами.

$25{,}324_{10} = 11001{,}01010010111100011101_2$.

2. Запишем в форме нормализованного двоичного числа с плавающей точкой:

$0{,}11001010100101111100011101 \cdot 10^{101}$.

Здесь мантисса, основание системы счисления ($2_{10} = 10_2$) и порядок ($5_{10}=101_2$) записаны в двоичной системе.

3. Вычислим машинный порядок:

$Mp_2 = 101 + 100\ 0000 = 100\ 0101$.

4. Запишем представление числа в ячейке памяти.

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

Это и есть искомый результат. Его можно переписать в более компактной шестнадцатеричной форме:

45	*C*A	97	8*D*

Для того чтобы получить внутреннее представление отрицательного числа –25,324, достаточно в полученном выше коде заменить в разряде знака числа 0 на 1.

Получим:

11000101	11001010	10010111	10001101

А в шестнадцатеричной форме:

*C*5	*C*A	97	8*D*

Никакого инвертирования, как для отрицательных чисел с фиксированной точкой, здесь не происходит.

Рассмотрим, наконец, вопрос о диапазоне чисел, представимых в форме с плавающей точкой. Очевидно, положительные и отрицательные числа расположены симметрично относительно нуля. Следовательно, максимальное и минимальное числа равны между собой

по модулю: $R_{max} = |R_{min}|$. Наименьшее по абсолютной величине число равно нулю. Чему же равно R_{max}? Это число с самой большой мантиссой и самым большим порядком:

$$0{,}11111111111111111111111 \cdot 10_2^{1111111}.$$

Если перевести в десятичную систему, то получится

$$R_{max} = (1 - 2^{-24}) \cdot 2^{64} = 10^{19}.$$

Очевидно, что диапазон вещественных чисел значительно шире диапазона целых чисел. Если в результате вычислений получается число, по модулю большее, чем R_{max}, то происходит прерывание работы процессора. Такая ситуация называется переполнением при вычислениях с плавающей точкой. Наименьшее по модулю ненулевое значение равно

$$(1/2) \cdot 2^{-64} = 2^{-66}.$$

Любые значения, меньшие данного по абсолютной величине, воспринимаются процессором как нулевые.

Как известно из математики, множество действительных чисел бесконечно и непрерывно. Множество же вещественных чисел, представимых в памяти ЭВМ в форме с плавающей точкой, является ограниченным и дискретным. Каждое следующее значение получается прибавлением к мантиссе предыдущего единицы в последнем (24-м) разряде. Количество вещественных чисел, точно представимых в памяти машины, вычисляется по формуле

$$N = 2^t \cdot (U - L + 1) + 1,$$

где t — количество двоичных разрядов мантиссы; U — максимальное значение математического порядка; L — минимальное значение порядка.

Для рассмотренного нами варианта ($t = 24$, $U = 63$, $L = -64$) получается

$$N = 2\,146\,683\,548.$$

Все же остальные числа, не попадающие в это множество, но находящиеся в диапазоне допустимых значений, представляются в памяти приближенно (мантисса обрезается на 24-м разряде). А поскольку числа имеют погрешности, то и результаты вычислений с этими числами также будут содержать погрешность. Из сказанного следует вывод: вычисления с вещественными числами в компьютере выполняются приближенно.

2.2. Логические основы цифровой техники

Основные сведения из алгебры логики

Математический аппарат, на основе которого осуществляется описание цифровых схем,— это алгебра логики, или, как ее еще называют по имени автора — английского математика Джорджа Буля (1815–1864), булева алгебра. В практических целях первым применил ее американский ученый Клод Шеннон в 1938 г. при исследовании электрических цепей с контактными выключателями.

Логика — наука, изучающая методы доказательств и опровержений, т.е. методы установления истинности или ложности одних высказываний (утверждений, суждений) на основе истинности или ложности других высказываний.

Алгебра логики также называется исчислением высказываний.

Высказывание — это некое предложение, относительно которого можно утверждать, что оно или истинно, или ложно, и других возможностей нет («принцип двузначности — исключенного третьего»).

Высказывания отличаются от других языковых образований тем, что им можно присвоить только два значения: «истина», если они истинны; «ложь», если они ложны.

Если логика имеет дело со смыслом высказываний, то в алгебре логики работают с формулами. Любое элементарное высказывание обозначается буквой латинского алфавита.

Примеры простых высказываний

- A = <2 плюс 2 будет 4> — это истинное высказывание;
- B = <Кишинев — столица Румынии> — это ложное высказывание.

Из простых высказываний можно строить более сложные, применяя так называемые связи.

Поскольку высказывание может принимать одно из двух значений, то говорят о «переменных высказываниях». Замещение переменной конкретным высказыванием означает предоставления одного из значений — «истина» или «ложь».

Из элементарных высказываний с помощью логических связок «и», «или», «не», «если, то» и других (логических операций) строятся сложные высказывания — формулы (или функции) алгебры логики.

Логические связи — это функция алгебры логики (ФАЛ), аргументами которой являются простые высказывания.

Примеры сложных высказываний

Возьмем простое высказывание: *A*=<один плюс один — три>, тогда сложное высказывание НЕ *A* означает: неверно, что *A*, т.е. не верно, что <один плюс один — три>. Данное высказывание реализуется схемой «НЕ» (инвертором).

Возьмем два простых высказывания:

1) A = <Москва — столица РФ>;

2) B = <дважды два — четыре>.

Тогда сложное высказывание $A\&B$ будет истинным, так как истинны оба этих высказывания. Поскольку таблица истинности для конъюнкции совпадает с таблицей умножения, т.е. истинному высказыванию присваивается значение «1», а ложному — «0», то сложное высказывание называется произведением.

Способы построения новых высказываний из заданных с помощью логических связок, их преобразования и установления истинности изучаются в логике высказываний с помощью алгебраических методов.

Базовыми элементами, которыми оперирует алгебра логики, являются высказывания.

Высказывания строятся над множеством B, состоящего всего из двух элементов: B = {Ложь, Истина}. Над элементами данного множества определены три операции:

- отрицание (унарная операция);
- конъюнкция (бинарная), дизъюнкция (бинарная);
- логический ноль 0 и логическая единица 1 — константы.

Алгебра логики рассматривает логические выражения как алгебраические, которые можно преобразовать по определенным правилам. В выражениях алгебры логики переменные являются логическими (0 и 1). Знаки операций обозначают логические операции (логические связки).

Законы алгебры логики

Связи, существующие между логическими операциями, отражают законы булевой алгебры. Сформулируем основные из них.

Аксиомы:

1) $x + 0 = x\ x \cdot 0 = 0$;

2) $x + 1 = 1\ x \cdot 1 = x$;

3) $x + x = x\ x \cdot x = x$;

4) $x + \bar{x} = 1\ x \cdot \bar{x} = 0$;

5) $\bar{\bar{x}} = x$.

Законы:

6. Коммутативный закон:

$x + y = y + x$;

$x \cdot y = y \cdot x$;

7. Ассоциативный закон:

$(x + y) + z = x + (y + z)$;
$(x \cdot y) \cdot z = x \cdot (y \cdot z)$.

8. Дистрибутивный закон:

$x \cdot (y + z) = x \cdot y + x \cdot z$;
$x + (y \cdot z) = (x + y) \cdot (x + z)$.

9. Правила де Моргана:

$\overline{x + y} = \bar{x} + \bar{y}$;
$\overline{x \cdot y} = \bar{x} + \bar{y}$.

10. Закон поглощения:

$x + x \cdot y = x$;
$x \cdot (x + y) = x$.

11. Закон свертки:

$x + \bar{x} \cdot y = x + y$;
$x \cdot (\bar{x} + y) = x \cdot y$.

12. Закон склеивания:

$x \cdot y + x \cdot \bar{y} = x$;
$(x + y) \cdot (x + \bar{y}) = x$.

Справедливость данных аксиом и законов можно доказать методом подстановки.

Функции алгебры логики

Логической функцией называется функция $F(X_1, X_2, X_3, ..., X_i, ..., X_n)$, которая, так же как и ее аргументы, может принимать только два значения (0 и 1).

Рассмотрим некоторый набор аргументов

$\{X_1, X_2, X_3, ..., X_i, ..., X_n\}$

и будем считать, что каждый из аргументов принимает только одно из двух возможных значений: «0» или «1».

Число различных наборов равно 2^n.

Так как функция на каждом наборе может принять значение «0» или «1», а число различных наборов 2^n, то общее число различных функций n аргументов есть 2^{2n}.

Зависимость количества различных переключательных функций от числа аргументов представлена в табл. 2.6.

Таблица 2.6

Зависимость количества функций от числа аргументов

Число аргументов	1	2	3	4	5
Количество переключательных функций	4	16	256	65 536	4 294 967 296

Элементарная логическая функция реализуется с помощью логического элемента компьютера (часть электронной логической схемы). С помощью элементарных схем можно реализовать любую сложную логическую функцию, описывающую работу устройств компьютера.

Работу логических элементов описывают с помощью таблиц истинности.

Таблица истинности — это табличное представление ФАЛ, в котором перечислены все возможные сочетания значений истинности входных сигналов (операндов) вместе со значением истинности выходного сигнала (результата операции) для каждого из этих сочетаний.

Чтобы задать ФАЛ, нужно задать ее значения на всех наборах аргументов. Эти функции реализуются на 4 элементах, каждый из которых имеет максимум один вход. ФАЛ от одного аргумента представлен в табл. 2.7, от двух аргументов — в табл. 2.8.

Таблица 2.7

ФАЛ от одного аргумента

Аргумент X	Значение		Наименование функции	Обозначение функции
	0	1		
$f0(X)$	0	0	Константа «Ноль»	$F(X)=0$
$f1(X)$	0	1	Переменная X	$F(X)=X$
$f2(X)$	1	0	Инверсия X (отрицание X)	$F(X)=X$ $F(X)=\overline{X}$
$f3(X)$	1	1	Константа «Единица»	$F(X)=1$

Таблица 2.8

ФАЛ от двух аргументов

№ функции	Значение функции на наборах логических переменных				Наименование функции	Обозначение функции
$X1$	0	0	1	1		
$X2$	0	1	0	1		

Окончание табл. 2.8

№ функции	Значение функции на наборах логических переменных				Наименование функции	Обозначение функции
$f0(X1, X2)$	0	0	0	0	Константа «Ноль»	$f(X1, X2) = 0$
$f1(X1, X2)$	0	0	0	1	Конъюнкция, произведение	$f(X1, X2) = X1 \& X2$ $f(X1, X2) = X1 \wedge X2$ $f(X1, X2) = X1 \cdot X2$ $f(X1, X2) = X1\, X2$
$f2(X1, X2)$	0	0	1	0	Запрет по $X2$	$X1 \Delta X2$
$f3(X1, X2)$	0	0	1	1	Переменная $X1$	$f(X1, X2) = X1$
$f4(X1, X2)$	0	1	0	0	Запрет по $X1$	$X2 \Delta X1$
$f5(X1, X2)$	0	1	0	1	Переменная $X2$	$f(X1, X2) = X2$
$f6(X1, X2)$	0	1	1	0	Сложение по mod2 (неравнозначность)	$f(X1, X2) = X1 \oplus X2$
$f7(X1, X2)$	0	1	1	1	Дизъюнкция	$f(X1, X2) = X1 \vee X2$ $f(X1, X2) = X1 + X2$
$f8(X1, X2)$	1	0	0	0	Стрелка Пирса	$f(X1, X2) = X1 \downarrow X2$
$f9(X1, X2)$	1	0	0	1	Равнозначность	$f(X1, X2) = X1 \equiv X2$ $f(X1, X2) = X1 \sim X2$
$f10(X1, X2)$	1	0	1	0	Инверсия $X2$	$f(X1, X2) = \neg X2$ $f(X1, X2) = \overline{X2}$
$f11(X1, X2)$	1	0	1	1	Импликация от $X2$ к $X1$	$f(X1, X2) = X2 \rightarrow X1$
$f12(X1, X2)$	1	1	0	0	Инверсия $X1$	$f(X1, X2) = \wedge X1$ $f(X1, X2) = X1$
$f13(X1, X2)$	1	1	0	1	Импликация от $X1$ к $X2$	$f(X1, X2) = X1 \rightarrow X2$
$f14(X1, X2)$	1	1	1	0	Штрих Шеффера	$f(X1, X2) = X1 \mid X2$
$f15(X1, X2)$	1	1	1	1	Константа «Единица»	$f(X1, X2) = 1$

Логика высказываний послужила основным математическим инструментом при создании компьютеров. Она легко преобразуется в битовую логику: истинность высказывания обозначается одним битом (0 — ЛОЖЬ, 1 — ИСТИНА).

Впоследствии булева алгебра была обобщена от логики высказываний путем введения характерных для логики высказываний аксиом. Это позволило рассматривать, например, логику кубитов, трой-

ственную логику (когда есть три варианта истинности высказывания: «истина», «ложь» и «не определено»), комплексную логику и др.

Математический аппарат алгебры логики используется для конструирования электронных устройств, так как основной системой счисления в компьютере является двоичная система.

Особенностью алгебры логики является применимость для описания работы дискретных устройств вычислительной техники.

Одни и те же устройства компьютера могут применяться для обработки и хранения как числовой информации, представленной в двоичной системе счисления, так и логических переменных.

Данные и команды представляются в виде двоичных последовательностей различной структуры и длины.

В соответствии с логическим соглашением (приложение 1, ГОСТ 2.743-91) двоичная логика имеет дело с переменными, которые могут принимать два логических состояния — состояния «логическая 1» (далее — LOG1) и состояния «логический 0» (далее — LOG0).

Символы логических функций, определенные ГОСТ 2.743-91, представляют собой связь между входами и выходами элементов в терминах логических состояний, не связанных с физической реализацией.

При конкретной физической реализации элементов логические состояния представляются физическими величинами (электрический потенциал, давление, световой поток и др.). В логике не требуется знание абсолютного значения величины, поэтому физическая величина идентифицируется просто как более положительная — H и менее положительная — L (рис. 2.1). Эти два значения называются логическими уровнями.

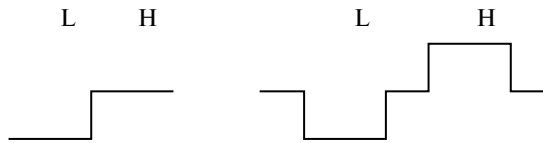


Рис. 2.1. Логические уровни физической величины

Соответствия между данными понятиями устанавливаются соглашениями положительной и отрицательной логики.

Соглашение положительной логики — более положительное значение физической величины (логический уровень H) соответствует LOG1, менее положительное значение физической величины (логический уровень L) соответствует LOG0.

Соглашение отрицательной логики — менее положительное значение физической величины (логический уровень L) соответствует LOG1, более положительное значение физической величины (логический уровень H) соответствует LOG0.

Логические элементы

Логические элементы — устройства, предназначенные для выполнения логических функций (операций) над входными сигналами (операндами, данными), представленными в цифровой форме.

Логические элементы выполняют логические операции «И», «ИЛИ», «НЕ» и комбинации этих операций. Указанные логические операции можно реализовать с помощью контактно-релейных и электронных схем. В настоящее время в подавляющем большинстве применяются электронные логические элементы, причем электронные логические элементы входят в состав микросхем. Имея в распоряжении логические элементы «И», «ИЛИ», «НЕ», можно сконструировать цифровое электронное устройство любой сложности. Электронная часть любого компьютера состоит из логических элементов.

Рассмотрим логические элементы с одним входом — элемент «НЕ» и «Повторитель» (буфер).

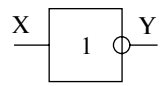
Логический элемент «НЕ» (Инвертор). Элемент, выполняющий функцию инверсии «НЕ», имеет один вход и один выход. Он меняет уровень сигнала на противоположный. Низкий потенциал на входе дает высокий потенциал на выходе, и наоборот.

Вход X	Выход Y
0	1
1	0

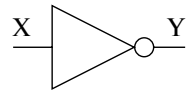
Мнемоническое правило для отрицания звучит так: На выходе будет:

- «1» тогда и только тогда, когда на входе «0»;
- «0» тогда и только тогда, когда на входе «1».

На принципиальных схемах логический элемент «НЕ» обозначают так:



В зарубежной документации элемент «НЕ» изображают следующим образом. Сокращенно называют его NOT:

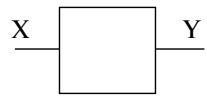


Все эти элементы в интегральных микросхемах могут объединяться в различных сочетаниях. Это элементы «И—НЕ», «ИЛИ—НЕ» и более сложные конфигурации.

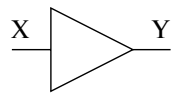
Логический элемент «Повторение». Элемент, выполняющий функцию «Повторение», имеет один вход и один выход. Он не меняет уровень сигнала на противоположный. Низкий потенциал на входе дает низкий потенциал на выходе, и высокий потенциал на входе дает высокий потенциал на выходе.

Вход *X*	Выход *Y*
0	0
1	1

На принципиальных схемах логический элемент «Повторение» обозначают так:



В зарубежной документации элемент «Повторение» изображают следующим образом:



Преобразование информации требует выполнения операций с группами знаков, простейшей из которых является группа из двух знаков. Оперирование с большими группами всегда можно разбить на последовательные операции с двумя знаками.

Рассмотрим основные логические элементы из $2(^{2^2})^{\cdot 1} = 2^4 = 16$ возможных бинарных логических операций с двумя знаками с унарным выходом.

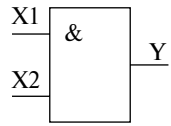
Логический элемент «И». На рисунке представлена таблица истинности элемента «И» с двумя входами. Хорошо видно, что логическая единица появляется на выходе элемента только при наличии единицы на первом входе и на втором. В трех остальных случаях на выходе будут нули.

Вход *X*1	Вход *X*2	Выход *Y*
0	0	0
1	0	0
0	1	0
1	1	1

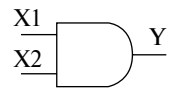
Логический элемент, реализующий функцию конъюнкции, называется *схемой совпадения*. Мнемоническое правило для конъюнкции с любым количеством входов звучит так. На выходе будет:

- «1» тогда и только тогда, когда на всех входах действуют «1»;
- «0» тогда и только тогда, когда хотя бы на одном входе действует «0».

На принципиальных схемах логический элемент «И» обозначают так:



На зарубежных схемах обозначение элемента «И» имеет другое начертание. Его кратко называют AND:



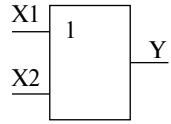
Логический элемент «ИЛИ». Элемент «ИЛИ» с двумя входами работает несколько по-другому. Достаточно логической единицы на первом входе или на втором, как на выходе будет логическая единица. Две единицы также дадут единицу на выходе.

Вход *X*1	Вход *X*2	Выход *Y*
0	0	0
1	0	1
0	1	1
1	1	1

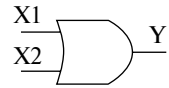
Мнемоническое правило для дизъюнкции с любым количеством входов звучит так. На выходе будет:

- «1» тогда и только тогда, когда хотя бы на одном входе действует «1»;
- «0» тогда и только тогда, когда на всех входах действует «0».

На схемах элемент «ИЛИ» изображают так:

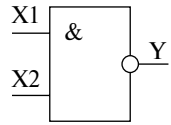


На зарубежных схемах его изображают чуть по-другому и называют элементом OR:

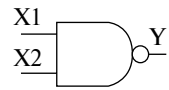


Логический элемент 2И-НЕ (штрих Шеффера). В логическом элементе «2И-НЕ» цифра всегда обозначает число входов логического элемента. В данном случае это двухвходовой элемент «И», выходной сигнал которого инвертируется, «0» превращается в «1», а «1» — «0». Обратим внимание на кружочек на выходах — это символ инверсии.

На принципиальных схемах логический элемент «2И-НЕ» обозначают так:



Зарубежное обозначение элемента «И-НЕ» (в данном случае «2И-НЕ») называется NAND:



По сути, это упрощенное изображение двух объединенных элементов: элемента «2И» и элемента «НЕ» на выходе.

Таблица истинности для элемента «2И-НЕ».

Вход *X*1	Вход *X*2	Выход *Y*
0	0	1
1	0	1
0	1	1
1	1	0

В таблице истинности элемента «2И-НЕ» мы видим, что благодаря инвертору получается картина, противоположная элементу «И». В отличие от трех нулей и одной единицы мы имеем три единицы и ноль. Элемент «И-НЕ» часто называют элементом Шеффера.

Мнемоническое правило для «ИЛИ-НЕ» с любым количеством входов звучит так. На выходе будет:

- «1» тогда и только тогда, когда на всех входах действуют «0»;
- «0» тогда и только тогда, когда хотя бы на одном входе действует «1».

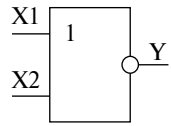
Логический элемент «2ИЛИ-НЕ» (стрелка Пирса). Логический элемент «2ИЛИ-НЕ» — это двухвходовой элемент «ИЛИ», выходной сигнал которого инвертируется. На выходе только один высокий потенциал при условии, что на оба входа одновременно действует низкий потенциал. Здесь, как и на любых других принципиальных схемах, кружочек на выходе подразумевает инвертирование сигнала.

Таблица истинности также отличается от схемы «ИЛИ» применением инвертирования выходного сигнала.

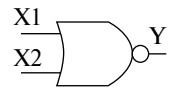
Таблица истинности для логического элемента «2ИЛИ-НЕ»:

Вход *X*1	Вход *X*2	Выход *Y*
0	0	1
1	0	0
0	1	0
1	1	0

Изображение на схеме:



На зарубежный лад изображается так. Называют как NOR:



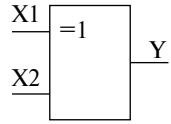
Логический элемент «исключающее ИЛИ» (неравнозначность). К числу базовых логических элементов принято относить элемент, реализующий функцию «исключающее ИЛИ». Иначе эта функция называется «неравнозначность».

Высокий потенциал на выходе возникает только в том случае, если входные сигналы не равны, т.е. на одном из входов должна быть единица, а на другом — ноль.

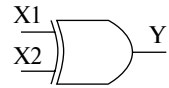
Таблица истинности:

Вход *X*1	Вход *X*2	Выход *Y*
0	0	0
1	0	1
0	1	1
1	1	0

Эти логические элементы находят свое применение в сумматорах. «Исключающее ИЛИ» изображается на схемах знаком равенства перед единицей «=1»:



На зарубежный манер «исключающее ИЛИ» называют XOR и на схемах изображают так:



Мнемоническое правило для суммы по модулю два с любым количеством входов звучит так. На выходе будет:

- «1» тогда и только тогда, когда на входе действует нечетное количество;
- «0» тогда и только тогда, когда на входе действует четное количество.

Словесное описание: «истина на выходе — только при истине на входе 1 либо только при истине на входе 2».

Логический элемент «исключающее ИЛИ-НЕ» (равнозначность).

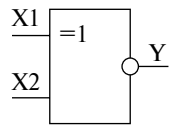
К числу базовых логических элементов принято относить элемент, реализующий функцию «исключающее ИЛИ». Иначе эта функция называется «равнозначность».

Если на выходе логического элемента «исключающее ИЛИ-НЕ» имеется инвертор, то функция выполняется противоположная — «равнозначность». Высокий потенциал на выходе возникает только в том случае, если входные сигналы равны.

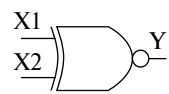
Таблица истинности:

Вход *X*1	Вход *X*2	Выход *Y*
0	0	1
1	0	0
0	1	0
1	1	1

Эти логические элементы находят свое применение в сумматорах. «Исключающее ИЛИ-НЕ» изображается на схемах следующим образом:



На зарубежный манер «исключающее ИЛИ-НЕ» называют XNOR и на схемах изображают так:



Мнемоническое правило эквивалентности с любым количеством входов звучит так. На выходе будет:

- «1» тогда и только тогда, когда на входе действует четное количество;
- «0» тогда и только тогда, когда на входе действует нечетное количество.

Словесная запись: «истина на выходе при истине на входе 1 и входе 2 или при лжи на входе 1 и входе 2».

Рассмотрим несколько реальных логических элементов на примере серии транзисторно-транзисторной логики (ТТЛ) КР1533 с малой степенью интеграции. На рисунке представлена микросхема КР1533ЛА3 (рис. 2.2), которая содержит четыре независимых элемента «2И-НЕ». Аналог — SN74ALS00A.

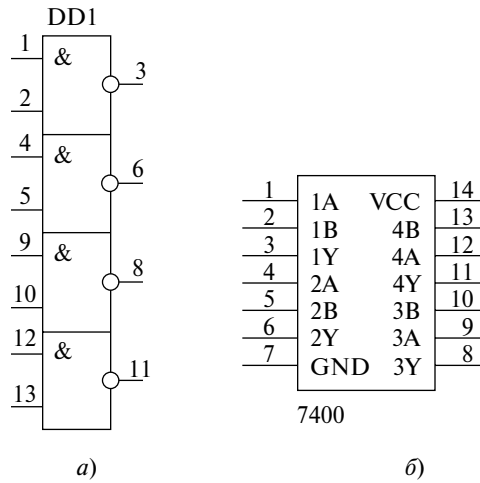


Рис. 2.2. Четыре логических элемента «2И-НЕ»:
а — микросхема КР1533ЛА3; *б* — аналог SN74ALS00A

В той же серии существуют элементы «3И-НЕ», «4И-НЕ», «8И-НЕ», что означает элементы «И» с различным числом входов. Например, микросхема КР1533ЛА4 (аналог — SN74ALS10) содержит три логических элемента «3И-НЕ», микросхема КР1533ЛА1 (аналог — SN74ALS20) содержит два логических элемента «4И-НЕ», а микросхема КР1533ЛА2 (аналог — SN74ALS30) содержит один логический элемент «8И-НЕ».

Логический элемент «2ИЛИ-НЕ» представлен микросхемой КР1533ЛЕ 1 (рис. 2.3). Она содержит в одном корпусе четыре независимых элемента. Аналог — SN74ALS02.

Количество логических элементов «2ИЛИ-НЕ» в одном корпусе может достигать шести — микросхемы К155ЛН1 и К561ЛН2.

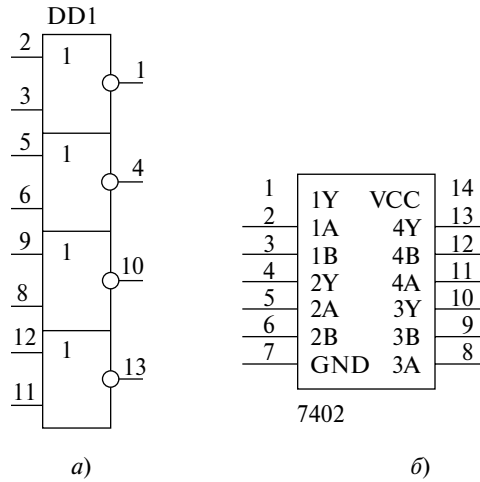


Рис. 2.3. Четыре логических элемента «2ИЛИ-НЕ»:
а — микросхема КР1533ЛЕ 1; *б* — аналог SN74ALS02

Часто используются элементы, объединенные в различных сочетаниях. Например, микросхема К555ЛР4 (аналог — SN74LS55) называется «2-4И-2ИЛИ-НЕ» (рис. 2.4).

Любая логическая схема без памяти полностью описывается таблицей истинности. При построении сложных логических схем с произвольной таблицей истинности используется сочетание простейших схем «И», «ИЛИ», «НЕ».

При построении схемы, реализующей произвольную таблицу истинности, каждый выход анализируется (и строится схема) отдельно. Для реализации таблицы истинности при помощи логических элементов «И» достаточно рассмотреть только те строки таблицы истинности, которые содержат логические «1» в выходном сигнале. Строки, содержащие в выходном сигнале логический «0», в построении схемы не участвуют. Каждая строка, содержащая в выходном сигнале логическую единицу, реализуется схемой логического «И» с количеством входов, совпадающим с количеством входных сигналов в таблице истинности. Входные сигналы, описанные в таблице истинности логической единицы, подаются на вход этой схемы непосредственно, а входные сигналы, описанные в таблице истинности логическим «0», подаются на вход через инверторы. Объединение сигналов с выходов схем, реализующих отдельные строки таблицы истинности, производится при помощи схемы логического «ИЛИ». Количество входов в этой схеме определяется количеством строк в таблице истинности, в которых в выходном сигнале присутствует логическая единица.

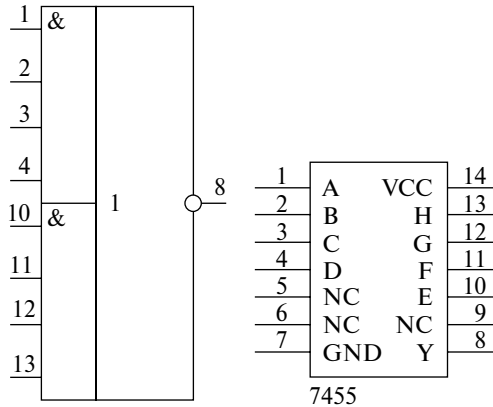


Рис. 2.4. Логический элемент «2-4И-2ИЛИ-НЕ»:
а — микросхема К555ЛР4, *б* — аналог SN74LS55

Построение логических схем с произвольной таблицей истинности

Рассмотрим конкретный пример. Пусть необходимо реализовать таблицу истинности, приведенную на рис. 2.5.

Входы				Выходы	
In_0	In_1	In_2	In_3	Out_0	Out_0
0	0	0	0	0	0
0	0	0	1	1	0
0	0	1	0	0	0
0	0	1	1	0	0
0	1	0	0	0	0
0	1	0	1	0	1
0	1	1	0	1	0
0	1	1	1	0	0
1	0	0	0	0	0
1	0	0	1	0	1
1	0	1	0	0	0
1	0	1	1	0	0
1	1	0	0	1	0
1	1	0	1	0	0
1	1	1	0	0	0
1	1	1	1	0	1

Рис. 2.5. Произвольная таблица истинности

Для построения схемы, реализующей сигнал *Out*0, достаточно рассмотреть строки, в которых сигнал *Out*0 = 1 (в таблице строки подчеркнуты прямой линией). Эти строки реализуются микросхемой *D*2 на рис. 2.6. Каждая строка реализуется своей схемой «И», затем выходы этих схем объединяются схемой «ИЛИ». Для построения схемы, реализующей сигнал *Out*1, достаточно рассмотреть строки, в которых сигнал *Out*1 = 1 (в таблице строки подчеркнуты извилистой линией). Эти строки реализуются микросхемой *D*3.

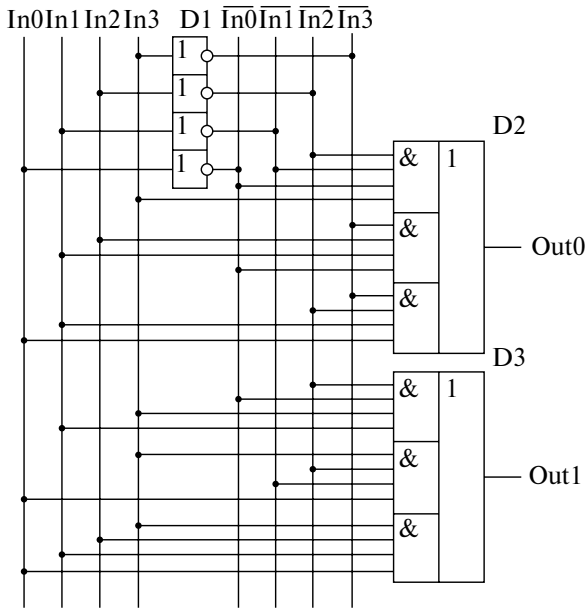


Рис. 2.6. Принципиальная схема, реализующая таблицу истинности, приведенную на рис. 2.5

Минимизация логических функций

Стоимость схемы, реализующей логическую функцию, пропорциональна числу вхождений переменных или их отрицаний и числу логических операций. Поэтому логические функции необходимо минимизировать (упрощать). Для минимизации используют специальные алгоритмические методы, такие как метод Квайна, метод карт Карно, метод испытания импликант, метод импликантных матриц, метод Квайна—Мак-Класки и др.

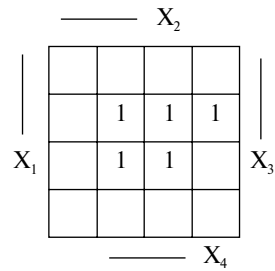
При минимизации логической функции учитывается, каким методом эффективнее будет реализовать ее минимальную форму при помощи электронных схем.

Если число переменных невелико (меньше или равно шести), то представление функций удобно с помощью карт Карно. Карта Карно представляет собой таблицу, которая содержит 2^n клеток, где n — число переменных ($n = 1, 2, 3, ..., n$). К каждой клетке таблицы содержится логическое произведение переменных или их инверсий. Одни переменные располагаются по горизонтали, другие — по вертикали.

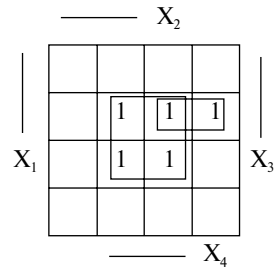
Пример. Функция алгебры логики состоит из четырех переменных: x_1, x_2, x_3, x_4. Карта Карно будет содержать 16 клеток ($2^4 = 16$):

$$y = x_1x_2x_3x_4 + \bar{x}_1x_2x_3x_4 + x_1\bar{x}_2x_3x_4 + x_1\bar{x}_2x_3\bar{x}_4 + \bar{x}_1\bar{x}_2x_3x_4.$$

Составим карту Карно:



Если переменная без инверсии, то в карте Карно ставится единица, если с инверсией — ноль. Вторая сверху слева ячейка, в ней стоит единица, и этой единице соответствует первое слагаемое в формуле. Единица стоит именно там, поскольку в этой ячейке перекрываются все переменные. Стоит уйти на ячейку влево, вправо, вверх, вниз — и одна из переменных уже перекрывать ее не будет. Если одна из переменных с инверсией, что характерно для второго слагаемого, единица ставится с учетом неперекрытия ячейки этой переменной, т.е. для второго слагаемого x_1 с инверсией и единица стоит в третьей (сверху) строке третьего столбца. Для четвертого слагаемого с инверсией переменные x_2 и x_4. Поэтому единица стоит в последней строке (второй столбец). Так проставляются все единицы. В пустых ячейках по идее стоят нули (не показаны для наглядности). После проставления всех единиц начинается объединение ячеек, в которых стоят эти самые единицы. Объединяются клетки по следующим правилам: число объединяемых клеток 2, 4, 8, 16, 32 и т.д., т.е. 2^n; объединять надо как можно больше клеток, а самих объединений должно быть как можно меньше. На рисунке ниже показано, как это делается.



Следует иметь в виду, что не объединяются единицы в количестве три, пять, шесть, семь, девять, десять и т.п. Не объединяются также единицы, которые стоят буквой Т, Г.

После объединения ячеек составляется уравнение функции. Объединенные клетки условно считаются за одну. Как видно из рисунка, четырем объединенным единицам соответствует выражение x_3x_4. То есть область объединения попадает в зону видимости всех переменных. Во втором объединении не участвует x_2, поэтому эта переменная будет с инверсией. Таким образом, после преобразования с помощью карты Карно получаем следующую функцию:

$$y = x_3x_4 + x_1\bar{x}_2x_3.$$

Таким образом, из пяти слагаемых получили два. Такая форма записи функции называется *совершенной дизъюнктивной нормальной формой* (СДНФ). Это такая ДНФ, которая удовлетворяет трем условиям:

- в ней нет одинаковых элементарных конъюнкций;
- в каждой конъюнкции нет одинаковых пропозициональных букв;
- каждая элементарная конъюнкция содержит каждую пропозициональную букву из входящих в данную ДНФ пропозициональных букв, причем в одинаковом порядке.

Для любой функции алгебры логики существует своя СДНФ, причем единственная.

Существует также конъюнктивная нормальная форма (КНФ) — форма представления функции в виде конъюнкции (логического умножения) ряда членов, каждый из которых является простой дизъюнкцией (логическим сложением) аргументов.

СКНФ (*совершенная конъюнктивная нормальная форма*) — это такая КНФ, которая удовлетворяет трем условиям:

- в ней нет одинаковых элементарных дизъюнкций;
- в каждой дизъюнкции нет одинаковых пропозициональных букв;
- каждая элементарная дизъюнкция содержит каждую пропозициональную букву из входящих в данную КНФ пропозициональных букв.

Техническая реализация логических функций

При реализации цифровых устройств на интегральных микросхемах широко используются базисы «И-НЕ» или «ИЛИ-НЕ». Для этого минимизированные логические функции путем преобразований приводятся к соответствующему виду.

Пусть минимальная ДНФ функция

$$F(A,B,C,) = AB \vee BC \vee AC.$$

Применим к этому выражению двойное отрицание и теорему де Моргана:

$$F = \overline{\overline{F}} = \overline{\overline{AB \vee BC \vee AC}} = \overline{(\overline{AB})(\overline{BC})(\overline{AC})}.$$

Как видно, функция F включает только операции «И-НЕ», и ее реализация в базисе «И-НЕ» имеет вид (рис. 2.7):

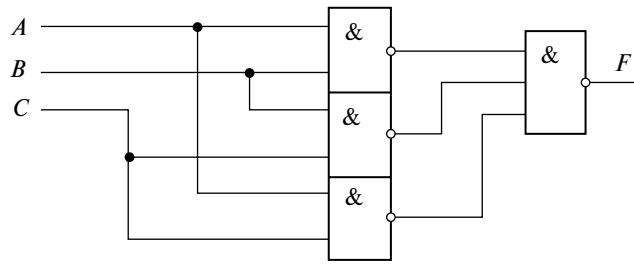


Рис. 2.7. Реализация функции $F = AB \vee BC \vee AC$ в базисе «И-НЕ»

Аналогичным образом от КНФ функции можно перейти к ее форме, удобной для реализации в базисе «ИЛИ-НЕ».

2.3. Основные цифровые логические устройства

Цифровые устройства — это электронные функциональные узлы, которые обрабатывают цифровые сигналы.

Логическое цифровое устройство может быть представлено в виде многополюсника, выполняющего определенную логическую функцию между входными (X_0, X_1, ..., X_{n-1}) и выходными (Y_0, Y_1, ..., Y_{m-1}) сигналами (рис. 2.8).

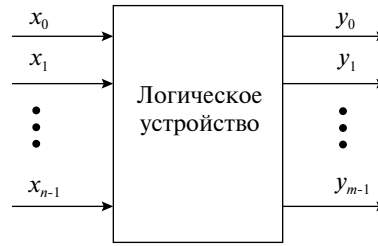


Рис. 2.8. Обобщенная схема логического устройства

Классификация логических устройств

Классификация логических устройств может быть по различным признакам.

По способу ввода-вывода переменных (информации) они подразделяются:

- на последовательные;
- параллельные;
- последовательно-параллельные (смешанные).

Последовательные устройства — это устройства, в которых входные переменные подаются на вход, а выходные переменные снимаются с выхода не одновременно, а последовательно, разряд за разрядом.

Параллельные устройства — устройства, в которых все разряды входных переменных подаются на вход и все разряды выходных переменных снимаются с выхода одновременно.

В *последовательно-параллельных* устройствах входные и выходные переменные представлены в различных формах: либо на вход переменные подаются последовательно символ за символом, а с выхода они снимаются одновременно, либо наоборот. По принципу действия все логические устройства делятся на два класса:

- комбинационные;
- последовательностные.

Комбинационными устройствами или *автоматами без памяти* называют логические устройства, выходные сигналы которых однозначно определяются только действующей в настоящий момент на входе комбинацией переменных и не зависят от значений переменных, действовавших на входе ранее.

Последовательностными устройствами, или *автоматами с памятью*, называют логические устройства, выходные сигналы которых определяются не только действующей в настоящий момент на входе комбинацией переменных, но и всей последовательностью входных

переменных, действовавших в предыдущие моменты времени. Этот тип устройств часто называют *цифровыми автоматами*.

Структура последовательностного и комбинационного устройств приведена на рис. 2.9.

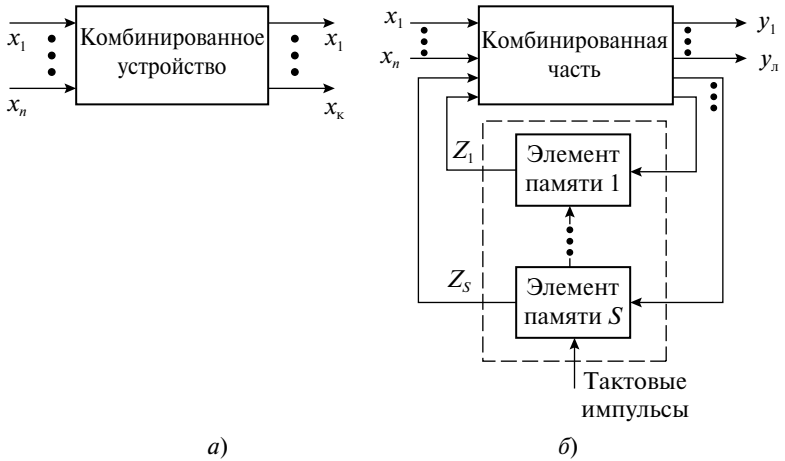


Рис. 2.9. Структура цифрового устройства:
а — комбинационного; *б* — последовательностного

В комбинационных устройствах взаимосвязь между входными и выходными переменными задается таблицей истинности, а алгебраическая форма этих связей описывается системой уравнений:

$$y_1 = y_1(x_1, x_2, ..., x_n);$$
$$y_k = y_k(x_1, x_2, ..., x_n).$$

В последовательностных устройствах выходные переменные y_i зависят не только от входных сигналов x_m, но и от сигналов элементов памяти, поступающих за этот же такт. Цифровые устройства с памятью называют конечными автоматами, так как запоминающее устройство может хранить информацию только в течение ограниченного числа тактов. Для описания работы последовательностных устройств используются таблицы переходов состояний.

Цифровые устройства можно разделить на асинхронные и синхронные. В *асинхронных* изменение входных сигналов сразу же вызывает изменение выходных сигналов. В *синхронных* изменение выходных сигналов, соответствующее новому сочетанию входных, происходит только после подачи синхронизирующих (тактовых) импульсов,

управляющих работой автомата. Период синхроимпульсов является, таким образом, минимальным временем между выполнением автоматом двух последовательных микроопераций, т.е. служит единицей машинного времени, называемой *тактом*. В зависимости от структуры автомата за один такт могут выполняться одна или несколько микроопераций, если они совмещены во времени.

В асинхронных устройствах отсутствуют синхронизирующие сигналы, поэтому в их структуры обычно включаются специальные схемы, которые после окончания каждой микрооперации вырабатывают сигнал готовности к выполнению следующей микрооперации.

Синхронные устройства в принципе имеют меньшее быстродействие, чем асинхронные, однако в них легко устраняются опасные состязания.

Технологии реализации цифровых интегральных логических элементов

В процессе развития интегральной электроники выделилось несколько типов схем логических элементов, имеющих достаточно хорошие характеристики и удобных для реализации в интегральном исполнении, которые служат элементной базой современных цифровых микросхем.

Базовые элементы, независимо от их микросхемотехники и особенностей технологий изготовления, строятся в одном из базисов (как правило, в базисе «И-НЕ» или «ИЛИ-НЕ»).

Базовые элементы выпускаются в виде отдельных микросхем либо входят в состав функциональных узлов и блоков, реализованных в виде инженерных схем (ИС), больших интегральных схем (БИС), сверхбольших интегральных схем (СБИС).

В процессе реализации базовые логические элементы строят из двух частей: входной логики, выполняющей операции «И» или «ИЛИ», и выходного каскада, выполняющего операцию «НЕ».

Входная логика может быть выполнена на диодах, биполярных и полевых транзисторах. В зависимости от этого различают:

- транзисторно-транзисторную логику (ТТЛ, ТТЛШ);
- интегральную инжекционную логику (ИИЛ, И2Л);
- логику на металл-диэлектрик-полупроводниковых (МДП) транзисторах (МДП, МОП).

МОП — транзисторная логика на комплементарных транзисторах (КМОП-логика).

В перечисленных группах логических элементов в качестве выходного каскада используется ключевая схема (инвертор). Другая

группа логических элементов основана на переключателях тока — эмиттерно-связанная логика (ЭСЛ-логика).

Транзисторно-транзисторная логика (ТТЛ). Основой транзисторно-транзисторной логики является базовый элемент на основе многоэмиттерного транзистора Т 1 (рис. 2.10), который легко реализуется в едином технологическом цикле с транзистором Т 2. В ТТЛ-логике многоэмиттерный транзистор осуществляет в положительной логике операцию «И», а на транзисторе Т 2 собран инвертор. Таким образом, по данной схеме реализован базис «И-НЕ».

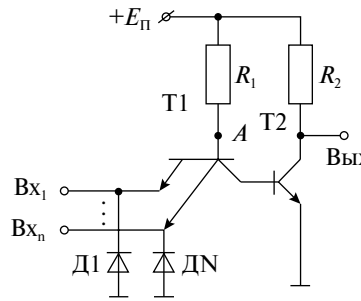


Рис. 2.10. Базовый элемент ТТЛ

В случае подачи на все входы схемы высокого потенциала все переходы эмиттер — база транзистора Т1 окажутся запертыми, так как потенциал в точке A примерно равен входным сигналам. В то же время переход база–коллектор будет открытым, поэтому по цепи $E_п$ — R_1 — база Т1 — коллектор Т1 — база Т2 — эмиттер Т2 — корпус течет ток $I_{б\,нас}$, который открывает транзистор Т2 и вводит его в насыщение. Потенциал на выходе схемы оказывается близким к нулю (на уровне ≈ 0,1 В). Сопротивление R_1 подобрано таким, чтобы за счет падения напряжения на нем от тока $I_{б\,нас}$ транзистора Т2 потенциал в точке A был бы ниже, чем потенциал входов, и эмиттеры Т1 оставались бы запертыми.

При подаче низкого потенциала логического нуля хотя бы на один из входов открывается этот переход эмиттер — база транзистора Т1, появляется значительный ток $I_э$ и потенциал в точке A, равный $E_п - I_э R_1$, приближается к нулевому. Разность потенциалов между базой и эмиттером Т2 также становится равной нулю, ток $I_б$ транзистора Т2 прекращается и он закрывается (переходит в режим отсечки). В результате выходное напряжение приобретает значение, равное напряжению питания (логической единице).

Входные диоды Д1, ..., ДN предназначены для демпфирования (отсечки) отрицательных колебаний, которые могут присутствовать во входных сигналах за счет паразитных элементов предыдущих каскадов.

Существенным недостатком рассмотренной схемы элемента «И-НЕ» являются низкие нагрузочная способность и экономичность ее инвертора, поэтому в практических схемах используют более сложный инвертор.

В конце 1970-х гг. началось широкое применение серий элементов на транзисторах Шоттки с повышенным быстродействием за счет уменьшения задержки выключения ключей. По принципу действия базовый элемент ТТЛШ аналогичен ТТЛ-элементу.

Необходимо заметить, что схемам ТТЛ и ТТЛШ свойственен большой логический перепад напряжений, равный

$$U_{\text{л}} = E_{\text{к}} - U_{\text{кэ нас}} \approx E_{\text{к}}.$$

Интегральная инжекционная логика (И^{2}Л). Схемы И^{2}Л не имеют аналогов в дискретных транзисторных схемах, т.е. характерны именно для интегрального исполнения. Основой И^{2}Л элементов является инвертор (рис. 2.11), составленный из двух транзисторов.

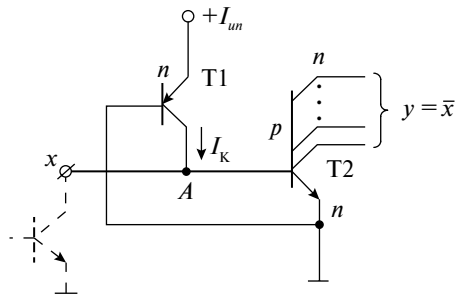


Рис. 2.11. Базовый элемент инжекционной логики

Транзистор Т1 является транзистором *n-p-n*-типа, а транзистор Т2 — *p-n-p*-типа, причем одна из областей *n*-типа является как базой транзистора Т1, называемого инжектором (отсюда и название логики), так и эмиттером транзистора Т2, а база транзистора Т2 является коллектором инжектора. Функционально транзистор Т1 выполняет роль нагрузочного резистора, а Т2 — полупроводникового ключа.

Выходной транзистор — многоколлекторный, что обеспечивает развязку выходов друг от друга. Если ключевой транзистор предыдущей схемы открыт, то через него замыкается на корпус ток $I_{\text{к}}$ тран-

зистора Т1, заданный внешним источником тока, и не поступает в базу транзистора Т2, оставляя его закрытым.

Если же ключевой транзистор предыдущей схемы заперт, то ток $I_к$ потечет в базу Т2 и вызовет его открывание. Таким образом, рассматриваемый базовый элемент реализует операцию «НЕ», принимая открытое состояние Т2 за ноль, а запертое — за единицу.

Соединив параллельно (рис. 2.12) два базовых элемента, можно получить реализацию базиса «ИЛИ-НЕ».

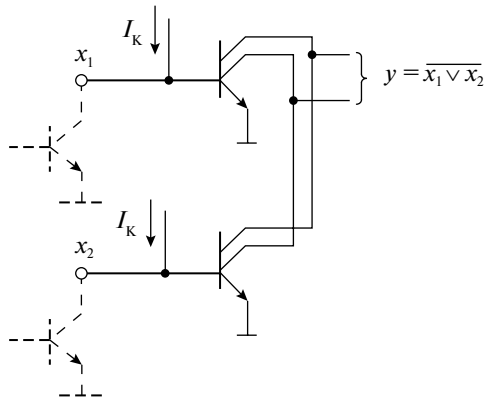


Рис. 2.12. Реализация схемы «ИЛИ-НЕ» в логике $И^2Л$

В качестве источников тока питания $I_{ип}$ служат генераторы токов на *p-n-p*-транзисторах, включенных по схеме с общей базой. Из-за отсутствия в схеме резисторов и общих для обоих транзисторов областей *p*- и *n*-типа схема очень технологична и в интегральном исполнении позволяет достичь плотности упаковки в 50 раз выше, чем при ТТЛ-технологии.

При напряжении питания 1,5 В значение высокого потенциала порядка 0,7 В, а низкого — 0,05 В. Так как транзистор Т 1 представляет высокоомную нагрузку, потребляемая элементом мощность может быть снижена до чрезвычайно низкой величины (в 100 раз меньше, чем у ТТЛ-элементов). Поэтому элементы $И^2Л$ нашли широкое применение в БИС (серии КР582, 584).

В сериях ИС невысокой степени интеграции логика $И^2Л$ неэффективна из-за низкого логического перепада, равного 0,65 В, и поэтому низкой помехоустойчивости. Кроме того, по быстродействию вследствие глубокого насыщения транзисторов инвертора $И^2Л$-элементы уступают ТТЛШ-элементам.

Логические элементы на МДП-транзисторах. В настоящее время в логических схемах используются МДП-транзисторы с диэлектриком SiO_2 (МОП-транзисторы).

Анализ МОП-транзисторных логических элементов достаточно прост, так как из-за отсутствия входных токов их можно рассматривать отдельно от других элементов даже при работе в цепочке.

На рис. 2.13 показаны два варианта построения логических элементов на МОП-транзисторах с *n* каналами.

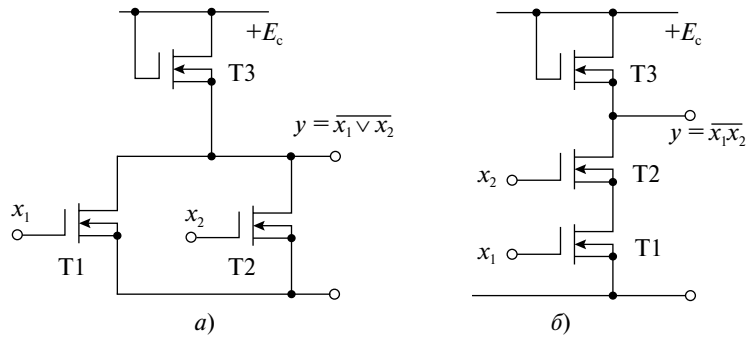


Рис. 2.13. Логические элементы на МОП-транзисторах:
а — элемент «ИЛИ-НЕ»; *б* — элемент «И-НЕ»

Транзисторы Т 3 выполняют роль нагрузки.

Логические уровни в обеих схемах не зависят от нагрузки и соответствуют выходным напряжениям открытого и закрытого ключа:

$$U^0_{\text{вых}} \approx 0{,}1\text{В};\ U^1_{\text{вых}} \approx E_\text{с}.$$

Соответственно, логический перепад составляет

$$U_\text{л} = U^1_{\text{вых}} - U^0_{\text{вых}} \approx E_\text{с}.$$

Напряжение питания $E_\text{с}$ МОП-логики выбирают в 3–4 раза больше порогового напряжения $U_\text{о}$ открывания транзисторов. Если $U_\text{о}$ = 1,5–3 В, то получаемый логический перепад в 5–10 В намного превышает значения, свойственные схемам $И^2Л$, ЭСЛ и даже ТТЛ (при напряжении питания 4–5 В). Поэтому МОП-логика обладает повышенной помехоустойчивостью.

Более высоким быстродействием и низким энергопотреблением характеризуется логика на комплементарных транзисторах вследствие причин, рассмотренных ранее. По принципу действия и схемотехнике КМОП-логика очень близка МОП-логике.

Эмиттерно-связанная логика (ЭСЛ). В основе схемы ЭСЛ лежит переключатель тока, в одно из плеч которого включено параллельно несколько транзисторов. Эти транзисторы равноправны — отпирание любого из них (или всех вместе) приводит к изменению логического состояния переключателя. Поэтому ЭСЛ-элементы выполняют логическую функцию «ИЛИ-НЕ».

Вследствие ненасыщенного режима работы транзисторов логический перепад в схеме не превышает 0,65 В.

Параметры интегральных логических элементов

Независимо от принадлежности к той или иной серии все логические элементы характеризуются определенным, одним и тем же набором параметров, которые являются справочными данными. Значения этих параметров обусловлены схемотехническим, конструктивным и технологическим исполнением элементов.

Значения параметров, как правило, задаются с запасом и не исчерпывают физических возможностей микросхемы, однако превышать их не следует.

Оценивают микросхемы по следующим основным параметрам: быстродействию, напряжению питания, потребляемой мощности, коэффициенту разветвления по выходу, коэффициенту объединения по входу, помехоустойчивости, энергии переключения, надежности, стойкости к климатическим и механическим воздействиям. Рассмотрим основные из них.

Уровни выходных напряжений. Техническими условиями для каждой серии логических элементов задаются наибольший и наименьший уровни выходных напряжений, соответствующих логическим единице и нулю при допустимых изменениях напряжения питания, нагрузки, температуры. Напряжение $U^1_{\text{вых min}}$ соответствует минимальному уровню логической единицы на выходе (для ТТЛ $U^1_{\text{вых min}} = 2{,}4$ В), а напряжение $U^0_{\text{вых max}}$ — максимальному уровню логического нуля (для ТТЛ $U^0_{\text{вых max}} = 0{,}4$ В).

Статическая помехоустойчивость. Этот параметр определяет допустимое напряжение помех на входах микросхемы и оценивается для низкого и высокого уровней напряжения.

Статической помехоустойчивостью по низкому уровню считают разность

$$U^0_{\text{пом}} = |U^0_{\text{вых max}} - U^0_{\text{вх min}}|,$$

где $U^0_{\text{вых max}}$ — максимальное допустимое напряжение низкого уровня на выходе нагруженной микросхемы; $U^0_{\text{вх max}}$ — максималь-

ное допустимое напряжение низкого уровня на входе нагружающей микросхемы.

Помехоустойчивость по высокому уровню определяют так:

$$U^1_{\text{пом}} = |U^1_{\text{вых max}} - U^1_{\text{вх min}}|,$$

где $U^1_{\text{вых min}}$ — минимальное напряжение высокого уровня на выходе нагруженной микросхемы; $U^1_{\text{вх min}}$ — минимальное допустимое напряжение высокого уровня на нагружающем входе.

Например, ТТЛ-логика еще будет нормально работать, если на ее входе напряжение логического нуля достигнет 0,8 В, а напряжение логической единицы снизится до 2 В. Таким образом, гарантированный запас помехоустойчивости в обоих состояниях составляет 0,4 В. Реальный же запас помехоустойчивости гораздо больше и превышает 1 В.

Коэффициент разветвления по выходу. Этот параметр $K_{\text{раз}}$ (нагрузочная способность) определяет максимальное число входов элементов данной серии, которым можно нагружать выходы микросхемы без нарушения ее нормального функционирования.

Коэффициент объединения по входу. $K_{\text{об}}$ определяет число логических входов, которые имеет логический элемент. Простейшие логические элементы выпускаются с 2, 3, 4 и 8 входами. Более сложные устройства содержат и другие входы: адресные, установочные, разрешающие, входы синхронизации и др.

Входные токи. Эти параметры определяют нагрузку, которую представляет рассматриваемая схема, на предшествующую схему или другой источник сигнала. Различают входные токи $I^0_{\text{вх}}$ и $I^1_{\text{вх}}$ при подаче логических нуля или единицы.

Средняя статическая потребляемая мощность. Определяется следующим образом:

$$P_{\text{ст. ср}} = {}^1\!/_2(P^0_{\text{пот}} + P^1_{\text{пот}}),$$

где $P^0_{\text{пот}}$ и $P^1_{\text{пот}}$ — мощности, потребляемые интегральным логическим элементом в состоянии логического нуля и логической единицы. Это вытекает из того, что в сложных многоэлементных устройствах в среднем половина логических элементов находится в состоянии 1, а половина — в состоянии 0.

Быстродействие. Характеризуется максимальной частотой смены входных сигналов, при которой еще не нарушается нормальное функционирование устройства.

Инерционность полупроводниковых приборов и паразитные емкости служат причиной того, что каждое переключение сопровождается переходными процессами, отчего фронты импульсов растягиваются.

Для оценки временных свойств микросхем обычно пользуются задержкой распространения сигнала, которая представляет собой интервал времени между входным и выходным импульсами, измеренными на уровне 0,5. Задержки распространения сигнала при включении $t^{1,0}_{\text{зд.р}}$ и при выключении $t^{0,1}_{\text{зд.р}}$ не равны, поэтому пользуются усредненным параметром $t_{\text{зд.р. ср}} = 0{,}5(t^{1,0}_{\text{зд.р}} + t^{0,1}_{\text{зд.р}})$.

Для последовательностных устройств (триггеры, счетчики и др.) вводятся некоторые дополнительные временные параметры, обусловленные принципом действия: разрешающее время, длительность входного импульса и др.

В общем случае анализ физических, технологических и схемотехнических особенностей интегральных логических элементов показывает, что можно создать различные их варианты, но их особенностью будут либо относительно высокое (высокое) быстродействие при низкой экономичности, либо высокая экономичность при относительно низком (низком) быстродействии. Обобщенные характеристики известных типов интегральных логических элементов приведены в табл. 2.9.

Таблица 2.9

Обобщенные характеристики интегральных логических элементов

Тип логики	$P_{\text{ст. ср}}$, мВт	$t_{\text{зд.р. ср}}$, нс	$U_{\text{пом}}$, В	$K_{\text{об}}$	$K_{\text{раз}}$
ТТЛ ТТЛШ	1–20	5–20 2–10	0,8–1 0,5–0,8	2–8	10–30 10–40
ЭСЛ	20–50	0,5–2	0,2–0,3	2–8	1–20
$И^2Л$	0,01–0,1	10–100	0,02–0,05	1	3–5
МОП КМОП	1–10 0,01–0,1	20–200 10–50	2–3 1–2	2–8	10–20

Как видно из таблицы, наиболее быстродействующими являются в настоящее время схемы ЭСЛ и ТТЛШ, наиболее экономичными — схемы $И^2Л$ и КМОП.

Функциональные цифровые узлы комбинационного типа

Интегральные логические элементы являются основой для построения цифровых устройств, которые выполняют уже более сложные операции и относятся к классу комбинационных устройств. Основные из них: двоичные сумматоры; дешифраторы и шифраторы; мультиплексоры и демультиплексоры; цифровые компараторы; преобразователи кодов и др.

Сумматоры. Построение двоичных сумматоров обычно начинается с сумматора по модулю два. На рис. 2.14 приведена таблица истинности этого сумматора.

X	Y	Out
0	0	0
0	1	1
1	0	1
1	1	0

Рис. 2.14. Таблица истинности сумматора по модулю два

В соответствии с принципами построения произвольной таблицы истинности получим схему сумматора по модулю два. Эта схема приведена на рис. 2.15.

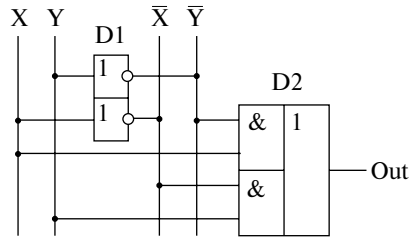


Рис. 2.15. Принципиальная схема, реализующая таблицу истинности сумматора по модулю два

Сумматор по модулю два (схема исключающего «ИЛИ») изображается на схемах, как показано на рис. 2.16.

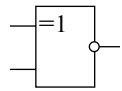


Рис. 2.16. Изображение схемы, выполняющей логическую функцию исключающего «ИЛИ»

Сумматор по модулю два выполняет суммирование без учета переноса. В обычном двоичном сумматоре требуется учитывать перенос, поэтому требуются схемы, позволяющие формировать перенос в следующий двоичный разряд. Таблица истинности такой схемы, называемой полусумматором, приведена на рис. 2.17.

A	B	S	PO
0	0	0	0
0	1	1	0
1	0	1	0
1	1	0	1

Рис. 2.17. Таблица истинности полусумматора

В соответствии с принципами построения произвольной таблицы истинности получим схему полусумматора. Эта схема приведена на рис. 2.18.

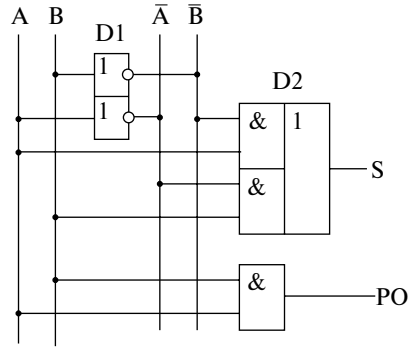


Рис. 2.18. Принципиальная схема, реализующая таблицу истинности полусумматора

Полусумматор изображается на схемах, как показано на рис. 2.19.

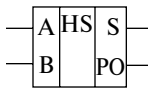


Рис. 2.19. Изображение полусумматора на схемах

Схема полусумматора формирует перенос в следующий разряд, но не может учитывать перенос из предыдущего разряда, поэтому она и называется полусумматором. Таблица истинности полного двоичного одноразрядного сумматора приведена на рис. 2.20.

В соответствии с принципами построения схемы по произвольной таблице истинности получим схему полного двоичного одноразрядного сумматора. Эта схема приведена на рис. 2.21.

PI	A	B	S	PO
0	0	0	0	0
0	0	1	1	0
0	1	0	1	0
0	1	1	0	1
1	0	0	1	0
1	0	1	0	1
1	1	0	0	1
1	1	1	1	1

Рис. 2.20. Таблица истинности полного двоичного одноразрядного сумматора

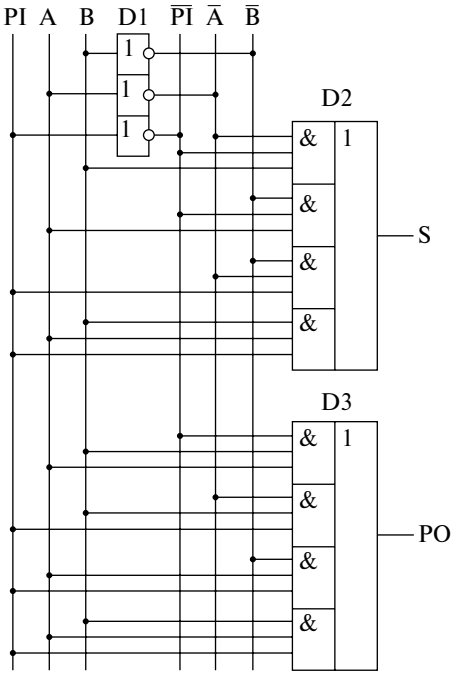


Рис. 2.21. Принципиальная схема, реализующая таблицу истинности полного двоичного одноразрядного сумматора

Полный двоичный одноразрядный сумматор изображается на схемах, как показано на рис. 2.22.

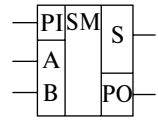


Рис. 2.22. Изображение полного двоичного одноразрядного сумматора на схемах

Для того чтобы получить многоразрядный сумматор, необходимо соединить входы и выходы переносов соответствующих двоичных разрядов. Схема соединения приведена на рис. 2.23.

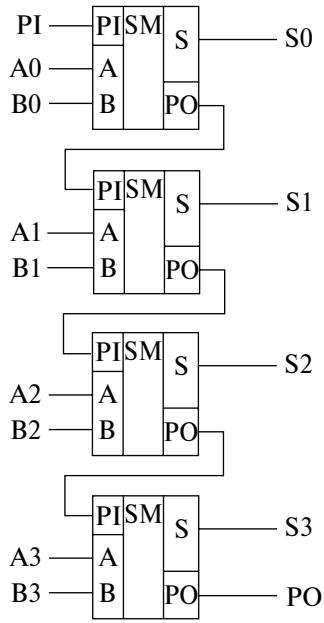


Рис. 2.23. Принципиальная схема многоразрядного двоичного сумматора

Полный двоичный многоразрядный сумматор изображается на схемах, как показано на рис. 2.24.

Естественно, в этой схеме рассматриваются только принципы работы двоичных сумматоров. В реальных схемах для увеличения скорости работы применяется отдельная схема формирования переносов для каждого двоичного разряда. Таблицу истинности для такой схемы легко получить из принципов суммирования двоичных чисел, а затем применить хорошо известные нам принципы построения схемы по произвольной таблице истинности.

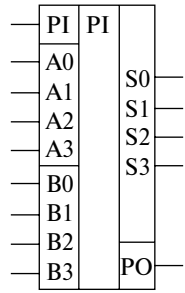


Рис. 2.24. Изображение полного двоичного многоразрядного сумматора на схемах

Цифровые компараторы. *Компаратор* — устройство сравнения кодов чисел. В общем случае компаратор параллельных кодов двух m-разрядных двоичных чисел представляет собой комбинационную схему с 2^m входами и тремя выходами («равно», «больше», «меньше»). При поступлении на входы кодов двух сравниваемых чисел сигнал логической единицы появляется только на одном из выходов. В некоторых случаях компаратор может иметь менее трех выходов.

Одноразрядный компаратор имеет два входа, на которые одновременно поступают одноразрядные двоичные числа x_1 и x_2, и три выхода (=, >, <).

Из таблицы истинности логические уравнения компаратора при сравнении x_1 с x_2 получаются в виде:

x_1	x_2	$y^=$	$y^>$	$y^<$
0	0	1	0	0
0	1	0	0	1
1	0	0	1	0
1	1	1	0	0

Реализация такого компаратора в базисе «И-НЕ» приводит к следующей схеме (рис. 2.25).

Многоразрядные компараторы обычно выполняют на базе одноразрядных. При этом используется принцип последовательного сравнения разрядов многоразрядных чисел, начиная с их старших разрядов, так как уже на этом этапе, если $x_{1m}\ 4 \neq x_{2m}$, задача может быть решена однозначно и сравнение следующих за старшими разрядов не потребуется.

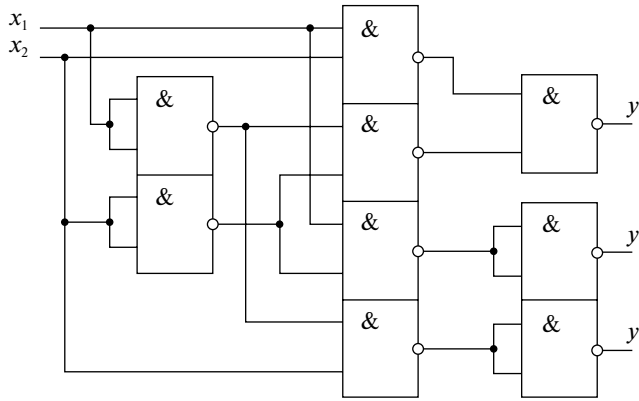


Рис. 2.25. Одноразрядный компаратор двоичных чисел

Мультиплексоры и демультиплексоры. Мультиплексорами называются устройства, которые позволяют подключать несколько входов к одному выходу. **Демультиплексорами** называются устройства, которые позволяют подключать один вход к нескольким выходам. В простейшем случае такую коммутацию можно осуществить при помощи ключей (рис. 2.26).

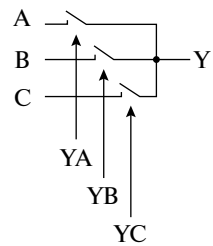


Рис. 2.26. Коммутатор (мультиплексор), собранный на ключах

В цифровых схемах требуется управлять ключами при помощи логических уровней. То есть нужно подобрать устройство, которое могло бы выполнять функции электронного ключа с электронным управлением цифровым сигналом.

Рассмотрим таблицу истинности логического элемента «И-НЕ» (рис. 2.27).

Y	X	Out
0	0	1
0	1	1
1	0	1
1	1	0

Рис. 2.27. Таблица истинности логического элемента «И-НЕ»

Теперь один из входов элемента будем рассматривать как информационный вход электронного ключа, а другой вход — как управляющий. По таблице истинности отчетливо видно, что, пока на управляющий вход *Y* подан логический уровень «0», сигнал со входа X на выход Out не проходит. При подаче на управляющий вход Y логической «1», сигнал, поступающий на вход X, поступает на выход Out. То есть логический элемент «И» можно использовать в качестве электронного ключа. При этом неважно, какой из входов элемента «И» будет использоваться в качестве управляющего входа, а какой — в качестве информационного. Остается только объединить выходы элеметов «И» на один выход. Это делается при помощи элемента «ИЛИ». Такая схема коммутатора приведена на рис. 2.28.

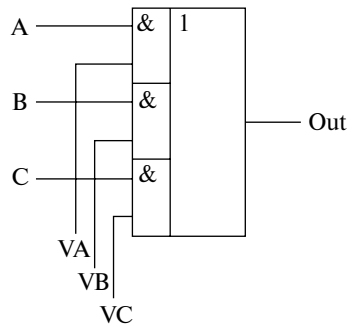


Рис. 2.28. Схема мультиплексора
на логических элементах

В этой схеме можно одновременно включать несколько входов на один выход. Однако обычно это приводит к непредсказуемым последствиям. Кроме того, для управления требуется много входов, поэтому в состав мультиплексора включают дешифратор. Это позволяет управлять переключением входов микросхемы на выход при помощи двоичных кодов (рис. 2.29).

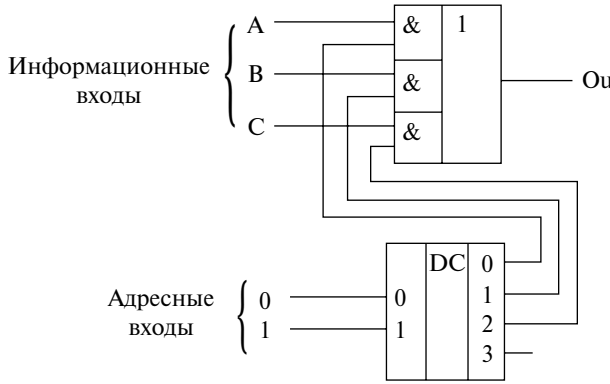


Рис. 2.29. Принципиальная схема мультиплексора, управляемого двоичным кодом

Мультиплексор изображается на принципиальных схемах, как показано на рис. 2.30.

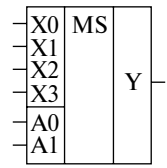


Рис. 2.30. Обозначение мультиплексора на принципиальных схемах

Задача передачи сигнала с одного входа микросхемы на один из нескольких выходов называется *демультиплексированием*. Демультиплексор можно построить на основе точно таких же схем логического «И». Существенным отличием от мультиплексора является возможность объединения нескольких входов в один без дополнительных схем. Для выбора конкретного выхода демультиплексора, как и в мультиплексоре, используется двоичный дешифратор. Схема демультиплексора приведена на рис. 2.31.

Декодеры. Декодеры (дешифраторы) позволяют преобразовывать одни виды двоичных кодов в другие. Преобразование производится по правилам, описанным в таблицах истинности, поэтому построение дешифраторов не представляет трудностей. Для построения дешифратора можно воспользоваться правилами построения схемы для произвольной таблицы истинности.

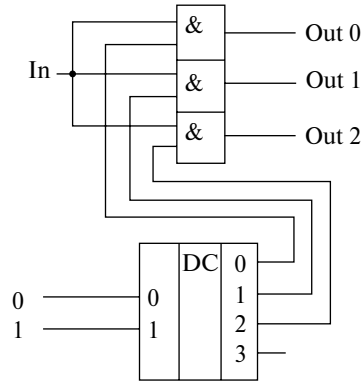


Рис. 2.31. Принципиальная схема демультиплексора, управляемого двоичным кодом

Рассмотрим пример построения декодера из двоичного кода в десятичный. Таблица истинности такого декодера приведена на рис. 2.32.

Входы				Выходы									
8	4	2	1	0	1	2	3	4	5	6	7	8	9
0	0	0	0	1	0	0	0	0	0	0	0	0	0
0	0	0	1	0	1	0	0	0	0	0	0	0	0
0	0	1	0	0	0	1	0	0	0	0	0	0	0
0	0	1	1	0	0	0	1	0	0	0	0	0	0
0	1	0	0	0	0	0	0	1	0	0	0	0	0
0	1	0	1	0	0	0	0	0	1	0	0	0	0
0	1	1	0	0	0	0	0	0	0	1	0	0	0
0	1	1	1	0	0	0	0	0	0	0	1	0	0
1	0	0	0	0	0	0	0	0	0	0	0	1	0
1	0	0	1	0	0	0	0	0	0	0	0	0	1

Рис. 2.32. Таблица истинности десятичного декодера

В соответствии с принципами построения произвольной таблицы истинности получим схему декодера. Эта схема приведена на рис. 2.33.

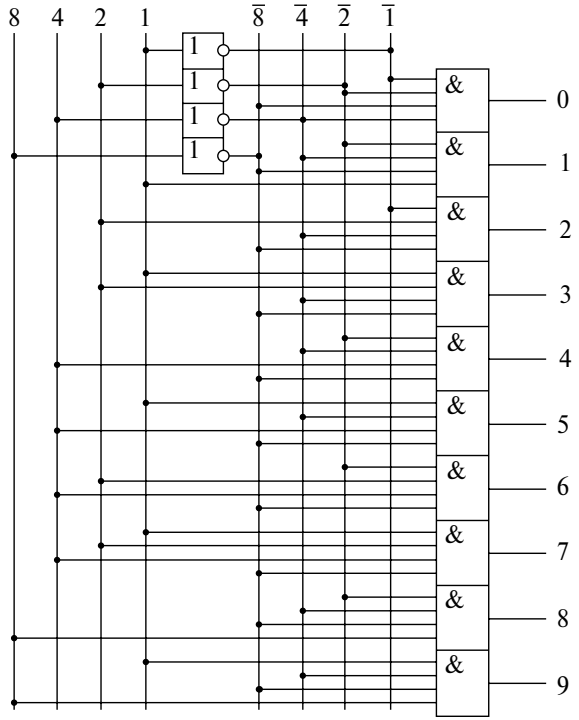


Рис. 2.33. Принципиальная схема двоично-десятичного декодера

Точно так же можно получить схему для любого другого декодера или дешифратора.

Шинные формирователи. Мультиплексоры предназначены для объединения нескольких выходов в тех случаях, когда заранее известно, сколько выходов нужно объединять. Часто это неизвестно. Более того, часто количество объединяемых микросхем изменяется в процессе эксплуатации устройств. Наиболее яркий пример — компьютеры, в которых в процессе эксплуатации изменяется объем оперативной памяти, количество портов ввода-вывода, количество дисководов. В таких случаях невозможно для объединения нескольких выходов воспользоваться логическим элементом «ИЛИ».

Для объединения нескольких выходов на один вход, в случае когда заранее неизвестно, сколько микросхем нужно объединять, используется два способа:

- монтажное «ИЛИ»;
- шинные формирователи.

Исторически первой схемой объединения выходов были схемы с открытым коллектором (монтажное «ИЛИ»). Схема монтажного «ИЛИ» приведена на рис. 2.34.

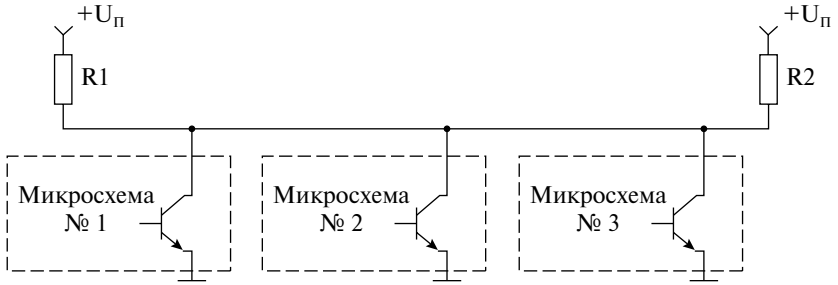


Рис. 2.34. Схема монтажного «ИЛИ»

Такое объединение микросхем называется **шиной** и позволяет объединять до 10 микросхем на один провод. Естественно, для того, чтобы микросхемы не мешали друг другу, только одна из микросхем должна выдавать информацию на общий провод. Остальные микросхемы в этот момент времени должны быть отключены от шины (т.е. выходной транзистор должен быть закрыт). Это обеспечивается внешней микросхемой управления, не показанной на данном рисунке.

На принципиальных схемах такие элементы обозначаются, как показано на рис. 2.35.

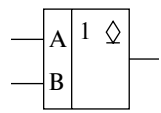


Рис. 2.35. Обозначение микросхемы с открытым коллектором на выходе

Недостатком приведенной схемы объединения нескольких микросхем на один провод является низкая скорость передачи информации, обусловленная затягиванием переднего фронта. Это явление связано с различным сопротивлением заряда и разряда паразитной емкости шины. Заряд паразитной емкости происходит через сопротивления *R*1 и *R*2, которые много больше сопротивления открытого транзистора. Величину этого сопротивления невозможно сделать менее некоторого предела, определяемого напряжением низкого

уровня, который определяется, в свою очередь, допустимым током потребления всей схемы в целом. Временная диаграмма напряжения на шине с общим коллектором приведена на рис. 2.36.

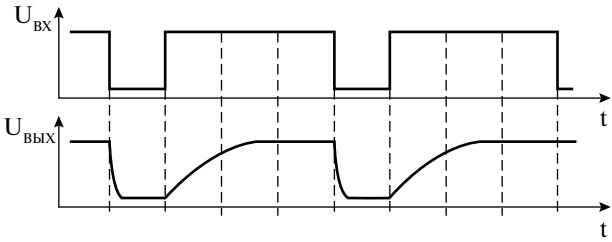


Рис. 2.36. Временные диаграммы напряжения на входе и выходе микросхемы с открытым коллектором

Естественным решением этой проблемы было бы включение транзистора в верхнее плечо схемы, но при этом возникает проблема сквозных токов, из-за которой невозможно соединять выходы цифровых микросхем непосредственно. Причина возникновения сквозных токов поясняется на рис. 2.37.

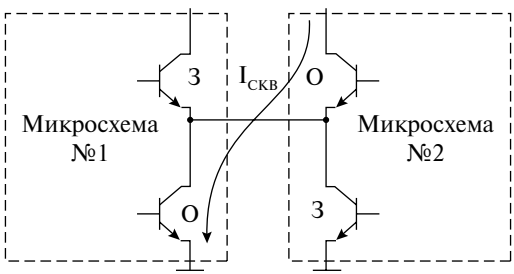


Рис. 2.37. Путь протекания сквозного тока при непосредственном соединении выходов цифровых микросхем

Эта проблема исчезает, если появляется возможность закрывать транзисторы как в верхнем, так и в нижнем плече выходного каскада. Если в микросхеме закрыты оба транзистора, то такое состояние выхода микросхемы называется третьим состоянием или *z*-состоянием выхода микросхемы. Такая возможность появляется в специализированных микросхемах с третьим состоянием на выходе микросхемы. Принципиальная схема выходного каскада микросхемы с тремя состояниями на выходе микросхемы приведена на рис. 2.38.

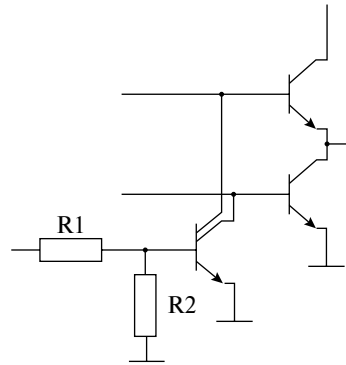


Рис. 2.38. Принципиальная схема выходного каскада микросхемы с тремя состояниями на выходе

На принципиальных схемах такие элементы обозначаются, как показано на рис. 2.39.

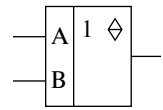


Рис. 2.39. Обозначение микросхемы с тремя состояниями на выходе

Часто в микросхеме, содержащей элементы с тремя состояниями выходного каскада, объединяют управляющие сигналы всех элементов в один провод. Такие микросхемы называют шинными формирователями и изображают на схемах, как показано на рис. 2.40.

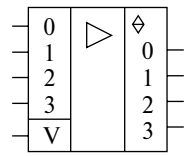


Рис. 2.40. Обозначение шинного формирователя

Последовательностные устройства

Триггеры. Простейшая схема, позволяющая запоминать двоичную информацию, строится на основе простейших логических элементов «ИЛИ» или «И». Такая схема, построенная на элементах «И», приведена на рис. 2.41. Вход S (Set) позволяет устанавливать выход триг-

гера Q в единичное состояние при подаче на его вход логического нуля. Вход R (Reset) позволяет сбрасывать выход триггера Q в нулевое состояние при подаче на его вход логического нуля.

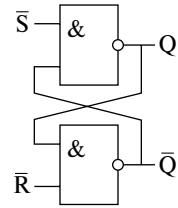


Рис. 2.41. Схема простейшего триггера на схемах «И»

Входы R и S — инверсные (активный уровень «0»).

Точно так же можно построить RS-триггер и на логических элементах «ИЛИ». Схема RS-триггера, построенного на логических элементах «ИЛИ», приведена на рис. 2.42. Единственное отличие будет заключаться в том, что сброс и установка триггера будут производиться единичными логическими уровнями.

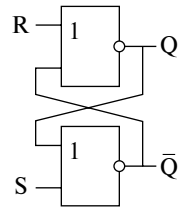


Рис. 2.42. Схема простейшего триггера на схемах «ИЛИ».
Входы R и S — прямые (активный уровень «1»)

Так как триггер при построении его на различных элементах работает одинаково, то его изображение на принципиальных схемах тоже одинаково. Изображение простейшего триггера на принципиальных схемах приведено на рис. 2.43.

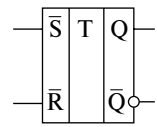


Рис. 2.43. Обозначение простейшего триггера
на принципиальных схемах

Схема триггера позволяет запоминать состояние логической схемы, но так как в начальный момент времени может возникать переходный процесс (в цифровых схемах этот процесс называется «опасные гонки»), то запоминать состояния логической схемы нужно только в определенные моменты времени, когда все переходные процессы закончены, т.е. цифровые схемы требуют синхросигнала. Все переходные процессы должны закончиться за время периода синхросигнала.

Для таких цифровых схем требуются синхронные триггеры. Схема синхронного триггера приведена на рис. 2.44, а обозначение на принципиальных схемах — на рис. 2.45.

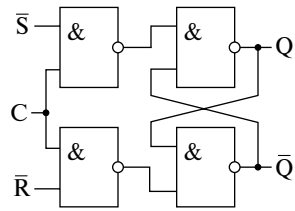


Рис. 2.44. Схема синхронного триггера на схемах «И»

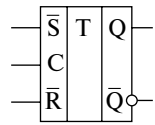


Рис. 2.45. Обозначение синхронного триггера на принципиальных схемах

В приведенной схеме для записи логического нуля и логической единицы требуются разные входы, что не всегда удобно. Поэтому для запоминания дискретной информации применяются D-триггеры. Схема такого триггера приведена на рис. 2.46, а обозначение на принципиальных схемах — на рис. 2.47.

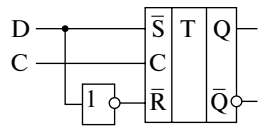


Рис. 2.46. Схема D-триггера (защелки)

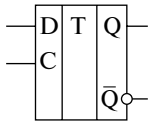


Рис. 2.47. Обозначение D-триггера (защелки) на принципиальных схемах

Во всех приведенных схемах синхросигнал работает по уровню, поэтому триггеры называются «триггеры-защелки». Легче всего объяснить появление этого названия по временной диаграмме, приведенной на рис. 2.48.

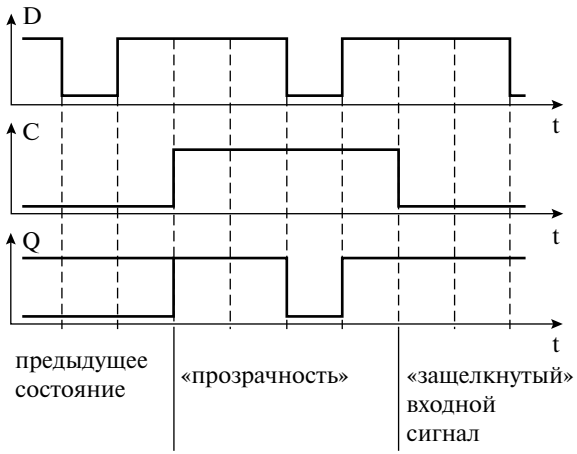


Рис. 2.48. Временная диаграмма D триггера (защелки)

По этой временной диаграмме видно, что триггер-защелка хранит данные на выходе только при нулевом уровне на входе синхронизации. Если же на вход синхронизации подать активный высокий уровень, то напряжение на выходе триггера будет повторять напряжение, подаваемое на вход этого триггера. Входное напряжение запоминается только в момент изменения уровня напряжения на входе синхронизации C с высокого уровня на низкий уровень. Входные данные как бы «защелкиваются» в этот момент, отсюда и название — «триггер-защелка».

Принципиально в этой схеме входной переходной процесс может беспрепятственно проходить на выход триггера. Поэтому там, где это важно, необходимо сокращать длительность импульса синхронизации до минимума. Чтобы преодолеть такое ограничение, были

разработаны триггеры, работающие по фронту. Схема такого триггера приведена на рис. 2.49, а обозначение на принципиальных схемах — на рис. 2.50.

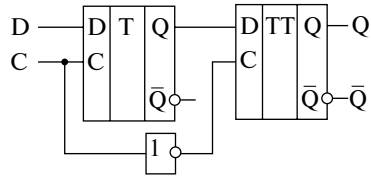


Рис. 2.49. Схема универсального D-триггера

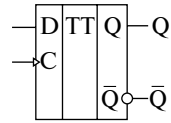


Рис. 2.50. Обозначение универсального D-триггера на принципиальных схемах

Счетчики. Счетчик числа импульсов — устройство, на выходах которого получается двоичный (двоично-десятичный) код, определяемый числом поступивших импульсов. Счетчики могут строиться на двухступенчатых D-триггерах, T-триггерах и JK-триггерах.

Основной параметр счетчика — модуль счета — максимальное число единичных сигналов, которое может быть сосчитано счетчиком. Счетчики обозначают через CT (от англ. counter).

Таблица истинности двоичного счетчика — последовательность двоичных чисел от нуля до 2^n-1, где n — разрядность счетчика. Наблюдение за разрядами чисел, составляющих таблицу, приводит к пониманию структурной схемы двоичного счетчика. Состояния младшего разряда при его просмотре по соответствующему столбцу таблицы показывают чередование нулей и единиц вида 01010101..., что естественно, так как младший разряд принимает входной сигнал и переключается от каждого входного воздействия. В следующем разряде наблюдается последовательность пар нулей и единиц вида 00110011.... В третьем разряде образуется последовательность из четверок нулей и единиц 00001111... и т.д. Из этого наблюдения видно, что следующий по старшинству разряд переключается с частотой, в два раза меньшей, чем данный.

Так как счетный триггер делит частоту входных импульсов на два, то, сопоставив этот факт с указанной выше закономерностью, видим,

что счетчик может быть построен в виде цепочки последовательно включенных счетных триггеров. Заметим, кстати, что согласно ГОСТу входы элементов изображаются слева, а выходы — справа. Соблюдение этого правила ведет к тому, что в числе, содержащемся в счетчике, младшие разряды расположены левее старших.

Пример асинхронного трехразрядного двоичного суммирующего счетчика приведен на рис. 2.51, а его условно-графическое обозначение — на рис. 2.52. Для построения этого счетчика использованы JK-триггеры с динамической синхронизацией по спаду синхросигнала. Каждый JK-триггер в счетчике включен в режим инвертирования своего состояния при переключении синхросигнала с высокого уровня на низкий. Идеализированная временная диаграмма работы этого счетчика показана на рис. 2.53.

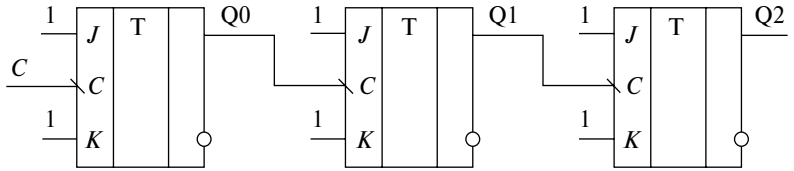


Рис. 2.51. Схема асинхронного трехразрядного счетчика

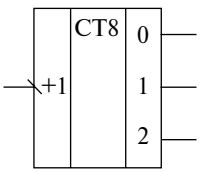


Рис. 2.52. Условно-графическое обозначение трехразрядного суммирующего счетчика

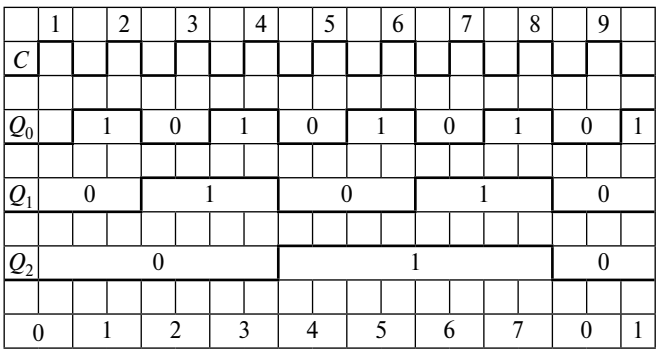


Рис. 2.53. Временная диаграмма работы счетчика

Быстродействие асинхронного счетчика определяется максимальной задержкой от изменения сигнала на его счетном входе до полного установления состояния всех его выходов. Проведем оценку быстродействия на примере переключения выходов счетчика после поступления восьмого синхросигнала на его вход (рис. 2.54), так как именно в этом такте время переключения выходов счетчика будет максимальным.

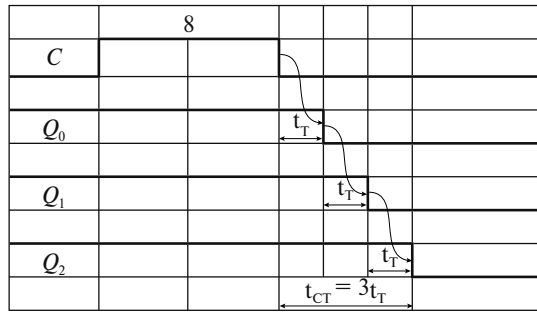


Рис. 2.54. Оценка быстродействия асинхронного счетчика

По фронту 1–0 сигнала $C(8)$ с задержкой сигнала, равной t_T, на триггере Q_0 происходит изменение сигнала на выходе Q_0. Это изменение, в свою очередь, приведет к переключению сигнала Q_1 с соответствующей задержкой относительно переключения Q_0. Вслед за этим с задержкой сигнала на следующем триггере переключится сигнал Q_2. То есть общее время задержки переключения сигналов на выходе трехразрядного счетчика составит $3t_T$. Очевидно, что для n-разрядного счетчика время задержки составит

$$t_{CT} = nxt_T.$$

Таким образом, с увеличением разрядности асинхронного счетчика увеличивается его задержка и, следовательно, уменьшается быстродействие.

Этот недостаток устраняется в счетчиках, работающих по синхронной схеме. В них за счет дополнительных управляющих комбинационных схем обеспечивается одновременное переключение всех разрядов при поступлении сигнала на счетный вход (с задержкой, равной задержке одного триггера вне зависимости от разрядности счетчика).

Обычно счетчик имеет вход установки в нулевое состояние (асинхронный сброс составляющих его триггеров в ноль). Некоторые счетчики имеют цепи установки в произвольное начальное состояние, с которого уже будет начинаться операция счета.

Регистры. *Регистр хранения.*

Регистр — внутреннее запоминающее устройство процессора или внешнего устройства, предназначенное для временного хранения обрабатываемой или управляющей информации. Регистры представляют собой совокупность триггеров, количество которых равняется разрядности регистра, и вспомогательных схем, обеспечивающих выполнение некоторых элементарных операций. Набор этих операций в зависимости от функционального назначения регистра может включать в себя одновременную установку всех разрядов регистра в «0», параллельную или последовательную загрузку регистра, сдвиг содержимого регистра влево или вправо на требуемое число разрядов, управляемую выдачу информации из регистра (обычно используется при работе нескольких схем на общую шину данных) и т.д.

Регистры хранения используются для приема, хранения и выдачи многоразрядного кода. Они представляют собой совокупность одноступенчатых триггеров (как правило, D-типа) с общим входом синхронизации. Иногда в регистре имеется также и общий вход асинхронной установки всех триггеров в «0». Схема четырехразрядного регистра хранения приведена на рис. 2.55, а его условно-графическое обозначение — на рис. 2.56.

Регистр сдвига. Регистр сдвига — регистр, обеспечивающий помимо хранения информации сдвиг влево или вправо всех разрядов одновременно на одинаковое число позиций. При этом выдвигаемые за пределы регистра разряды теряются, а в освобождающиеся разряды заносится информация, поступающая по отдельному внешнему входу регистра сдвига. Обычно эти регистры обеспечивают сдвиг кода на одну позицию влево или вправо. Но существуют и универсальные регистры сдвига, которые выполняют сдвиг как влево, так и вправо в зависимости от значения сигнала на специальном управляющем входе или при подаче синхросигналов на разные входы регистра. Регистр сдвига может быть спроектирован и таким образом, чтобы выполнять сдвиг одновременно не на одну, а на несколько позиций.

Регистры сдвига строятся на двухступенчатых триггерах. Схема четырехразрядного регистра, выполняющего сдвиг на один разряд от разряда 0 к разряду 3, показана на рис. 2.57, а его условно-графическое обозначение — на рис. 2.58. Ввод информации в данный регистр — последовательный через внешний вход D_0. Регистр имеет вход асинхронной установки всех разрядов в «0». Для наглядности каждый двухступенчатый регистр представлен двумя одноступенча-

тыми с соответствующей организацией синхронизации первой и второй ступеней. Пунктиром обозначен реальный двухступенчатый триггер.

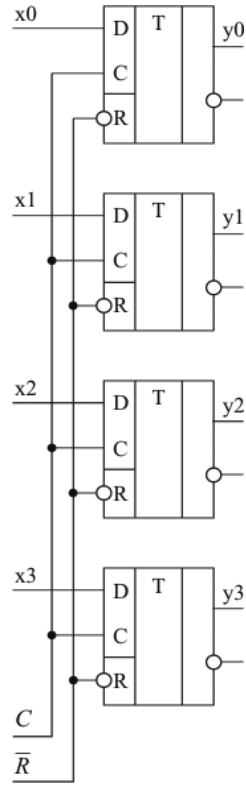


Рис. 2.55. Структура четырехразрядного регистра хранения с асинхронным входом установки в «0»

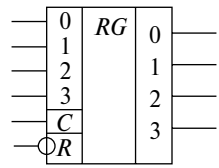


Рис. 2.56. Условно-графическое обозначение четырехразрядного регистра хранения с асинхронным входом установки в «0»

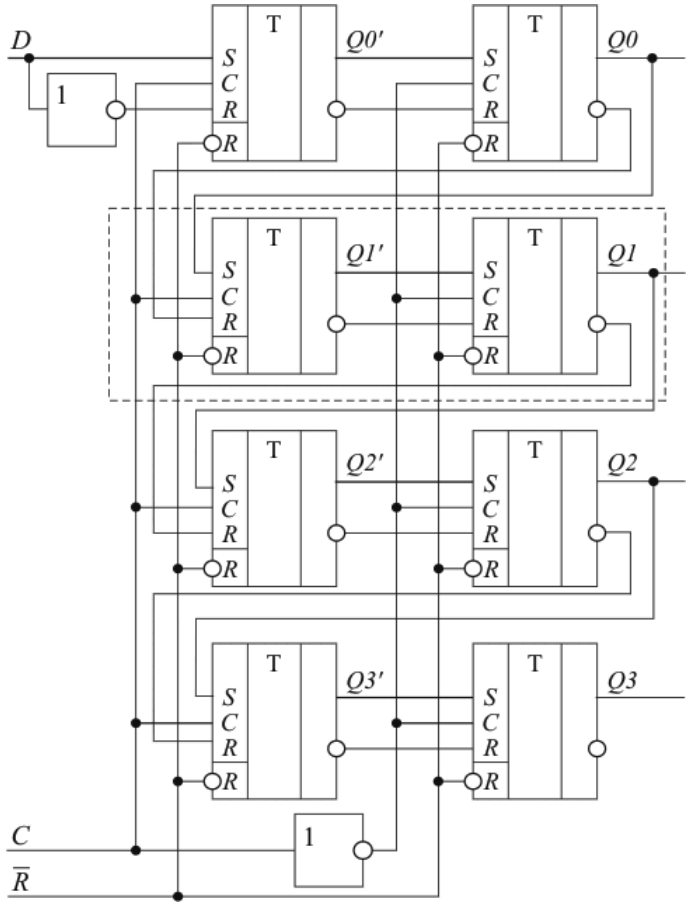


Рис. 2.57. Структура регистра сдвига

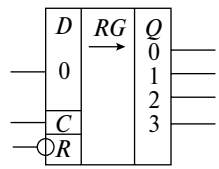


Рис. 2.58. Условно-графическое обозначение четырехразрядного регистра сдвига с асинхронным входом установки в «0»

Идеализированная временная диаграмма работы регистра сдвига, структура которого представлена на рис. 2.58, показана на рис. 2.59.

Предполагаем, что начальное состояние регистра следующее: Q_0=0, Q_1=1, Q_2=1, Q_3=0.

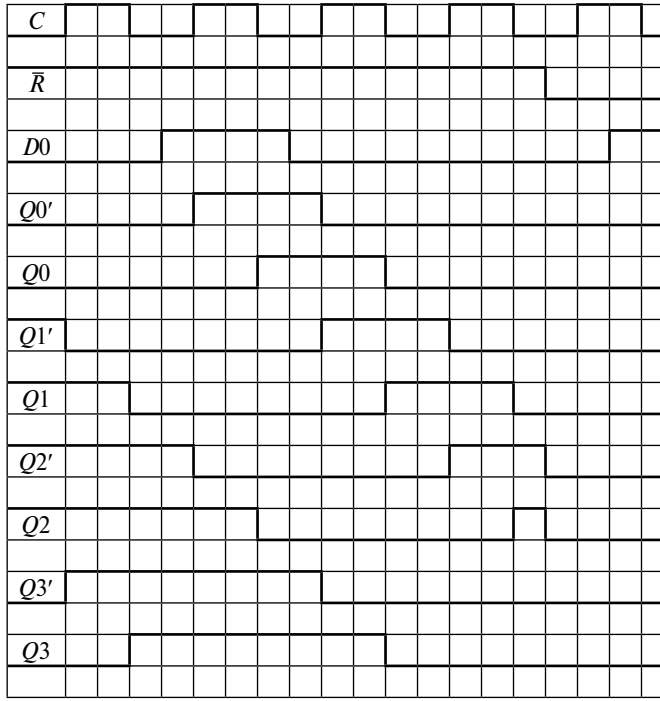


Рис. 2.59. Временная диаграмма работы регистра сдвига

Работа регистра сдвига в каждом периоде сигнала синхронизации разбивается на две фазы: при высоком и при низком значении синхросигнала.

При *высоком* уровне синхросигнала проводится запись значения выхода $(i-1)$-го разряда регистра в первую ступень i-го разряда. Вторая ступень каждого разряда сохраняет свое прежнее значение. В этой фазе состояние первой ступени i-го триггера повторяет состояние второй ступени $(i-1)$-го триггера. Вторые ступени каждого триггера, а следовательно, и выходы регистра в целом остаются неизменными.

При *низком* уровне синхросигнала значение, записанное в первой ступени каждого триггера, перезаписывается в его вторую ступень. Запись в первую ступень триггера запрещена. В этой фазе состояния первой и второй ступеней каждого триггера становятся одинаковыми.

Поступление сигнала $R = 0$ вне зависимости от значения сигнала на входе синхронизации C и сигнала на входе D_0 устанавливает все разряды регистра в нулевое состояние.

Запоминающие устройства

Постоянные запоминающие устройства (ПЗУ). Очень часто в различных применениях требуется хранение информации, которая не изменяется в процессе эксплуатации устройства. Это такая информация, как программы в микроконтроллерах, начальные загрузчики и BIOS в компьютерах, таблицы коэффициентов цифровых фильтров в сигнальных процессорах. Практически всегда эта информация не требуется одновременно, поэтому простейшие устройства для запоминания постоянной информации можно построить на мультиплексорах. Схема такого постоянного запоминающего устройства (ПЗУ) приведена на рис. 2.60.

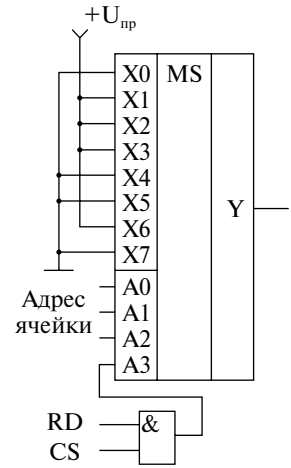


Рис. 2.60. Схема ПЗУ, построенная на мультиплексоре

В этой схеме построено постоянное запоминающее устройство на восемь одноразрядных ячеек. Запоминание конкретного бита в одноразрядную ячейку производится запайкой провода к источнику питания (запись единицы) или запайкой провода к корпусу (запись нуля). На принципиальных схемах такое устройство обозначается, как показано на рис. 2.61.

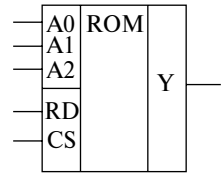


Рис. 2.61. Обозначение ПЗУ на принципиальных схемах

Для того чтобы увеличить разрядность ячейки памяти ПЗУ, эти микросхемы можно соединять параллельно (выходы и записанная информация, естественно, остаются независимыми). Схема параллельного соединения одноразрядных ПЗУ приведена на рис. 2.62.

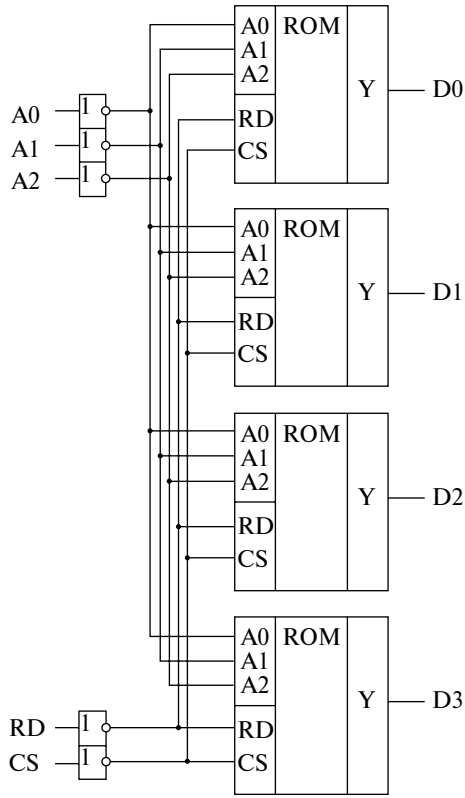


Рис. 2.62. Схема многоразрядного ПЗУ

В реальных ПЗУ запись информации производится при помощи последней операции производства микросхемы — металлизации. Металлизация производится при помощи маски, поэтому такие ПЗУ получили название масочных ПЗУ. Еще одно отличие реальных микросхем от упрощенной модели, приведенной выше, — это использование кроме мультиплексора еще и демультиплексора. Такое решение позволяет превратить одномерную запоминающую структуру в многомерную и тем самым существенно сократить объем схемы дешифратора, необходимого для работы схемы ПЗУ. Эта ситуация иллюстрируется рис. 2.63.

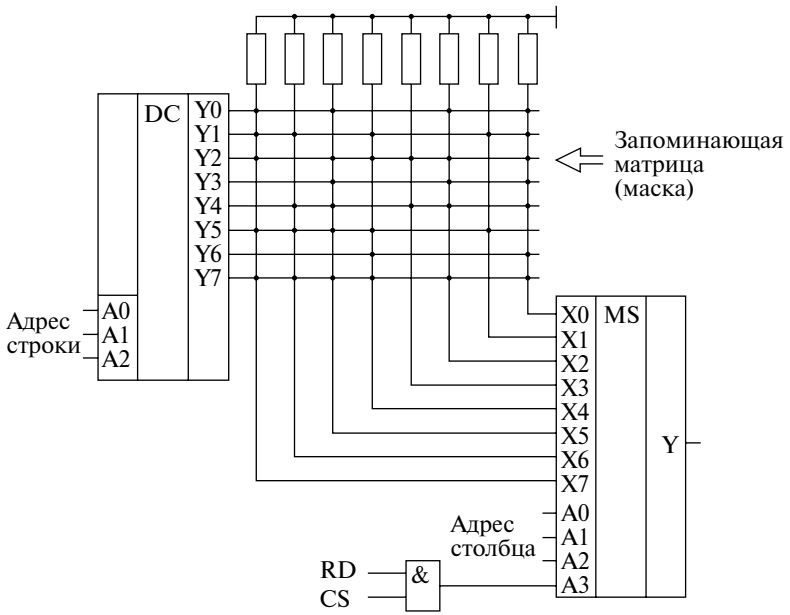


Рис. 2.63. Схема масочного ПЗУ

Масочные ПЗУ изображаются на принципиальных схемах, как показано на рис. 2.64. Адреса ячеек памяти в этой микросхеме подаются на выводы A0–A9. Микросхема выбирается сигналом CS. При помощи этого сигнала можно наращивать объем ПЗУ (пример использования сигнала CS приведен при обсуждении ОЗУ). Чтение микросхемы производится сигналом RD.

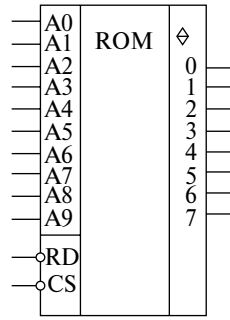


Рис. 2.64. Обозначение масочного ПЗУ на принципиальных схемах

Программирование масочного ПЗУ производится на заводе-изготовителе, что очень неудобно для мелких и средних серий производства, не говоря уже о стадии разработки устройства. Естественно, что для крупносерийного производства масочные ПЗУ являются самым дешевым видом ПЗУ и поэтому широко применяются в настоящее время. Для мелких и средних серий производства радиоаппаратуры были разработаны микросхемы, которые можно программировать в специальных устройствах — программаторах. В этих микросхемах постоянное соединение проводников в запоминающей матрице заменяется плавкими перемычками, изготовленными из поликристаллического кремния. При производстве микросхемы изготавливаются все перемычки, что эквивалентно записи во все ячейки памяти логических единиц. В процессе программирования на выводы питания и выходы микросхемы подается повышенное питание. При этом если на выход микросхемы подается напряжение питания (логическая единица), то через перемычку ток протекать не будет и перемычка останется неповрежденной. Если же на выход микросхемы подать низкий уровень напряжения (присоединить к корпусу), то через перемычку будет протекать ток, который испарит эту перемычку и при последующем считывании информации из этой ячейки будет считываться логический ноль.

Такие микросхемы называются программируемыми ПЗУ (ППЗУ) и изображаются на принципиальных схемах, как показано на рис. 2.65. В качестве примера можно назвать микросхемы 155РЕ 3, 556РТ 4, 556РТ 8 и др.

Программируемые ПЗУ оказались очень удобны при мелкосерийном и среднесерийном производстве. Однако при разработке радиоэлектронных устройств часто приходится менять записываемую в ПЗУ программу. ППЗУ при этом невозможно использовать по-

вторно, поэтому раз записанное ПЗУ при ошибочной или промежуточной программе приходится выкидывать, что, естественно, повышает стоимость разработки аппаратуры. Для устранения этого недостатка был разработан еще один вид ПЗУ, который может стираться и программироваться заново.

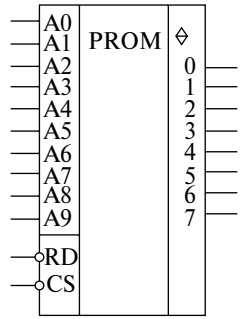


Рис. 2.65. Обозначение программируемого ПЗУ на принципиальных схемах

Структурная схема ПЗУ не отличается от описанного ранее масочного ПЗУ. В репрограммируемых ПЗУ стирание ранее записанной информации осуществляется ультрафиолетовым излучением. Для того чтобы этот свет мог беспрепятственно проходить к полупроводниковому кристаллу, в корпус микросхемы встраивается окошко из кварцевого стекла.

В качестве примера таких микросхем можно назвать микросхемы серии 573 российского производства, микросхемы серий 27cXXX зарубежного производства. В этих микросхемах чаще всего хранятся программы BIOS универсальных компьютеров. Репрограммируемые ПЗУ (РПЗУ) изображаются на принципиальных схемах, как показано на рис. 2.66.

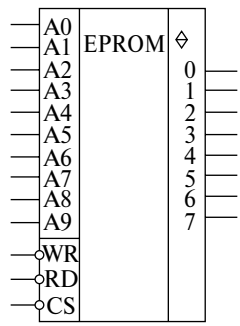


Рис. 2.66. Обозначение репрограммируемого ПЗУ на принципиальных схемах

Высокая стоимость корпусов с кварцевым окошком, а также малое количество циклов записи-стирания привели к поиску способов стирания информации из ППЗУ электрическим способом. На этом пути встретилось много трудностей, которые к настоящему времени практически решены. Сейчас достаточно широко распространены микросхемы с электрическим стиранием информации. В качестве запоминающей ячейки в них используются такие же ячейки, как и в РПЗУ, но они стираются электрическим потенциалом, поэтому количество циклов записи-стирания для этих микросхем достигает 1 млн раз. Время стирания ячейки памяти в таких микросхемах уменьшается до 10 мс. Схема управления для таких микросхем получилась сложная, поэтому наметилось два направления развития этих микросхем:

- ЕСППЗУ;
- FLASH-ПЗУ.

Электрически стираемые ППЗУ (ЭСППЗУ) дороже и меньше по объему, но зато позволяют перезаписывать каждую ячейку памяти отдельно. В результате эти микросхемы обладают максимальным количеством циклов записи-стирания. Область применения электрически стираемых ПЗУ — хранение данных, которые не должны стираться при выключении питания. К таким микросхемам относятся отечественные микросхемы 573РР3, 558РР и зарубежные микросхемы серии 28сХХ. Электрически стираемые ПЗУ обозначаются на схемах, как показано на рис. 2.67.

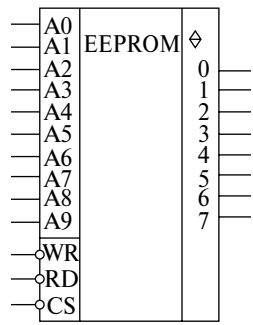


Рис. 2.67. Обозначение электрически стираемого ПЗУ на принципиальных схемах

В последнее время наметилась тенденция уменьшения габаритов ЭСППЗУ за счет уменьшения количества внешних ножек микросхем. Для этого адрес и данные передаются в микросхему и из микросхемы через последовательный порт. При этом используются два

вида последовательных портов — SPI-порт и I2C-порт (микросхемы 93cXX и 24cXX серий соответственно). Зарубежной серии 24cXX соответствует отечественная серия микросхем 558РРХ.

FLASH-ПЗУ отличаются от ЭСППЗУ тем, что стирание производится не каждой ячейки отдельно, а всей микросхемы в целом или блока запоминающей матрицы этой микросхемы, как это делалось в РПЗУ (рис. 2.68).

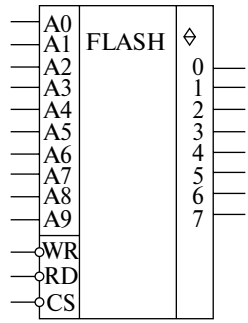


Рис. 2.68. Обозначение FLASH-памяти на принципиальных схемах

При обращении к постоянному запоминающему устройству сначала необходимо выставить адрес ячейки памяти на шине адреса, а затем произвести операцию чтения из микросхемы. Эта временная диаграмма приведена на рис. 2.69.

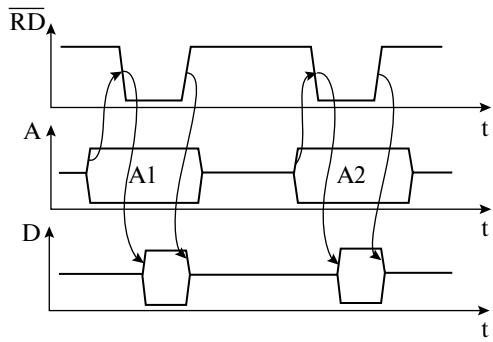


Рис. 2.69. Временная диаграмма чтения информации из ПЗУ

На рис. 2.69 стрелочками показана последовательность, в которой должны формироваться управляющие сигналы. На этом ри-

сунке RD — это сигнал чтения, A — сигналы выбора адреса ячейки (так как отдельные биты в шине адреса могут принимать разные значения, то показаны пути перехода как в единичное, так и в нулевое состояние), D — выходная информация, считанная из выбранной ячейки ПЗУ.

Статические оперативные запоминающие устройства (ОЗУ). В радиоаппаратуре часто требуется хранение временной информации, значение которой неважно при включении устройства. Такую память можно было бы построить на микросхемах EEPROM или FLASH-памяти, но, к сожалению, эти микросхемы дороги, обладают малым количеством перезаписей и чрезвычайно низким быстродействием при считывании и, особенно, записи информации. Для хранения временной информации можно воспользоваться параллельными регистрами. Так как запоминаемые слова не нужны одновременно, то можно воспользоваться механизмом адресации, который применяется в ПЗУ.

Схемы, в которых в качестве запоминающей ячейки используется параллельный регистр, называются статической ОЗУ, так как информация в ней сохраняется все время, пока к микросхеме подключено питание. В отличие от статической ОЗУ в микросхемах динамического ОЗУ постоянно требуется регенерировать их содержимое, иначе информация будет испорчена.

В микросхемах ОЗУ присутствуют две операции: записи и чтения. Для записи и чтения информации можно использовать различные шины данных (как это делается в сигнальных процессорах), но чаще используется одна и та же шина данных. Это позволяет экономить выводы микросхем, подключаемых к этой шине, и легко осуществлять коммутацию сигналов между различными устройствами.

Схема статического ОЗУ приведена на рис. 2.70. Вход и выход микросхемы в этой схеме объединены при помощи шинного формирователя. Естественно, что схемы реальных ОЗУ будут отличаться от приведенной на этом рисунке. Тем не менее, приведенная схема позволяет понять, как работает реальное ОЗУ. Изображение ОЗУ на принципиальных схемах приведено на рис. 2.71.

Сигнал записи WR позволяет записать логические уровни, присутствующие на информационных входах во внутреннюю ячейку ОЗУ. Сигнал чтения RD позволяет выдать содержимое внутренней ячейки памяти на информационные выходы микросхемы. В приведенной на рис. 2.71 схеме невозможно одновременно производить операцию записи и чтения, но обычно это и не нужно.

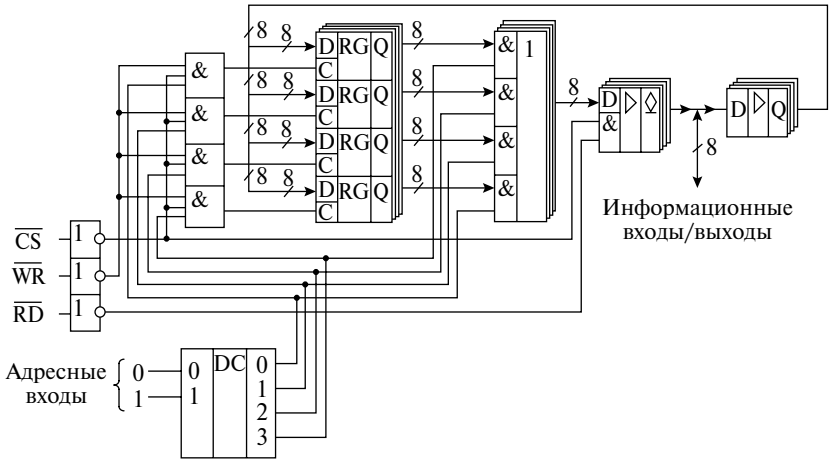


Рис. 2.70. Структурная схема ОЗУ

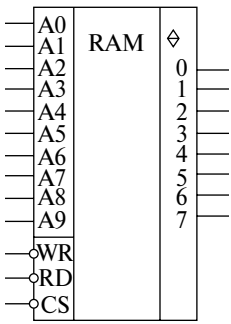


Рис. 2.71. Изображение ОЗУ на принципиальных схемах

Конкретная ячейка микросхемы выбирается при помощи двоичного кода — адреса ячейки. Объем памяти микросхемы зависит от количества ячеек, содержащихся в ней, или, что то же самое, от количества адресных проводов. Количество ячеек в микросхеме можно определить по количеству адресных проводов, возводя 2 в степень, равную количеству адресных выводов в микросхеме:

$M = 2^n$.

Вывод выбора кристалла CS позволяет объединять несколько микросхем для увеличения объема памяти ОЗУ. Такая схема приведена на рис. 2.72.

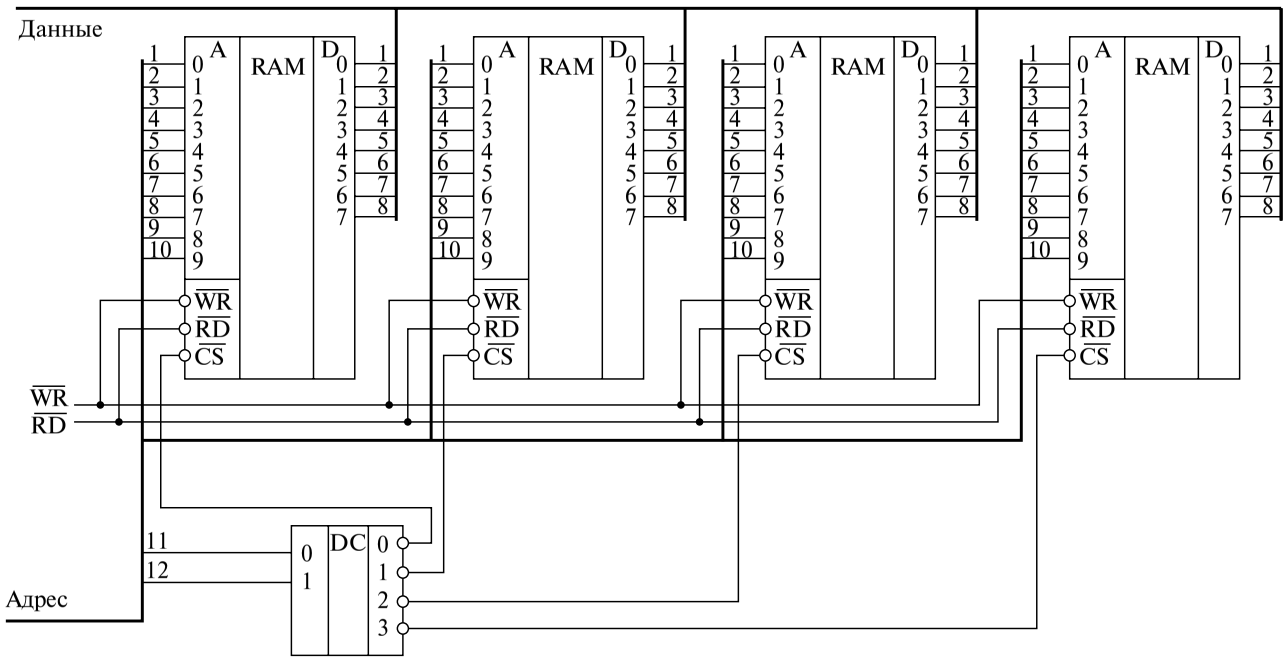


Рис. 2.72. Схема ОЗУ, построенного на нескольких микросхемах памяти

Статические ОЗУ требуют для своего построения большой площади кристалла, поэтому их емкость относительно невелика. Статические ОЗУ применяются для построения микроконтроллерных схем из-за простоты построения принципиальной схемы и возможности работать на сколь угодно низких частотах, вплоть до постоянного тока. Кроме того, статические ОЗУ применяются для построения кэш-памяти в универсальных компьютерах из-за высокого быстродействия статического ОЗУ.

Временные диаграммы чтения из статического ОЗУ совпадают с временными диаграммами чтения из ПЗУ. Временные диаграммы записи в статическое ОЗУ и чтения из него приведены на рис. 2.73.

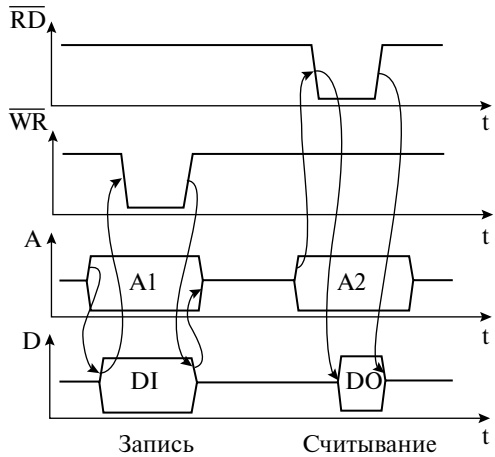


Рис. 2.73. Временная диаграмма обращения к ОЗУ, принятая для схем, совместимых со стандартом фирмы «Intel»

На рис. 2.73 стрелочками показана последовательность, в которой должны формироваться управляющие сигналы. На этом рисунке RD — это сигнал чтения; WR — сигнал записи; A — сигналы выбора адреса ячейки (так как отдельные биты в шине адреса могут принимать разные значения, то показаны пути перехода как в единичное, так и в нулевое состояние); DI — входная информация, предназначенная для записи в ячейку ОЗУ, расположенную по адресу A1; DO — выходная информация, считанная из ячейки ОЗУ, расположенной по адресу A2.

На рис. 2.74 стрелочками показана последовательность, в которой должны формироваться управляющие сигналы. На этом рисунке R/W — это сигнал выбора операции записи или чтения; DS — сигнал стробирования данных; A — сигналы выбора адреса ячейки (так как

отдельные биты в шине адреса могут принимать разные значения, то показаны пути перехода как в единичное, так и в нулевое состояние); DI — входная информация, предназначенная для записи в ячейку ОЗУ, расположенную по адресу A1; DO — выходная информация, считанная из ячейки ОЗУ, расположенной по адресу A2.

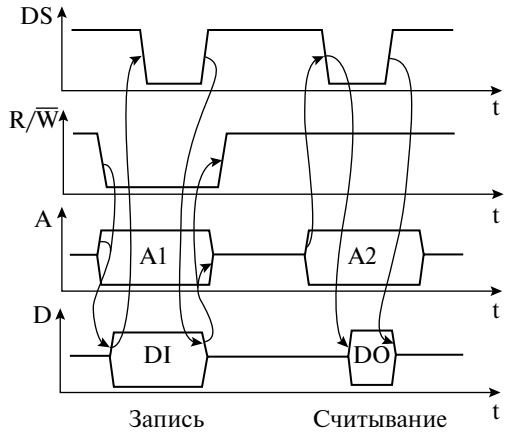
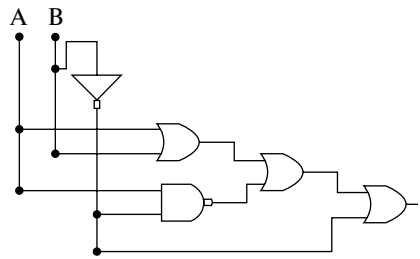


Рис. 2.74. Временная диаграмма обращения к ОЗУ, принятая для схем, совместимых со стандартом фирмы «Motorola»

Контрольные вопросы и задания

1. Перевести число DCCXIX из римской системы счета в арабскую.
2. Составить таблицу натурального ряда чисел от 0 до 25 в десятичной и семеричной системах счисления.
3. Составить алгоритм перевода целых и дробных чисел из десятичной системы счисления в шестнадцатеричную систему счисления.
4. Составить алгоритм перевода из двоичной в восьмеричную и шестнадцатеричную системы.
5. Перевести числа из десятичной системы счисления в восьмеричную систему счисления:
   $23{,}13_{10} - X_8$;
   $54{,}39_{10} - X_8$.
6. Перевести числа из двоичной системы счисления в шестнадцатеричную систему счисления:
   $11101101{,}1011_2 - X_{16}$;
   $1000100101{,}101001_2 - X_{16}$.
7. Перевести числа из восьмеричной системы счисления в шестнадцатеричную систему счисления:
   $25{,}5_8 - X_{10}$;
   $36{,}31_8 - X_{10}$.

8. Перевести числа из десятичной системы счисления в шестнадцатеричную систему счисления:
   $51{,}23_{10} - X_{16}$;
   $97{,}29_{10} - X_{16}$.
9. Получить прямой, обратный и дополнительный коды следующих чисел: $A_{10} = 31$; $B_{10} = -23$ — и выполнить сложение этих чисел.
10. Выполнить арифметические операции в восьмеричной системе счисления: $A_8 = 750$; $B_8 = 236$. Найти $A_8 + B_8, A_8 - B_8$.
11. Привести примеры простых и сложных высказываний.
12. Написать формулы СКНФ и СДНФ для переменных X_0, X_1, X_2.
13. Написать формулу зависимости количества различных переключательных функций от числа аргументов.
14. С помощью таблицы истинности проверить правило де Моргана.
15. Построить таблицу истинности и условное обозначение логических схем «НЕ», «И», «ИЛИ», «И-НЕ», «ИЛИ-НЕ».
16. Построить таблицу истинности, логическую схему и, используя законы алгебры, минимизировать следующую функции алгебры логики:
    $F = \overline{(A + \overline{B})} + \overline{A}B + B$.
17. Исследовать логическую схему и определить, при каких входных значениях A и B выходное значение логической схемы равно единице.



18. Исследовать работу асинхронного RS-триггера.
19. Исследовать работу полусумматора двоичных чисел.
20. Исследовать работу одноразрядного компаратора двоичных чисел.
21. Согласно приведенной ниже таблице истинности построить схему мультиплексора.

A_1	A_0	Y
0	0	D_0
0	1	D_1
1	0	D_2
1	1	D_3

22. Построить схему и исследовать принцип работы двоично-десятичного декодера.
23. Построить и исследовать структуру восьмиразрядного регистра хранения с асинхронным входом установки в «0».

24. Описать принцип работы АЛУ.
25. Описать принцип работы схемы масочного постоянного запоминающего устройства.
26. Исследовать схему ОЗУ, построенного на четырех микросхемах памяти, и описать процедуру обращения к постоянному запоминающему устройству на примере операции чтения из микросхемы.
27. Какие системы счисления вы знаете?
28. Чем отличаются непозиционная система счисления от позиционной системы счисления?
29. Изменяется ли длина разрядной сетки при записи одного и того же числа в разных системах счисления?
30. Чему равно максимальное по абсолютному значению число, которое можно записать при заданной длине разрядной сетки $N = 4$ для восьмеричной системы счисления?
31. Что такое количество информации?
32. Какие способы представления чисел применяются в вычислительной технике?
33. Чему равна максимальная относительная ошибка при представлении чисел с фиксированной запятой?
34. Чему равна минимальная относительная ошибка при представлении чисел с фиксированной запятой?
35. Можно ли выходить при выполнении операции за диапазон представляемых в данной разрядной сетке чисел, записанных с фиксированной запятой?
36. Для чего нужен дополнительный код? Как кодируются положительные и отрицательные числа в дополнительном коде?
37. Для чего нужен обратный код? Как кодируются положительные и отрицательные числа в обратном коде?
38. Что такое «положительный» и «отрицательный» нуль в обратном коде?
39. Применяется ли алгебра логики в современных ЭВМ?
40. Можно ли записать сложное и длинное высказывание простым высказыванием без нарушения истинности исходного высказывания?
41. Как зависит количество различных переключательных функций от числа аргументов?
42. Какие физические способы кодирования двоичной информации вы знаете?
43. Какие логические элементы вы знаете? Область применения.
44. Для чего нужен триггер? Где применяются триггеры?
45. Имеет ли принципиальное значение, выход какого из элементов считать прямым выходом триггера?
46. Что произойдет, если на входы R и S триггера подать одновременно две единицы?
47. В чем заключается отличие полного одноразрядного двоичного сумматора от полусумматора?

48. Перечислите основные функциональные цифровые узлы комбинационного типа и укажите их назначение.
49. Каково назначение арифметико-логического устройства?
50. Каково назначение статической и динамической памяти? В чем заключается их отличие?
51. Как зависит количество ячеек памяти ОЗУ от количества адресных выводов в микросхеме?
52. Какие микросхемы репрограммируемого ПЗУ российского и зарубежного производства вы знаете?

Глава 3
УРОВЕНЬ МИКРОАРХИТЕКТУРЫ

В этой главе рассмотрим уровень микроархитектуры, расположенный над цифровым логическим уровнем.

Прежде набор инструкций задавался жестко, каждая машинная инструкция (сложение, сдвиг, копирование) реализовывалась непосредственно в схеме. Это давало высокую скорость, но, по мере того как набор инструкций рос, все сложнее становилось реализовать в виде схемы и отладить инструкции все возрастающей сложности. Микрокод смягчил эту проблему тем, что позволил инженерам-проектировщикам при реализации сложной инструкции заменить создание сложной схемы на написание микропрограммы. Более того, микрокод можно было с легкостью изменить на поздних этапах проектирования, схему же изменить намного сложнее. Таким образом, микрокод облегчил проектирование процессоров, что привело к усложнению набора команд.

Использование микропрограмм также смягчило проблему пропускной способности памяти. В 1970-х гг. рост скорости процессора намного обгонял рост скорости памяти. Некоторые способы ускорения, такие как многоуровневые кэши, несколько смягчали проблему, но не решали ее. Использование микрокода здесь очень помогло, поскольку меньшее количество более сложных инструкций требовало меньшего обмена с памятью. Например, если вся операция над строкой символов выполняется одной машинной инструкцией, то во время ее выполнения не требуется выбирать из памяти другие инструкции.

3.1. Микропрограммное управление

Принципы микропрограммного управления

В условиях, когда микропрограммирование не используется, выполнение команды обеспечивается электрической схемой. Но в большинстве современных вычислительных машин непосредственная связь между аппаратурой и программными средствами осуществляется через микропрограммный уровень. Любая машинная команда исполняется аппаратурой не непосредственно, а путем их интерпретации в соответственную последовательность более

простых действий. А значит, всегда существует задача программирования машинных команд из более простых действий — микропрограммирование. Впервые этот термин был введен в 1953 г. специалистом по ВТ Уилксом. Но это было применимо только к аппаратным средствам. Примерно в середине 1960-х гг. усилиями разработчиков IBM идеи Уилкса превратились в принцип организации вычислительных машин.

Микропрограммирование обеспечило переход к модульному построению ЭВМ. Развивая идеи микропрограммирования, Глушков показал, что в любом устройстве обработки информации функционально можно выделить операционный автомат и управляющий автомат (рис. 3.1).

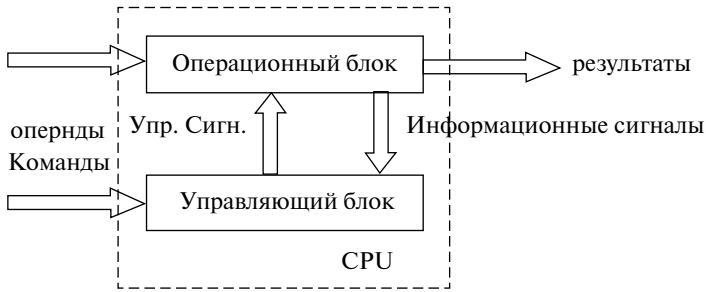


Рис. 3.1. Схема устройства обработки информации

Управляющий блок предназначен для обеспечения работы всех узлов и устройств ЭВМ в соответствии с выполняемой программой.

Основные функции управляющего блока:

- организация пуска и остановки ЭВМ;
- определение очередности выбора команд из оперативной памяти;
- формирование физических адресов операндов;
- формирование последовательности управляющих сигналов для выполнения арифметических, логических и иных операций при выполнении программы;
- обеспечение работы ЭВМ в различных режимах;
- автоматически выполняемая программа;
- пошаговое выполнение программы;
- режим прерывания;
- режим прямого доступа к памяти и т.д.

На рис. 3.2 приведена обобщенная структура устройства управления (УУ).

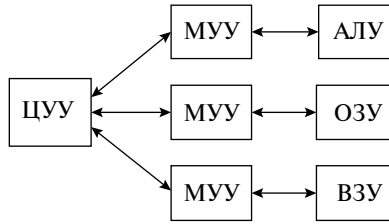


Рис. 3.2. Обобщенная структура УУ

ЦУУ — центральное УУ, которое выполняет основные функции по реализации программы.

МУУ — местное УУ (находится при каждом из устройств, входящих в состав ЭВМ). Оно реализует специфические алгоритмы, соответствующие принципам действия различных внешних устройств.

Рис. 3.3. Иерархическая структура понятий при постановке задач на ЭВМ

Алгоритм — это система последовательных операций (в соответствии с определенными правилами) для решения поставленной задачи.

Программа — кодированная запись алгоритма.

Команда — кодированная запись вычислительной, логической или иной операции. В устройствах ЭВМ команда физически выполняется с помощью микроопераций.

Микрооперация — некоторое простейшее преобразование данных, например прием байта данных в регистр, инверсия переменной и т.д. (рис. 3.3).

Порядок функционирования устройства базируется на следующих положениях.

1. Любая машинная команда рассматривается как некоторое сложное действие, которое состоит из последовательности элементарных действий над словами информации — микроопераций.

2. Порядок следования микроопераций зависит не только от значений преобразуемых слов, но также от их информационных сигналов, вырабатываемых операционным автоматом. Примерами таких сигналов могут быть признаки результата операции, значения отдельных битов данных и т.п.

3. Процесс выполнения машиной команды описывается в виде некоторого алгоритма в терминах микроопераций и логических условий. Описание информационных сигналов — микропрограмма.

4. Микропрограмма служит не только для обработки данных, но и обеспечивает управление работой всего устройства в целом — принцип микропрограммного управления.

Таким образом, основная задача уровня микроархитектуры — интерпретация команд второго уровня архитектуры команд (см. рис. 1.2). На этом уровне регистры вместе с АЛУ формируют **тракт данных**, по которому поступают данные. На некоторых машинах работа тракта данных контролируется особой программой, которая называется *микропрограммой*. На других машинах тракт данных контролируется аппаратными средствами. Строение уровня микроархитектуры зависит от уровня архитектуры команд, а также от стоимости и назначения компьютера.

Однако до недавнего времени этот принцип микропрограммного управления в вычислительных машинах широкого применения не нашел. Объясняется это несколькими причинами.

С одной стороны, не существовали достаточно надежные и дешевые быстродействующие запоминающие устройства для хранения микропрограмм; с другой стороны, неправильно понимались задачи микропрограммирования и те выгоды, которые оно может принести. Предполагалось, что главная ценность микропрограммирования состоит в том, что каждый потребитель может сконструировать себе из микропрограмм тот набор команд, который наиболее выгоден для его конкретной задачи. Переход от одного набора команд к другому достигался бы путем простой замены информации в запоминающем устройстве без физического переконструирования устройства.

Чтобы освободить программиста от необходимости детально изучать устройство машины, необходимо использовать методы автоматического программирования и в максимальной степени приблизить

язык программирования к языку человека. И уже с 1970 г., когда микропрограммирование стало обычным делом, у производителей появилась возможность вводить новые машинные команды путем расширения микропрограммы, т.е. с помощью программирования. Это открытие привело к виртуальному взрыву в производстве программ машинных команд, поскольку производители начали конкурировать друг с другом, стараясь выпустить лучшие программы. Эти команды не представляли особой ценности, поскольку те же задачи можно было легко решить, используя уже существующие программы, но обычно они работали немного быстрее.

Связь с микрокодом и архитектурой набора команд

В компьютерной инженерии микроархитектура (англ. microarchitecture), также называемая организацией компьютера, — это способ, которым данная архитектура набора команд (АНК) реализована в процессоре. Каждая АНК может быть реализована с помощью различных микроархитектур. Реализации могут варьироваться в зависимости от целей конкретной разработки или в результате технологических сдвигов. Архитектура компьютера является комбинацией микроархитектуры, микрокода и архитектуры набора команд.

Архитектура набора команд (англ. instruction set architecture, ISA) определяет программируемую часть ядра микропроцессора.

Более подробно АНК будет рассмотрена в гл. 5.

Микроархитектура описывает модель, топологию и реализацию АНК на микросхеме микропроцессора. На этом уровне определяется:

- конструкция и взаимосвязь основных блоков центрального процессора;
- структура ядер, исполнительных устройств, АЛУ, а также их взаимодействия;
- блоки предсказания переходов;
- организация конвейеров;
- организация кэш-памяти;
- взаимодействие с внешними устройствами.

В рамках одного семейства микропроцессоров микроархитектура со временем расширяется путем добавления новых усовершенствований и оптимизации существующих команд с целью повышения производительности, энергосбережения и функциональных возможностей микропроцессора. При этом сохраняется совместимость с предыдущей версией ISA. Во многих случаях работа элементов микроархитектуры контролируется микрокодом, встроенным в процес-

сор. В случае наличия слоя микрокода в архитектуре процессора он выступает своеобразным интерпретатором, преобразуя команды уровня АНК в команды уровня микроархитектуры. При этом различные системы команд могут быть реализованы на базе одной микроархитектуры.

Микроархитектура машины обычно представляется в виде диаграмм определенной степени детализации, описывающих взаимосвязи различных микроархитектурных элементов, которые могут быть чем угодно: от отдельных вентилей и регистров до целых арифметико-логических устройств и даже более крупных элементов. На этих диаграммах обычно выделяют тракт данных (где размещены данные) и тракт управления (который управляет движением данных).

Машины с различной микроархитектурой могут иметь одинаковую АНК и, таким образом, быть пригодными для выполнения тех же программ. Новые микроархитектуры и/или схемотехнические решения вместе с прогрессом в полупроводниковой промышленности являются тем, что позволяет новым поколениям процессоров достигать более высокой производительности, используя ту же АНК.

В настоящее время на уровне микроархитектуры команд обычно находятся простые команды, которые выполняются за один цикл (таковы, в частности, RISC-машины). В других системах (например, в Pentium 4) на этом уровне имеются более сложные команды; выполнение одной такой команды занимает несколько циклов. Чтобы выполнить команду, нужно найти операнды в памяти, считать их и записать полученные результаты обратно в память. Управление уровнем команд со сложными командами отличается от управления уровнем команд с простыми командами, так как в первом случае выполнение одной команды требует определенной последовательности операций.

3.2. Принципы реализации микропроцессоров

В общем случае все центральные процессоры, одночиповые микропроцессоры и многочиповые реализации выполняют программы, производя следующие шаги:

1) чтение инструкции и ее декодирование;

2) поиск всех связанных данных, необходимых для обработки инструкции;

3) обработка инструкции;

4) запись результатов.

Эта последовательность выглядит просто, но осложняется тем фактом, что иерархия памяти (где располагаются инструкции и дан-

ные), которая включает в себя кэш, основную память и энергонезависимые устройства хранения, такие как жесткие диски, всегда была медленнее самого процессора. Шаг 2 часто привносит длительные (по меркам центрального процессора) задержки, пока данные поступают по компьютерной шине.

В настоящее время существуют два направления развития микропроцессоров:

- RISC-процессоры (процессоры с сокращенным набором команд);
- CISC-процессоры (процессоры с полным набором команд).

В процессорах с полным набором команд (CISC-процессорах) используется уровень микропрограммирования, для того чтобы декодировать и выполнить команду микропроцессора. В этих процессорах формат команды не зависит от аппаратуры процессора. На одной и той же аппаратуре при смене микропрограммы могут быть реализованы различные микропроцессоры. С другой стороны, смена аппаратуры никак не влияет на программное обеспечение микропроцессора. С точки зрения пользователя у микропроцессора только увеличивается производительность, снижается потребление энергии, уменьшаются габариты устройств. Неявным недостатком таких процессоров является то, что производители микросхем стараются увеличить количество команд, которые может выполнять микропроцессор, тем самым увеличивая сложность микропрограммы и замедляя выполнение каждой команды в целом.

В процессорах с сокращенным набором команд (RISC-процессорах) декодирование и исполнение команды производятся аппаратно, поэтому количество команд ограничено минимальным набором. В этих процессорах команда и микрокоманда совпадают. Преимуществом этого типа процессоров является то, что команда может быть в принципе выполнена за один такт (не требуется выполнение микропрограммы), однако для выполнения тех же действий, которые выполняет команда CISC-процессора, требуется выполнение целой программы.

Так как RISC-процессоры выполняют одну команду за один такт, то производители провозглашают однозначное превосходство RISC-процессоров над CISC-процессорами, однако при выборе процессора нужно учитывать все параметры в целом.

Обычно тактовая частота RISC-процессора значительно ниже по сравнению с CISC-процессором, поэтому общая производительность микропроцессорной системы, построенной на RISC-процессоре, может оказаться той же или ниже по сравнению с микропроцессорной системой, построенной на CISC-микропроцессоре.

Разрядность команды RISC-процессора может оказаться выше, чем у CISC-процессора (что чаще всего и бывает). В результате общий объем исполняемой программы для RISC-процессора превысит объем подобной программы для CISC-процессора, что ведет к повышенным требованиям к объему ПЗУ.

В качестве примера внутреннего устройства микропроцессора рассмотрим устройство процессора с полным набором команд. Здесь будет рассматриваться упрощенная модель процессора для облегчения понимания работы. CISC-микропроцессор состоит из двух частей:

- обрабатывающего блока;
- блока микропрограммного управления.

Блок обработки микропроцессора (операционный блок). Блок обработки сигналов предназначен для считывания команд из системной памяти и выполнения считанных команд. Эти действия он осуществляет под управлением блока микропрограммного управления, который формирует последовательность микрокоманд, необходимую для выполнения команды (рис. 3.4).

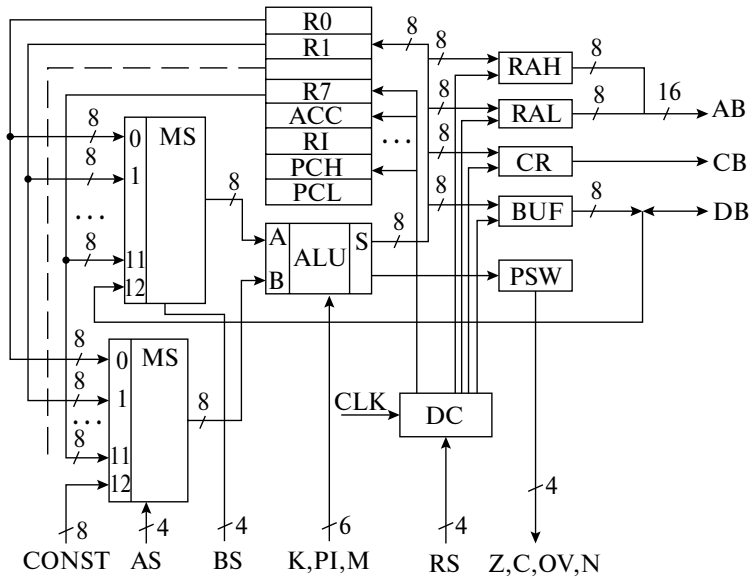


Рис. 3.4. Операционный блок микропроцессора

В этой схеме явно просматривается, что отдельные биты микрокоманды (показанной внизу схемы) управляют различными блоками обработки сигналов (БОС), поэтому их можно рассматривать неза-

висимо друг от друга. Такие группы битов называются полями микрокоманды и составляют формат этой микрокоманды. Кроме битов, управляющих блоком обработки сигналов, есть биты, управляющие блоком микропрограммного управления. Результат выполнения микрокоманды записывается по сигналу общей синхронизации CLK.

Основным принципом работы любого цифрового устройства с памятью, в том числе и микропроцессора, является наличие цепи синхронизации. Этот сигнал, как и цепь питания, подводится к любому регистру цифрового устройства.

Для хранения и декодирования выполняемой команды выделим 8-разрядный регистр, который назовем RI.

Для реализации более простой системы команд выберем аккумуляторный процессор. Соответственно, необходимо один из регистров выделить в качестве аккумулятора ACC.

Так как мы выбрали для примера 8-разрядный микропроцессор, то и все регистры в этом процессоре 8-разрядные. Максимальное число, которое можно записать в такой регистр, — 255, но для большинства программ такого объема памяти недостаточно. В приведенной на рис. 3.4 схеме, для того чтобы получить 16-разрядный адрес, используется два 8-разрядных регистра адреса. Теперь максимальное число, которое можно записать в этих двух регистрах, будет 65535, что вполне достаточно для записи программ и обрабатываемых ими данных. Для того чтобы различать регистр старшего и младшего байта адреса, обозначим их как PCH — старший байт и PCL — младший байт. Это позволяет при помощи 8-разрядного АЛУ формировать 16-разрядный адрес.

Программный счетчик хранит текущее значение ячейки памяти, из которой считывается команда, но процессор может обращаться и к данным, поэтому для формирования адреса выделим еще пару регистров: RAH — старший байт и RAL — младший байт. Выходы этих регистров выведем за пределы микросхемы и будем использовать в качестве 16-разрядной шины адреса.

Еще один регистр используется для формирования сигналов управления системной шины микропроцессора. В простейшем случае это сигналы записи (WR) и чтения (RD). Для формирования необходимых сигналов достаточно записывать в определенный бит регистра логический ноль или логическую единицу. Определим формат регистра управления. Пусть нулевой бит этого регистра будет сигналом записи, а первый бит этого регистра будет сигналом чтения. Остальные биты этого регистра пока не важны. Полученный формат приведен на рис. 3.5.

7	6	5	4	3	2	1	0
						RD	WR

Рис. 3.5. Формат регистра управления (CR)

Блок микропрограммного управления микропроцессора. Блок микропрограммного управления предназначен для формирования последовательности микрокоманд блока обработки сигналов. В простейшем случае его можно построить на счетчике с возможностью предзаписи и ПЗУ. Схема такого блока приведена на рис. 3.6.

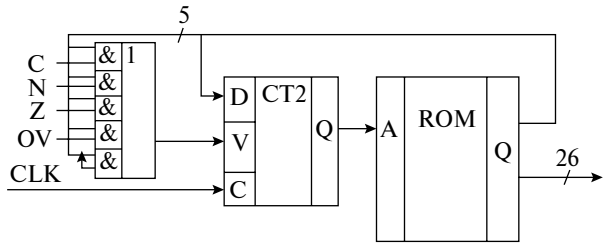


Рис. 3.6. Блок микропрограммного управления микропроцессора

В этой схеме адрес очередной микрокоманды формирует двоичный счетчик. Если требуется осуществить безусловный или условный переход, то новый адрес записывается из ПЗУ в этот счетчик как в обычный параллельный регистр по сигналу параллельной записи V. Переход к следующему адресу микрокоманды производится по сигналу общей синхронизации CLK (рис. 3.7).

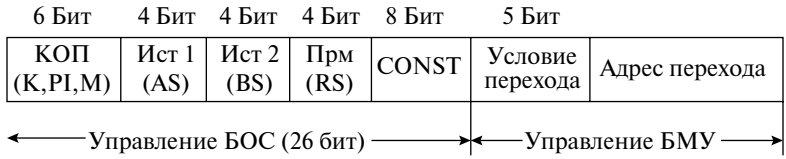


Рис. 3.7. Формат микрокоманды процессора

3.3. Микропрограммирование процессора

Все действия микропроцессора и сигналы на его выводах определяются последовательностью микрокоманд, подаваемых на управляющие входы блока обработки. Эта последовательность микрокоманд называется микропрограммой.

При изучении принципов работы ОЗУ и ПЗУ приводились временные диаграммы, которые необходимо сформировать, для того чтобы записать или прочитать необходимую информацию. Выберем одну из этих диаграмм (рис. 3.8).

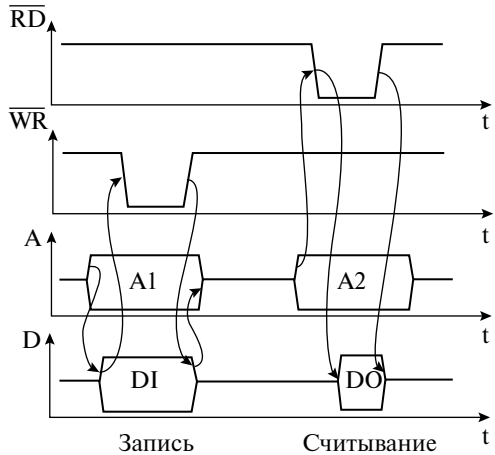


Рис. 3.8. Временные диаграммы считывания команды микропроцессором

Любую временную диаграмму формирует микропроцессор. Устройство микросхемы, на примере которой мы будем формировать необходимые для работы сигналы, рассматривалось при обсуждении блока обработки микропроцессора. По принципиальной схеме блока обработки сигнала можно определить формат микрокоманды, управляющей этим блоком.

Работа любого цифрового устройства начинается с заранее заданных начальных условий. Эти начальные условия формируются специальным сигналом RESET (сброс), который формируется после подачи питания на схему. Договоримся, что сигнал сброса микропроцессора будет записывать в регистр программного счетчика PC нулевое значение. (Это условие справедливо не для всех процессоров. Например, IBM-совместимые процессоры при сбросе микросхемы записывают в программный счетчик значение F0000h.)

Выполнение любой команды начинается с ее считывания из системной памяти (ОЗУ или ПЗУ). Необходимые для этого микрокоманды подаются на входы управления БОС из блока микропрограммного управления (БМУ), как только снимается сигнал сброса со счетчика микрокоманд БМУ. При считывании однобайтной команды достаточно считать из системной памяти только код операции и вы-

полнить эту операцию. Временная диаграмма этого процесса приведена на рис. 3.9. Последовательность операций, которые необходимо выполнить микропрограмме, показана стрелочками. Для считывания следующей команды микропрограмма запускается заново.

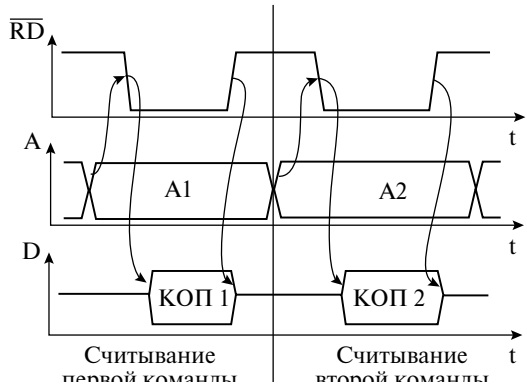


Рис. 3.9. Временные диаграммы сигналов считывания команд из ОЗУ

Для того чтобы считать код операции из системной памяти, необходимо выставить на шине адреса адрес этой команды. Этот адрес хранится в счетчике команд PC. Скопируем его в регистр адреса RA, выходы которого подключены к шине адреса.

		Поля микрокоманды БОС				
N	Описание	Константа	Источник *A*	Источник *B*	Команда	Приемник
1	PCH -> RAH	1111 1111	1111	1010	001100	1100
2	PCL -> RAL	1111 1111	1111	1011	001100	1101

Затем сформируем сигнал считывания. Для этого в регистр управления запишем константу 1111 1101.

		Поля микрокоманды БОС				
N	Описание	Константа	Источник *A*	Источник *B*	Команда	Приемник
3	const -> CR	1111 1101	1111	1111	001100	1110

Теперь можно считать число с шины данных, а так как системная память в этот момент выдает на нее код операции, то мы считаем именно этот код. Запишем его в регистр команд и снимем сигнал чтения с системной шины. Для этого в регистр управления запишем константу 1111 1111.

		Поля микрокоманды БОС				
N	Описание	Константа	Источник *A*	Источник *B*	Команда	Приемник
4	data -> RI	11111111	1100	1111	110000	1001
5	const -> CR	11111111	1111	1111	001100	1110

Прежде чем перейти к дальнейшему выполнению микропрограммы, увеличим содержимое счетчика команд на единицу.

		Поля микрокоманды БОС				
N	Описание	Константа	Источник A	Источник B	Команда	Приемник
6	PCL+1 -> PCL	11111111	1011	1111	110010	1011
7	PCH+C -> PCH	11111111	1010	1111	110010	1010

После считывания команды ее необходимо декодировать. Это можно выполнить микропрограммным способом, проверяя каждый бит регистра команд и осуществляя ветвление по результату проверки, или включить в состав блока микропрограммного управления аппаратный дешифратор команд, который сможет осуществить ветвление микропрограммы на 256 ветвей за один такт синхронизации микропроцессора. Выберем именно этот путь. Восьмым тактом микропрограмма направляется на одну из 256 ветвей, отвечающую за выполнение считанной инструкции. Например, если была считана команда MOV A, R0, то следующая микрокоманда будет выглядеть следующим образом:

		Поля микрокоманды БОС				
N	Описание	Константа	Источник *A*	Источник *B*	Команда	Приемник
8	R0 -> ACC	11111111	0000	1111	110000	1000

И так как в этом случае команда полностью выполнена, то счетчик микрокоманд сбрасывается для выполнения следующей команды.

Рассмотрим еще один **пример**. Пусть из системной памяти считывается команда безусловного перехода JMP 1234. Первые восемь микрокоманд совпадают для всех команд микропроцессора. Различие наступает начиная с девятой команды, которая зависит от конкретной инструкции. При выполнении команды безусловного перехода необходимо считать адрес новой команды, который записан в байтах, следующих за кодом операции. Этот процесс аналогичен считыванию кода операции.

		Поля микрокоманды БОС				
N	Описание	Константа	Источник *A*	Источник *B*	Команда	Приемник
9	PCH -> RAH	11111111	1111	1010	001100	1100
10	PCL -> RAL	11111111	1111	1011	001100	1101
11	const -> CR	11111110	1111	1111	001100	1110
12	data -> RI	11111111	1100	1111	110000	1001
13	const -> CR	11111111	1111	1111	001100	1110
14	PCL+1 -> PCL	11111111	1011	1111	110010	1011
15	PCH+C -> PCH	11111111	1010	1111	110010	1010

Теперь считаем второй байт адреса перехода.

		Поля микрокоманды БОС				
N	Описание	Константа	Источник A	Источник *B*	Команда	Приемник
16	PCH -> RAH	11111111	1111	1010	001100	1100
17	PCL -> RAL	11111111	1111	1011	001100	1101
18	const -> CR	11111110	1111	1111	001100	1110
19	data -> PCH	11111111	1100	1111	110000	1110
20	const -> CR	11111111	1111	1111	001100	1110
21	RI -> PCL	11111111	1001	1111	110010	1001

В результате выполнения этой микропрограммы в программный счетчик будет загружен адрес, записанный во втором и третьем байтах команды безусловного перехода JMP 1234. Временная диаграмма, формируемая рассмотренной микропрограммой, приведена на рис. 3.10.

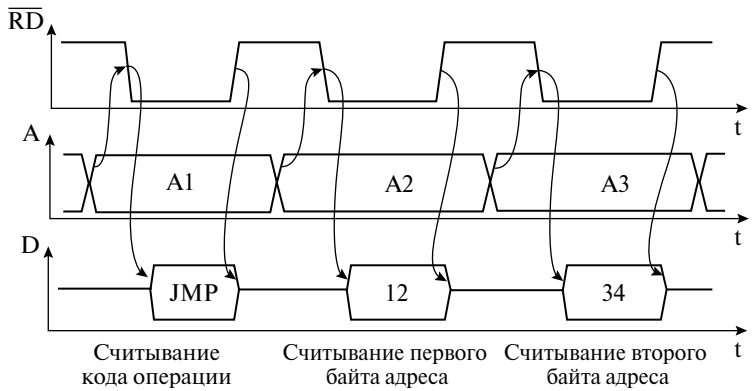


Рис. 3.10. Временная диаграмма выполнения команды JMP 1234

3.4. Конвейеризация инструкций

Одна из первых и наиболее мощных техник повышения производительности — использование конвейера инструкций.Конвейеры увеличивают производительность, позволяя нескольким инструкциям прокладывать свой путь через процессор в одно и то же время.

Многие современные процессоры управляются тактовым генератором. Процессор внутри состоит из логических элементов и ячеек памяти — триггеров. Когда приходит сигнал от тактового генератора, триггеры приобретают свое новое значение и «логике» требуется некоторое время для декодирования новых значений. Затем приходит следующий сигнал от тактового генератора, триггеры принимают новые значения и т.д. Разбивая последовательности логических элементов на более короткие и помещая триггеры между этими короткими последовательностями, уменьшают время, необходимое логике для обработки сигналов. В этом случае длительность одного такта процессора может быть соответственно уменьшена.

Например, простейший конвейер RISC-процессоров можно представить пятью стадиями с наборами триггеров между стадиями:

- получение инструкции;
- декодирование инструкции;
- выполнение;
- доступ к памяти, запись в регистр.

RISC сделал конвейеры меньше и значительно проще в конструировании, отделив каждый этап обработки инструкций, зафиксировав длину машинной инструкции и сделав время их выполнения одинаковым — один такт или как максимум один цикл доступа к памяти (из-за выделения инструкций load и store). Процессор в целом функционирует на манер сборочной линии с инструкциями, поступающими с одной стороны, и результатами, выходящими с другой. Из-за уменьшенной сложности классического RISC-конвейера конвейерезированное ядро и кэш инструкций могли быть размещены на кристалле того же размера, который содержал бы лишь ядро в случае CISC-архитектуры. Это и было истинной причиной того, что RISC был быстрее. Ранние разработки, такие как SPARC и MIPS, часто работали в 10 раз быстрее CISC-решений Intel и Motorola той же тактовой частоты и цены.

Преимущества и недостатки. Конвейер помогает не во всех случаях. Существует несколько возможных минусов. Конвейер инструкций можно назвать «полностью конвейерным», если он может принимать новую инструкцию каждый машинный цикл. Иначе в конвейер должны быть вынужденно вставлены задержки, которые выравнивают конвейер, при этом ухудшая его производительность.

Преимущества. Время цикла процессора уменьшается, таким образом увеличивая скорость обработки инструкций в большинстве случаев.

Некоторые комбинационные логические элементы, такие как сумматоры или умножители, могут быть ускорены путем увеличения количества логических элементов. Использование конвейера может предотвратить ненужное наращивание количества элементов.

Недостатки. Бесконвейерный процессор исполняет только одну инструкцию за раз. Это предотвращает задержки веток инструкций (фактически каждая ветка задерживается), и проблемы, связанные с последовательными инструкциями, которые исполняются параллельно. Следовательно, схема такого процессора проще и он дешевле для изготовления.

Задержка инструкций в бесконвейерном процессоре слегка ниже, чем в конвейерном эквиваленте. Это происходит из-за того, что в конвейерный процессор должны быть добавлены дополнительные триггеры.

У бесконвейерного процессора скорость обработки инструкций стабильна. Производительность конвейерного процессора предсказать намного сложнее, и она может значительно различаться в разных программах.

Рассмотрим пример. Допустим, типичная инструкция для сложения двух чисел — это СЛОЖИТЬ *A*, *B*, *C*. Эта инструкция суммирует значения, находящиеся в ячейках памяти *A* и *B*, а затем кладет результат в ячейку памяти *C*. В конвейерном процессоре контроллер может разбить эту операцию на последовательные задачи вида

- ЗАГРУЗИТЬ A, R1;
- ЗАГРУЗИТЬ B, R2;
- СЛОЖИТЬ R1, R2, R3;
- ЗАПИСАТЬ R3, C;
- загрузить следующую инструкцию.

Ячейки *R*1, *R*2 и *R*3 являются регистрами процессора. Значения, которые хранятся в ячейках памяти, которые мы называем *A* и *B*, загружаются (т.е. копируются) в эти регистры, затем суммируются, и результат записывается в ячейку памяти *C*.

В данном примере конвейер состоит из трех уровней — загрузки, исполнения и записи. Эти шаги называются, очевидно, уровнями или шагами конвейера.

В бесконвейерном процессоре только один шаг может работать в один момент времени, поэтому инструкция должна полностью закончиться, прежде чем следующая инструкция в принципе начнется. В конвейерном процессоре все эти шаги могут выполняться одновременно на разных инструкциях. Поэтому, когда первая инструкция

находится на шаге исполнения, вторая инструкция будет на стадии раскодирования, а третья инструкция будет на стадии прочтения.

Конвейер не уменьшает время, которое необходимо для того, чтобы выполнить инструкцию, но зато он увеличивает объем (число) инструкций, которые могут быть выполнены одновременно, и таким образом уменьшает задержку между выполненными инструкциями, увеличивая так называемую пропускную способность. Чем больше уровней имеет конвейер, тем больше инструкций могут выполняться одновременно и тем меньше задержка между завершенными инструкциями. Каждый микропроцессор, произведенный в наши дни, использует как минимум двухуровневый конвейер.

Рассмотрим еще один пример. Теоретический трехуровневый конвейер:

Шаг	Англ. название	Описание
Выборка	Fetch	Прочитать инструкцию из памяти
Исполнение	Execute	Исполнить инструкцию
Запись	Write-back	Записать результат в память и/или регистры

Псевдоассемблерный листинг, который нужно выполнить:

- ЗАГРУЗИТЬ 40, A — загрузить число 40 в A;
- КОПИРОВАТЬ A, B — скопировать A в B;
- СЛОЖИТЬ 20, B — добавить 20 к B;
- ЗАПИСАТЬ B, 0x0300 — записать B в ячейку памяти 0x0300.

Как это будет исполняться:

Такт	Выборка	Исполнение	Запись	Пояснение
Такт 1	ЗАГРУЗИТЬ			Инструкция ЗАГРУЗИТЬ читается из памяти
Такт 2	КОПИРОВАТЬ	ЗАГРУЗИТЬ		Инструкция ЗАГРУЗИТЬ выполняется, инструкция КОПИРОВАТЬ читается из памяти
Такт 3	СЛОЖИТЬ	КОПИРОВАТЬ	ЗАГРУЗИТЬ	Инструкция ЗАГРУЗИТЬ находится на шаге записи результата, где ее результат (т.е. число *40*) записывается в регистр *A*. В это же время инструкция КОПИРОВАТЬ исполняется. Так как она должна скопировать содержимое регистра *A* в регистр *B*, она должна дождаться окончания инструкции ЗАГРУЗИТЬ

Такт	Выборка	Исполнение	Запись	Пояснение
Такт 4	ЗАПИСАТЬ	СЛОЖИТЬ	СКОПИРОВАТЬ	Загружена инструкция ЗАПИСАТЬ, тогда как инструкция СКОПИРОВАТЬ прощается с нами, а по инструкции СЛОЖИТЬ в данный момент производятся вычисления

И так далее. Следует учитывать, что иногда инструкции будут зависеть от итогов других инструкций (например, как наша инструкция КОПИРОВАТЬ). Когда более чем одна инструкция ссылается на определенное место, читая его (т.е. используя в качестве входного операнда) либо записывая в него (т.е. используя его в качестве выходного операнда), исполнение инструкций не в порядке, который был изначально запланирован в оригинальной программе, может повлечь за собой «конфликт конвейера». Существует несколько зарекомендовавших себя приемов либо для предотвращения конфликтов, либо для их исправления, если они случились.

3.5. Кэш процессора

Когда улучшения в производстве чипов позволили размещать на кристалле еще больше логики, начался поиск способов применения этого ресурса. Одним из направлений стало размещение прямо на кристалле чипа очень быстрой кэш-памяти, доступ к которой происходил всего за несколько тактов процессора, в отличие от большого их количества при работе с основной памятью. При этом процессор также включал контроллер кэша, автоматизировавший чтение и запись данных в кэш.

RISC-процессоры стали снабжаться кэшем в середине — конце 1980-х гг. (часто объемом всего 4 КБ). Этот объем постоянно возрастал, и современные процессоры имеют по крайней мере 512 КБ, а наиболее мощные — 1, 2, 4, 6, 8 и даже 12 МБ кэш-памяти, организованной в иерархию. В целом, больший объем кэша означает большую производительность вследствие меньшего времени простоя процессора.

Кэш-память и конвейеры хорошо дополняют друг друга. Если первоначально не имело смысла создавать конвейеры, работающие быстрее времени доступа к основной памяти, то с появлением кэша конвейер стал ограничиваться лишь более короткими задержками доступа к быстрой памяти на чипе. В итоге это позволяло увеличивать тактовые частоты процессоров.

Уровни кэш-памяти

Кэш центрального процессора разделен на несколько уровней. Максимальное количество кэшей — четыре. В универсальном процессоре в настоящее время число уровней может достигать трех. Кэш-память уровня N+1, как правило, больше по размеру и медленнее по скорости доступа и передаче данных, чем кэш-память уровня N.

Самым быстрым является кэш первого уровня — L1 cache (level 1 cache). По сути, она является неотъемлемой частью процессора, поскольку расположена на одном с ним кристалле и входит в состав функциональных блоков. В современных процессорах обычно L1 разделен на два кэша — кэш команд (инструкций) и кэш данных (гарвардская архитектура). Большинство процессоров без L1 не могут функционировать. L1 работает на частоте процессора, и в общем случае обращение к нему может производиться каждый такт. Зачастую является возможным выполнять несколько операций чтения/записи одновременно.

Вторым по быстродействию является кэш второго уровня — L2 cache, который обычно, как и L1, расположен на одном кристалле с процессором. В ранних версиях процессоров L2 реализован в виде отдельного набора микросхем памяти на материнской плате. Объем L2 от 128 Кбайт до 1—12 Мбайт. В современных многоядерных процессорах кэш второго уровня, находясь на том же кристалле, является памятью раздельного пользования — при общем объеме кэша в n Мбайт на каждое ядро приходится по n/c Мбайта, где c — количество ядер процессора.

Кэш третьего уровня наименее быстродействующий, но он может быть очень большим — более 24 Мбайт. L3 медленнее предыдущих кэшей, но все равно значительно быстрее, чем оперативная память. В многопроцессорных системах находится в общем пользовании и предназначен для синхронизации данных различных L2.

Существует четвертый уровень кэша, применение которого оправдано только для многопроцессорных высокопроизводительных серверов и мейнфреймов. Обычно он реализован отдельной микросхемой.

Принцип работы кэша

Рассмотрим типичный кэш данных и некоторые виды кэшей инструкций. Кэш состоит из собственно кэш-памяти, и кэш-контроллера. Кэш-контроллер управляет кэш-памятью: загружает в нее нужные данные из оперативной памяти и возвращает, когда нужно, модифицированные процессором данные в оперативную память.

При доступе процессора в память сначала производится проверка, хранит ли кэш запрашиваемые из памяти данные. Для этого производится сравнение адреса запроса со значениями всех тегов кэша, в которых эти данные могут храниться. Случай совпадения с тегом какой-либо кэш-линии называется попаданием в кэш (англ. cache hit), обратный же случай называется кэш-промахом (англ. cache miss). Попадание в кэш позволяет процессору немедленно произвести чтение или запись данных в кэш-линии с совпавшим тегом. Отношение количества попаданий в кэш к общему количеству запросов к памяти называют рейтингом попаданий (англ. hit rate), он является мерой эффективности кэша для выбранного алгоритма или программы.

В случае промаха в кэше выделяется новая запись, в тег которой записывается адрес текущего запроса, а в саму кэш-линию — данные из памяти после их прочтения либо данные для записи в память. Промахи по чтению задерживают исполнение, поскольку они требуют запроса данных в более медленной основной памяти. Промахи по записи могут не давать задержку, поскольку записываемые данные сразу могут быть сохранены в кэше, а запись их в основную память можно произвести в фоновом режиме. Работа кэшей инструкций во многом похожа на вышеприведенный алгоритм работы кэша данных, но для инструкций выполняются только запросы на чтение. Кэши инструкций и данных могут быть разделены для увеличения производительности (принцип, используемый в гарвардской архитектуре) или объединены для упрощения аппаратной реализации.

Для добавления данных в кэш после кэш-промаха может потребоваться вытеснение (англ. evict) ранее записанных данных. Для выбора замещаемой строки кэша используется эвристика, называемая политикой замещения (англ. replacement policy). Основной проблемой алгоритма является предсказание, какая строка, вероятнее всего, не потребуется для последующих операций. Качественные предсказания сложны, и аппаратные кэши используют простые правила, такие как LRU. Пометка некоторых областей памяти как не кэшируемых (англ. non cacheable) улучшает производительность за счет запрета кэширования редко используемых данных. Промахи для такой памяти не создают копию данных в кэше.

При записи данных в кэш должен существовать определенный момент времени, когда они будут записаны в основную память. Это время контролируется политикой записи (англ. write policy). Для кэшей со сквозной записью (англ. write-through) любая запись в кэш приводит к немедленной записи в память. Другой тип кэшей, обрат-

ная запись (англ. write-back), иногда также называемый copy-back, откладывает запись на более позднее время. В таких кэшах отслеживается состояние кэш-линеек, еще не сброшенных в память (пометка битом «грязный», англ. dirty). Запись в память производится при вытеснении подобной строки из кэша. Таким образом, промах в кэше, использующем политику обратной записи, может потребовать двух операций доступа в память: одну для сброса состояния старой строки и другую — для чтения новых данных.

Существуют также смешанные политики. Кэш может быть со сквозной записью (англ. write-through), но для уменьшения количества транзакций на шине записи могут временно помещаться в очередь и объединяться друг с другом.

Данные в основной памяти могут изменяться не только процессором, но и периферией, использующей прямой доступ к памяти, или другими процессорами в многопроцессорной системе. Изменение данных приводит к устареванию их копии в кэше (состояние stale). В другой реализации, когда один процессор изменяет данные в кэше, копии этих данных в кэшах других процессоров будут помечены как stale. Для поддержания содержимого нескольких кэшей в актуальном состоянии используется специальный протокол кэш-когерентности.

Контрольные вопросы и задания

1. Опишите обобщенную структуру процессора.
2. Как принцип академика Глушкова реализуется в структуре процессора?
3. Почему устройства обработки цифровой информации имеют многоуровневую структуру?
4. Какие операции выполняются в АЛУ? Как в зависимости от реализации этих операций подразделяются АЛУ?
5. Чем отличаются АЛУ блочного типа от многофункциональных АЛУ?
6. Опишите структуру АЛУ простейшего микропроцессора.
7. Опишите общие принципы построения УУ.
8. Укажите основные отличия УУ на жесткой логике от УУ с хранимой в памяти логикой.
9. Перечислите преимущества УУ с жесткой логикой.
10. В чем заключается главный недостаток УУ на жесткой логике?
11. Какое решение было найдено для устранения главного недостатка УУ на жесткой логике?
12. Опишите структуру УУ с хранимой в памяти логикой.
13. Перечислите варианты взаимного расположения циклов выборка — реализация МК.
14. Охарактеризуйте основные способы формирования адреса следующей МК.

15. Какие бывают форматы микрокоманд?
16. Опишите алгоритмы формирования адреса следующей МК.
17. Назовите способы кодирования МК. Приведите для каждого способа схему кодирования МК.
18. Опишите достоинства и недостатки каждого способа кодирования микрокоманды.
19. Как подразделяются МК с точки зрения синхронизации?
20. Какие характеристики вычислительной машины охватывает понятие «архитектура системы команд»?
21. Какие особенности аккумуляторной архитектуры можно считать ее достоинствами и недостатками?
22. Какие доводы можно привести за и против увеличения числа регистров общего назначения в ВМ с регистровой архитектурой системы команд?
23. Какие факторы определяют выбор формата команд?
24. Какая особенность фон-неймановской архитектуры позволяет отказаться от указания в команде адреса очередной команды?
25. С какими ограничениями связано использование непосредственной адресации?
26. В каких случаях может быть удобна многоуровневая косвенная адресация?
27. В чем проявляются сходства и различия между базовой и индексной адресацией?
28. Для чего нужна кэш-память?
29. Перечислите уровни кэш центрального процессора. Какой уровень является самым быстродействующим?
30. Опишите принцип работы кэша.

Глава 4
УРОВЕНЬ АРХИТЕКТУРЫ НАБОРА КОМАНД

В этой главе рассмотрим уровень архитектуры набора команд (ISA). Как показано на рис. 1.2, он расположен между уровнями микроархитектуры и операционной системы. Исторически этот уровень развился прежде всех остальных уровней и изначально был единственным [1].

Уровень АНК определят программируемую часть ядра микропроцессора и является связующим звеном между программным и аппаратным обеспечением вычислительной машины.

Цель трансляции — преобразование текста с одного языка на язык, понятный адресату. При трансляции компьютерной программы адресатом может быть:

- устройство-процессор (трансляция называется компиляцией);
- программа-интерпретатор (трансляция называется интерпретацией).

Виды трансляции:

- компиляция;
- интерпретация;
- динамическая компиляция.

При разработке программ на различных языках высокого уровня их требуется транслировать в некую общую для всех промежуточную форму. Цель трансляции — преобразование текста с одного языка на язык, понятный адресату. При трансляции программы на языке высокого уровня адресатом может быть устройство-процессор (трансляция называется компиляцией) или программа-интерпретатор (трансляция называется интерпретацией).

Компилятор — это транслятор, который преобразует исходный код с какого-либо языка программирования в машинный язык процессора (машинный код). Код на машинном языке исполняется процессором.

Уровень АНК связывает компиляторы и аппаратное обеспечение. Это язык, который понятен и компиляторам, и устройствам.

При проектировании вычислительной машины разработчики должны не только стремиться к повышению производительности системы, но и делать уровень команд совместимым с предыдущими моделями, так как основная цель покупателя — убедиться, что старые программы работают на новой машине. Разработчики могут переходить от микропрограмм к непосредственному использованию

устройств, добавлять конвейеры, реализовывать суперскалярные схемы и т.п., но при условии, что они сохранят обратную совместимость с уровнем команд предыдущих моделей. Таким образом, на первый план выходит задача не просто создания хороших машин, а создания хороших машин при условии их обратной совместимости.

На этом уровне ISA определяются реализованные в микропроцессоре конкретного типа архитектура памяти, взаимодействие с внешними устройствами ввода/вывода, режимы адресации, регистры, машинные команды, различные типы внутренних данных (например, с плавающей запятой, целочисленные типы и т.д.), обработчики прерываний исключительных состояний.

Таким образом, уровень АНК — это тот уровень, на котором компьютер представляется программисту, пишущему программы на машинном языке, или то, что получается в результате работы компилятора. Чтобы получить программу уровня АНК, создатель компилятора должен знать, какая модель памяти используется в машине, какие регистры, типы данных и команды имеются в наличии и т.д. Вся эта информация в совокупности и определяет уровень архитектуры набора команд.

4.1. Типы данных, структура и форматы команд, способы адресации

Знание структуры и форматов команд вычислительных машин является необходимой предпосылкой для программирования на машинном языке на данной архитектуры. Структура команд ЭВМ в общем случае зависит от функций, которые она должна выполнять. Эти функции, в свою очередь, определяются классом задач, для решения которых предназначена вычислительная машина, и принципов, заложенных при ее реализации. Функциональные требования, как правило, определяют те параметры, которые влияют на структуру команд процессора. В перечень этих параметров обычно включают: способы представления данных; список операций над этими данными; способы адресации данных; режимы работы машины, организацию системы прерываний и др.

Кроме того, в процессорах применяют команды различных форматов в зависимости от уровней памяти, т.е. в зависимости от того, адрес какого типа указан в адресной части команды — адрес, в котором указывается номер регистров общего назначения, или адрес, в котором указывается номер ячейки оперативной памяти. Отсюда различные типы форматов: команды типа регистр—регистр, регистр—память или память—память.

Типы данных

Основными типами данных в вычислительных машинах архитектуры x86 являются: байт, слово, двойное слово, квадрослово и 128-разрядное слово (рис. 4.1).

Каждый из представленных на рис. 1.9 типов данных может начинаться с любого адреса: это означает, что слово не обязано начинаться с четного адреса; двойное слово — с адреса, кратного 4, и т.д. Таким образом достигается максимальная гибкость структур данных и эффективность использования памяти.

На базе основных типов данных строятся все остальные типы, распознаваемые командами процессора.

Целочисленные данные. Четыре формата данных (байт, слово, двойное слово, учетверенное слово) с фиксированной точкой могут быть как со знаком, так и без знака. Под знак отводится старший бит формата данных. Представление таких данных и выполнение операций в арифметико-логическом устройстве (ALU) производятся в дополнительном коде.

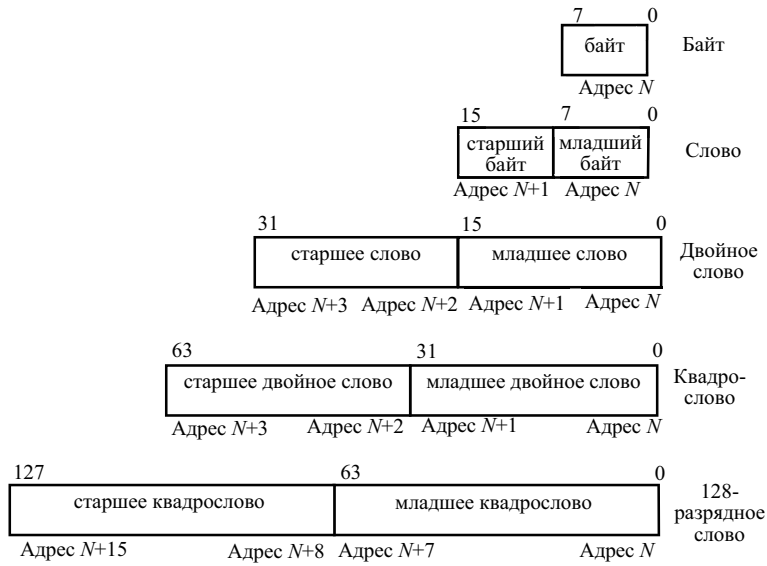


Рис. 4.1. Основные типы данных

Данные в формате с плавающей точкой x87. Формат включает три поля: знак (S), порядок и мантиссу (рис. 4.2). Поле мантиссы содержит значащие биты числа, а поле порядка содержит степень 2 и определяет

масштабирующий множитель для мантиссы. Форматы данных поддерживаются блоком обработки чисел с плавающей точкой (FPU).

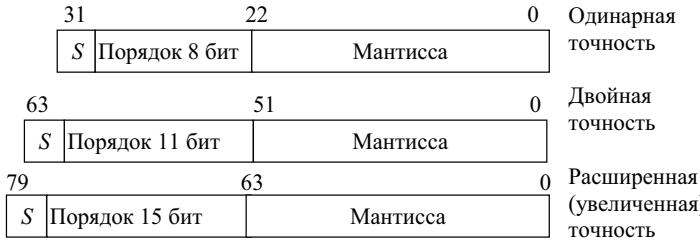


Рис. 4.2. Форматы данных с плавающей точкой

Двоично-десятичные данные (BCD). На рис. 4.3 приведены форматы двоично-десятичных данных.

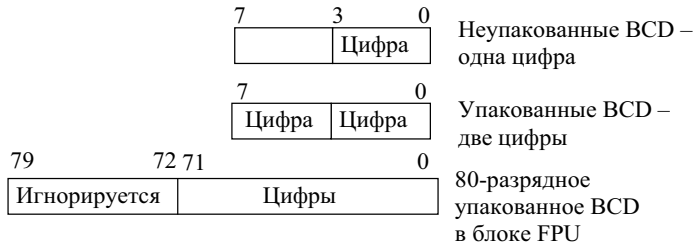


Рис. 4.3. Форматы двоично-десятичных данных

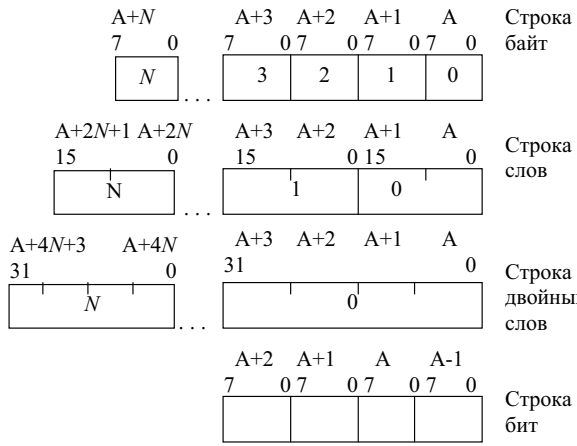


Рис. 4.4. Данные типа «строка»

Данные типа «строка». Строка представляет собой непрерывную последовательность битов, байтов, слов или двойных слов (рис. 4.4). Строка битов может быть длиной до 1 Гбита, а длина остальных строк может составлять от 1 байта до 4 Гбайтов. Поддерживается ALU.

Символьные данные. Поддерживаются строки символов в коде ASCII и арифметические операции (сложение, умножение) над ними (рис. 4.5). Поддержка осуществляется арифметико-логическим устройством.

7 + *N* 0		7 + 1 0	7 0 0
Символ *N*	...	Символ 1	Символ 0
ASCII		ASCII	ASCII

Рис. 4.5. Символьные данные

Данные типа «указатель». Указатель содержит величину, которая определяет адрес фрагмента данных. Поддерживается два типа указателей, приведенных на рис. 4.6.

<table>
<tr><td>4</td><td colspan="2">7 +5</td><td>+ 4</td><td>+ 3</td><td>+ 2</td><td>+ 1</td><td>0</td><td>0</td><td></td></tr>
<tr><td></td><td colspan="3"></td><td></td><td></td><td></td><td></td><td></td><td>Длинный указатель (дальний)</td></tr>
<tr><td></td><td colspan="3"></td><td></td><td></td><td></td><td></td><td></td><td></td></tr>
<tr><td></td><td colspan="3">селектор 16 р.</td><td colspan="5">смещение 32 р.</td><td></td></tr>
<tr><td></td><td></td><td colspan="2">3</td><td>1 + 3</td><td>+ 2</td><td>+ 1</td><td>0</td><td>0</td><td></td></tr>
<tr><td></td><td></td><td colspan="2"></td><td></td><td></td><td></td><td></td><td></td><td>Короткий указатель (ближний)</td></tr>
<tr><td></td><td></td><td colspan="2"></td><td></td><td></td><td></td><td></td><td></td><td></td></tr>
<tr><td></td><td></td><td colspan="2"></td><td colspan="5">смещение 32 р.</td><td></td></tr>
</table>

Рис. 4.6. Данные типа «указатель»

Данные MMX-технологии. Целочисленные данные могут быть как со знаком, так и без знака (рис. 4.7).

Рис. 4.7. Данные MMX-технологии

Данные SSE-расширения. На рис. 4.8 приведен 128-разрядный формат упакованных данных с плавающей точкой одинарной точности.

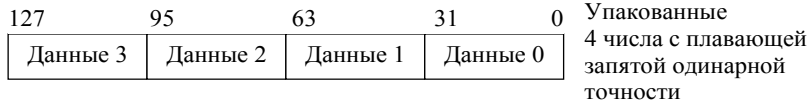


Рис. 4.8. Данные SSE-расширения

Данные расширения SSE2. На рис. 4.9 приведен 128-разрядный формат упакованных данных с плавающей точкой с двойной точностью.

Рис. 4.9. Данные SSE2 расширения с плавающей запятой

На рис. 4.10 показаны четыре формата упакованных в 128 бит целочисленных данных, которые могут быть как со знаком, так и без знака.

Рис. 4.10. Целочисленные данные SSE2 расширения

Данные в IA-64. В IA-64 непосредственно поддерживается шесть типов данных, в том числе три формата, используемых ранее (одинарная точность, двойная точность, расширенная точность), 82-разрядный формат данных с плавающей точкой (рис. 4.11) и 64-разрядные целые — со знаком и без знака.

81		630
S	Порядок 17 р.	Мантисса

Рис. 4.11. Формат данных с плавающей точкой 82-разрядный

Структура и форматы команд

Обработка информации в ЭВМ осуществляется автоматически путем программного управления. Программа представляет собой алгоритм обработки информации, записанный в виде последовательности команд, которые должны быть выполнены для получения решения задачи.

Команда представляет собой код, определяющий операцию вычислительной машины и данные, участвующие в операции.

Команды микропроцессора, в отличие от микрокоманд, разрабатываются независимо от аппаратуры микросхемы, поэтому их разрядность обычно совпадает с разрядностью микропроцессора. Команда микропроцессора состоит из инструкции и обозначается «код операции» (КОП) (или INS в англоязычной литературе). Команда микропроцессора может состоять только из кода операции, когда не требуется указывать адрес операнда (операнды — это данные, над которыми команда производит какое-либо действие), или может состоять из кода операции и адресов операндов или данных. Форматы команд очень сильно зависят от структуры процессора.

Команда содержит также в явной или неявной форме информацию об адресе, по которому помещается результат операции, и об адресе следующей команды.

По характеру выполняемых операций различают следующие основные группы команд:

а) команды арифметических операций над числами с фиксированной и плавающей точками;

б) команды десятичной арифметики;

в) команды логических операций и сдвигов;

г) команды передачи кодов;

д) команды операций ввода/вывода;

е) команды передачи управления;

ж) команды векторной обработки;

з) команды задания режима работы машины и др.

Команда в общем случае состоит из операционной и адресной частей (рис. 4.12, *а*).

В свою очередь, эти части, что особенно характерно для адресной части, могут состоять из нескольких полей.

Операционная часть содержит код операции (КОП), который задает вид операции (сложение, умножение и др.). Адресная часть содержит информацию об адресах операндов и результате операции.

Структура команды определяется составом, назначением и расположением полей в команде.

Форматом команды называют ее структуру с разметкой номеров разрядов (битов), определяющих границы отдельных полей команды, или с указанием числа битов в определенных полях.

Важной и сложной проблемой при проектировании вычислительных машин является выбор структуры и форматов команды, т.е. ее длины, назначения и размерности отдельных ее полей. Естественно стремление разместить в команде в возможно более полной форме информацию о предписываемой командой операции. Однако в условиях, когда в современных ЭВМ значительно возросло число выполняемых различных операций и соответственно команд (в системе команд x86 более 500 команд) и значительно увеличилась емкость адресуемой основной памяти (4, 6 Гбайт), это приводит к недопустимо большой длине формата команды.

Действительно, число двоичных разрядов, отводимых под код операции, должно быть таким, чтобы можно было представить все выполняемые машинные операции. Если ЭВМ выполняет M различных операций, то число разрядов в коде операции

$$n_{\text{коп}} \geq \log_2 M.$$

Например, при $M = 500$ $n_{\text{коп}} = 9$.

Если основная память содержит S адресуемых ячеек (байтов), то для явного представления только одного адреса необходимо в команде иметь адресное поле для одного операнда с числом разрядов

$$n_{\text{A}} \geq \log_2 S.$$

Например, при $S = 4$ Гбайт $n_{\text{A}} = 32$.

Отмечавшиеся ранее характерные для процесса развития ЭВМ расширение системы (наборы) команд и увеличение емкости основной памяти, а особенно создание микроЭВМ с коротким словом, потребовали разработки методов сокращения длины команды. При решении этой проблемы существенно видоизменилась структура команды, получили развитие различные способы адресации информации.

Проследим изменения классических структур команд.

Чтобы команда содержала в явном виде всю необходимую информацию о задаваемой операции, она должна, как это показано на рис. 4.12, *б*, содержать следующую информацию:

- A_1, A_2 — адреса операндов;
- A_3 — адрес результата;
- A_4 –адрес следующей команды (принудительная адресация команд).

Такая структура приводит к большой длине команды (например, при $M = 500$, $S = 4$ Гб длина команды — 137 бит) и неприемлема для прямой адресации операндов основной памяти. В компьютерах с RISC-архитектурой четырехадресные команды используются для адресации операндов, хранящихся в регистровой памяти процессора.

Можно установить, что после выполнения данной команды, расположенной по адресу K (и занимающей L ячеек), выполняется команда из $(K + L)$-й ячейки. Такой порядок выборки команды называется естественным. Он нарушается только специальными командами передачи управления. В таком случае отпадает необходимость указывать в команде в явном виде адрес следующей команды.

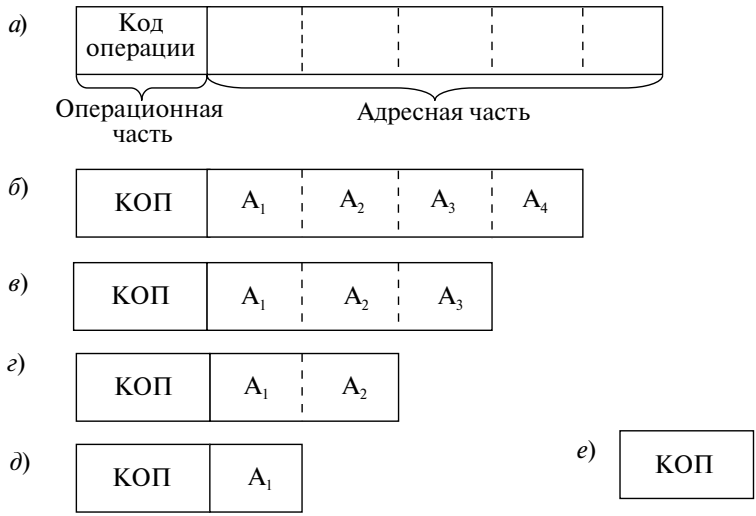


Рис. 4.12. Структуры команд:
а — обобщенная; *б* — четырехадресная; *в* — трехадресная; *г* — двухадресная; *д* — одноадресная; *е* — безадресная

В одноадресной команде (рис. 4.12, *д*) подразумеваемые адреса имеют уже и результат операции, и один из операндов. Один из операндов указывается адресом в команде, в качестве второго используется содержимое регистра процессора, называемого в этом случае

регистром результата или аккумулятором. Результат операции записывается в тот же регистр. Наконец, в некоторых случаях возможно использование безадресных команд (рис. 4.12, *е*), когда подразумеваются адреса обоих операндов и результата операции, например при работе со стековой памятью.

В трехадресной команде (рис. 4.12, *в*) первый и второй адреса указывают ячейки памяти, в которых расположены операнды, а третий определяет ячейку, в которую помещается результат операции. Можно условиться, что результат операции всегда помещается на место одного из операндов, например первого. Получим двухадресную команду (рис. 4.12, *г*), т.е. для результата используется подразумеваемый адрес.

Следует различать понятия «адресный код в команде A_R» и «исполнительный (физический адрес) адрес $A_и$».

Адресный код — это информация об адресе операнда, содержащаяся в команде.

Исполнительный адрес — это номер ячейки памяти, к которой производится фактическое обращение.

Таким образом, способ адресации можно определить как способ формирования исполнительного адреса операнда $A_и$ по адресному коду команды $A_к$.

В системах команд современных ЭВМ часто предусматривается возможность использования нескольких способов адресации операндов для одной и той же операции. Для указания способа адресации в некоторых системах команд выделяется специальное поле в команде — поле «метод» (указатель адресации, УА), рис. 4.13, *а*.

В этом случае любая операция может выполняться с любым способом адресации, что значительно упрощает программирование.

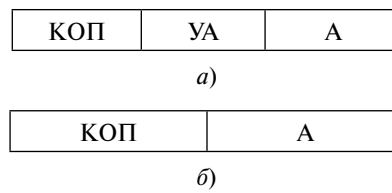


Рис. 4.13. Общая структура команды:
а — с указателем метода адресации; *б* — без указателя метода адресации

Классификация способов адресации по наличию адресной информации в команде:

- явная адресация;
- неявная адресация.

При *явной* адресации операнда в команде есть поле адреса этого операнда, в котором задается адресный код $A_к$. Большинство методов адресации являются явными.

При *неявной* адресации адресное поле в команде отсутствует, адрес операнда подразумевается кодом операции.

Классификация способов адресации по кратности обращения в память. Широко используются следующие методы адресации операнда с различной кратностью обращения (r) в память:

1) непосредственная ($r = 0$);

2) прямая ($r = 1$);

3) косвенная ($r \geq 2$).

Непосредственная адресация операнда. Обращение к регистровой памяти (РП) или ОП не производится ($R = 0$). Таким образом уменьшается время выполнения операции, сокращается объем памяти.

Прямая адресация операндов. При этом способе (рис. 4.13) адресации обращение за операндом в РП или ОП производится по адресному коду в поле команды, т.е. исполнительный адрес операнда совпадает с адресным кодом команды ($A_{исп} = A_к$). Обеспечивая простоту программирования, этот метод имеет существенные недостатки, так как для адресации к ячейкам памяти большой емкости требуется «длинное» адресное поле в команде.

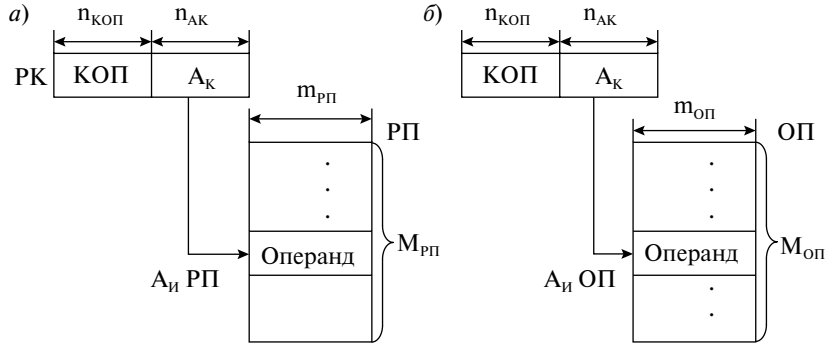


Рис. 4.14. Схема прямой адресации:
а — к регистровой памяти; *б* — к основной памяти

Косвенная адресация операндов. Адресный код команды указывает адрес ячейки памяти, в которой находится не сам операнд, а лишь адрес операнда, называемый указателем операнда. Адресация к операнду через цепочку указателей (косвенных адресов) называется косвенной.

Адрес указателя остается неизменным, а косвенный адрес может изменяться в процессе выполнения программы. Это обеспечивает переадресацию данных, т.е. упрощает обработку массивов и списковых структур данных, упрощает передачу параметров подпрограммам, но не обеспечивает перемещаемость программ в памяти (рис. 4.15, *а*).

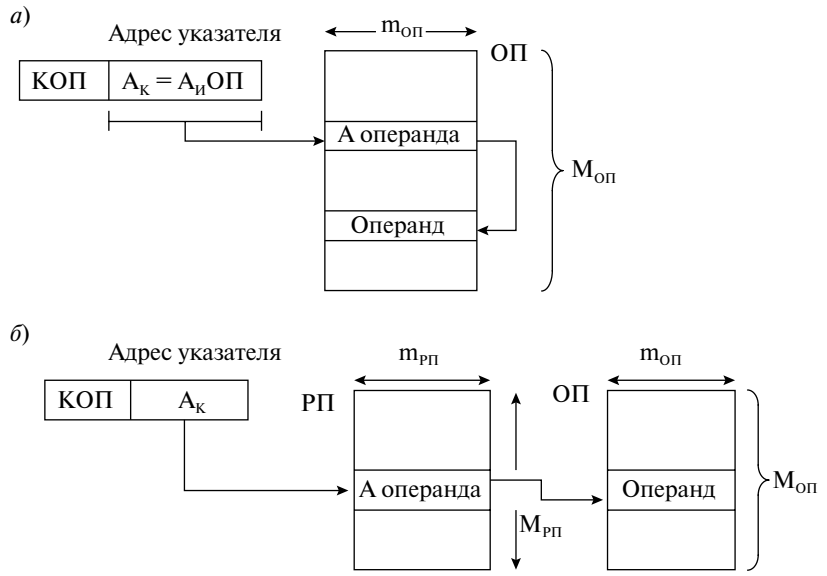


Рис. 4.15. Схема косвенной адресации:
а — указатель операнда и операнд расположены в одном адресном пространстве оперативной памяти; *б* — указатель операнда расположен в регистровой памяти, а операнд — в адресном пространстве оперативной памяти

В табл. 4.1 представлены основные методы адресации операндов на примере команды сложения (Add), хотя при описании архитектуры в документации разные производители используют разные названия для этих методов. В этой таблице знак «<—» используется для обозначения оператора присваивания, а буква M обозначает память (Memory). Таким образом, M[R1] обозначает содержимое ячейки памяти, адрес которой определяется содержимым регистра R1.

Использование сложных методов адресации позволяет существенно сократить количество команд в программе, но при этом значительно увеличивается сложность аппаратуры.

Таблица 4.1

Методы адресации операндов

Метод адресации	Пример команды	Смысл команды метода	Использование
Регистровая	Add R4, R3	R4<—R4+R5	Требуемое значение в регистре
Непосредственная или литеральная	Add R4,#3	R4<—R4+3	Для задания констант
Базовая со смещением	Add R4,100(R1)	R4<—R4+M[100+R1]	Для обращения к локальным переменным
Косвенная регистровая	Add R4,(R1)	R4<—R4+M[R1]	Для обращения по указателю или вычисленному адресу
Индексная	Add R3,(R1+R2)	R3<—R3+M[R1+R2]	Иногда полезна при работе с массивами: R1 — база, R3 — индекс
Прямая или абсолютная	Add R1,(1000)	R1<—R1+M[1000]	Иногда полезна для обращения к статическим данным
Косвенная	Add R1,@(R3)	R1<—R1+M[M[R3]]	Если R3-адрес указателя p, то выбирается значение по этому указателю
Автоинкрементная	Add R1,(R2)+	R1<—R1+M[R2] R2<—R2+d	Полезна для прохода в цикле по массиву с шагом; R2 — начало массива. В каждом цикле R2 получает приращение d
Автодекрементная	Add R1,(R2)-	R2<—R2-d R1<—R1+M[R2]	Аналогична предыдущей. Обе могут использоваться для реализации стека
Базовая индексная со смещением и масштабированием	Add R1,100(R2)[R3]	R1<—R1+M[100]+R2+ R3*d	Для индексации массивов

Примеры использования адресаций

К числу важнейших способов адресации относятся прямая адресация и косвенная адресация.

Адрес задается в квадратных скобках, например [0101] — прямая адресация, [BP] — косвенная адресация.

Для обозначения содержимого ячеек оперативной памяти при записи команд в квадратных скобках указывается не содержимое ячейки оперативной памяти, а адрес ячейки, например:

- [0150] — содержимое 1-байтовой ячейки оперативной памяти;
- 0150 — адрес 1-байтовой ячейки оперативной памяти;
- [BP] — содержимое 1-байтовой ячейки оперативной памяти;
- BP — регистр, содержащий адрес 1-байтовой ячейки памяти.

Рассмотрим примеры использования адресации.

1. *Пример использования прямой адресации с 1-байтовыми операндами*:

MOV AL, [0150].

До выполнения: [0102]= 3D.

После выполнения: [0102]= 3D и AL= 3D.

2. *Пример использования прямой адресации с 2-байтовыми операндами*:

MOV AX, [0150].

До выполнения: [0150]= 9F [0151]= 5A.

После выполнения: [0150]= 9F [0151]= 5A.

и AX= 5A9F, то есть AL= 9F и AH= 5A.

3. *Пример использования косвенной адресации с 1-байтовыми операндами*:

MOV AL, [SI];

До выполнения: [0150]= 9F SI= 0150;

После выполнения: [0150]= 9F SI= 0150

и AL= 9F.

Изменение порядка операндов отражается на результате следующим образом.

4. *Пример использования косвенной адресации с 2-байтовыми операндами*:

MOV AX, [SI];

До выполнения: [0150]= 9F [0151]= 5A SI= 0150;

После выполнения: [0150]= 9F [0151]= 5A SI= 0150;

и AX= 5A9F, то есть AL= 9F и AH= 5A.

5. *Практическое использование способов адресации (на примере команды MOV)*:

Команда MOV:

Адрес начала команды	Команда, записанная на языке машинных кодов	Та же самая команда, записанная на языке ассемблер	Комментарий, поясняющий, как выполняется команда
100	B8 34 12	MOV AX,1234	НЕПОСРЕДСТВЕННАЯ АДРЕСАЦИЯ: Занесение 2-байтового числа 1234 (12 — старший байт числа, 34 — младший байт числа) в 2-байтовый регистр AX
103	A3 20 01	MOV [0120], AX	ПРЯМАЯ АДРЕСАЦИЯ: Копирование 2-байтового числа, находящегося в AX, в 1-байтовые ячейки оперативной памяти 0120 и 0121 (в 0120 копируется младший байт AX, в 0121 копируется старший байт AX)
106	8B 1E 20 01	MOV BX, [0120]	ПРЯМАЯ АДРЕСАЦИЯ: Копирование содержимого ячеек оперативной памяти 0120 и 0121, в 2-байтовый регистр BX (в младший байт BX, регистр BL, копируется содержимое ячейки 0120, в старший байт BX, регистр BH, копируется содержимое ячейки 0121)
10A	BD 38 01	MOV BP,0138	НЕПОСРЕДСТВЕННАЯ АДРЕСАЦИЯ: Занесение 2-байтового числа 0138 в 2-байтовый регистр BP
10D	88 46 00	MOV [BP+00], AL	КОСВЕННАЯ АДРЕСАЦИЯ: Копирование 1-байтового числа, находящегося в AL, в 1-байтовую ячейку оперативной памяти, адрес которой содержится в регистре BP, т.е. в ячейку с вычисляемым адресом 0138(0138+00=0138)

Адрес начала команды	Команда, записанная на языке машинных кодов	Та же самая команда, записанная на языке ассемблер	Комментарий, поясняющий, как выполняется команда
110	894607	MOV [BP+07], AX	КОСВЕННАЯ АДРЕСАЦИЯ: Копирование 2-байтового числа, находящегося в AX, в ячейки оперативной памяти, на адреса которых указывает регистр BP: младший байт регистра AX, именуемый AL, копируется в ячейку с вычисляемым адресом 013F(0138 + 07 + 0 = 013F); старший байт регистра AX, именуемый AH, копируется в ячейку с вычисляемым адресом 0140(0138 + 07 + 1 = 0140)
113	CD 20	INT 20	Команда завершения работы программы

Типы команд

Команды традиционного машинного уровня можно разделить на несколько типов, которые показаны в табл. 4.2.

Таблица 4.2

Основные типы команд

Тип операции	**Примеры**
Арифметические и логические	Целочисленные арифметические и логические операции: сложение, вычитание, логическое сложение, логическое умножение и т.д.
Пересылки данных	Операции загрузки/записи
Управление потоком команд	Безусловные и условные переходы, вызовы процедур и возвраты
Системные операции	Системные вызовы, команды управления виртуальной памятью и т.д.
Операции с плавающей точкой	Операции сложения, вычитания, умножения и деления над вещественными числами
Десятичные операции	Десятичное сложение, умножение, преобразование форматов и т.д.
Операции над строками	Пересылки, сравнения и поиск строк

Типы и размеры операндов

Имеется два альтернативных метода определения типа операнда. В первом из них тип операнда может задаваться кодом операции в команде. Это наиболее употребительный способ задания типа операнда. Второй метод предполагает указание типа операнда с помощью тега, который хранится вместе с данными и интерпретируется аппаратурой во время выполнения операций над данными. Этот метод использовался, например, в машинах фирмы «Burroughs», но в настоящее время он практически не применяется и все современные процессоры пользуются первым методом.

Обычно тип операнда (например, целый, вещественный с одинарной точностью или символ) определяет и его размер. Однако часто процессоры работают с целыми числами длиною 8, 16, 32 или 64 бит. Как правило, целые числа представляются в дополнительном коде. Для задания символов (1 байт = 8 бит) в машинах компании «IBM» используется код EBCDIC, но в машинах других производителей почти повсеместно применяется кодировка ASCII. Еще до сравнительно недавнего времени (10 лет назад) каждый производитель процессоров пользовался своим собственным представлением вещественных чисел (чисел с плавающей точкой). Однако за последние несколько лет ситуация изменилась. Большинство поставщиков процессоров в настоящее время для представления вещественных чисел с одинарной и двойной точностью придерживаются стандарта IEEE 754.

В некоторых процессорах используются двоично кодированные десятичные числа, которые представляются в упакованном и неупакованном форматах. *Упакованный* формат предполагает, что для кодирования цифр 0–9 используются четыре разряда и что две десятичные цифры упаковываются в каждый байт. В *неупакованном* формате байт содержит одну десятичную цифру, которая обычно изображается в символьном коде ASCII.

В большинстве процессоров, кроме того, реализуются операции над цепочками (строками) битов, байтов, слов и двойных слов.

Рассмотрим особенности некоторых групп команд.

Команды передачи управления. Для определения адреса текущей команды МП имеет в своем составе специальный регистр — указатель адреса команды или счетчик команд PC, IP.

Модификация PC происходит сразу после выборки команды (или ее байта). Поскольку чаще всего используется естественная адресация команд, то возникает необходимость в командах передачи управления для реализации ветвления алгоритмов. Для этого используются специальные команды:

- команды перехода;
- команды замещения;
- команды смены состояния процессора;
- команды запроса прерывания.

Команды перехода. Адресная часть команды непосредственно или после суммирования с содержимым базового регистра передается в счетчик команд, т.е. адрес следующей команды задается командой перехода.

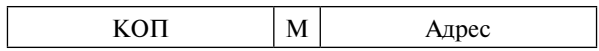
Используются команды безусловного и условного переходов.

Команды *безусловного* перехода

Переход может осуществляться и по косвенному адресу. На косвенную адресацию указывает либо КОП, либо специальный бит в поле команды.

Команды *условного* перехода

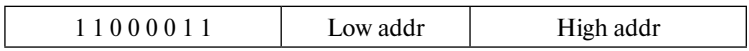
Адрес следующей команды зависит от выполнения некоторого условия:



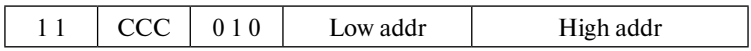
M — код признака (маска условия).

Команды могут быть с относительной и косвенной адресацией.

Пример 4.1. МП 8080.



JMP addr



после маски условия

J condition addr

Если условие выполняется, то в счетчик команд загружается новый адрес.

Команды вызова подпрограмм. Отличие от команд перехода заключается в том, чтобы по окончании подпрограммы реализовать возврат в прежнюю точку программы. Для этого необходимо сохранить адрес возврата.

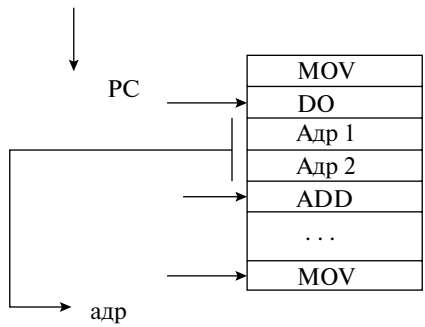
Перед выполнением передачи управления содержимое PC, указывающее на следующую команду программы, запоминается по адресу, указываемому в команде (обычно регистр или стек). При этом организуется дополнительная команда возврата из подпрограммы, которая восстанавливает содержимое счетчика команд.

МП 8080.

CALL	11001101		Low addr	High addr
Условный вызов C condition addr				
	11	CCC	100	Low addr	High addr
RET	11001001			
R condition	11	CCC	000	

Команды замещения. Команда замещения — вместо очередной команды используется замещающая команда, находящаяся по адресу, указанному в команде «выполнить». Выполнение этой замещающей команды не должно приводить к изменению PC. После исполнения этой команды продолжается естественный ход программы (это не JMP).

Команда «выполнение» — аналог подпрограммы, состоящей из одной команды без сохранения адреса возврата.



Модификация команд

Изменения команд программы могут быть как проведены самим микропроцессором, так и заложены в алгоритм выполнения программы, например изменение адреса данных для команды обработки. Такой способ называется модификацией команд и основан на воз-

можности проведения арифметических и логических операций над командами.

Автоматическая модификация команд и управление вычислением циклов в ЭВМ обеспечиваются механизмом индексации. Это понятие включает в себя специальный способ кодирования команд, командные и аппаратные средства задания и выполнения команд. Метод является развитием метода относительной адресации.

Для выполнения индексации в МП включены индексные регистры. В формате команды выделяется поле X для указания индексного регистра. Исполнительный адрес при индексации формируется путем сложения адресного кода команды (смещение) с содержанием индексного регистра, а при наличии базирования — и с базовым адресом (рис. 4.16).

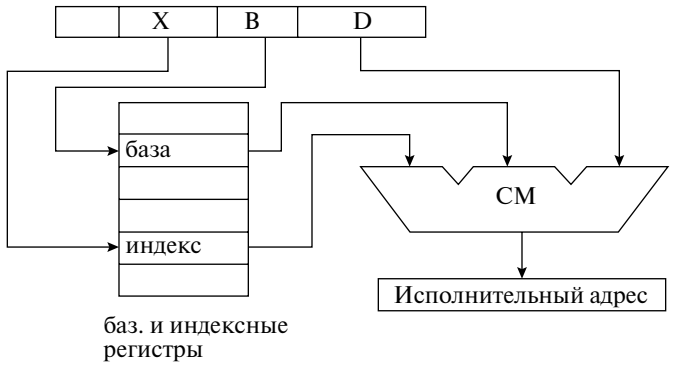


Рис. 4.16. Формирование исполнительного адреса операнда при индексации

Три вида индексных операций:

а) засылка в индексный регистр начального значения;

б) изменение индекса;

в) проверка окончания циклических вычислений.

Пример: команды условного перехода по счетчику (DJNZ).

КОП	R	X	B	D

1) уменьшаем на 1 содержание счетчика R1 (индексный регистр);

2) если R1 не равно 0, то $A_{исп}$=(B2)+B2+(Ч2),

иначе переход не осуществлять.

Пример: цепочные команды MOVS адресация с помощью индексных регистров SI.

Использование самоопределяемых данных. Понятие тегов и дескрипторов

Теги. В большинстве случаев формат данных, с которыми выполняется команда, указывается в самом коде операции. Однако это приводит к избыточности кодирования команд, так как при наличии нескольких команд, работающих с одними и теми же данными, мы фактически несколько раз указываем их тип.

При теговой организации каждое хранимое в памяти слово снабжается указателем — *тегом*, определяющим тип данных (целое, двоичное число, число с ПТ, адрес, строка и т.д.).

Тег	Данные

В поле тега кроме типа обычно указывают длину и формат данных и другие параметры. Теги формируются компилятором и прикрепляются к данным.

Наличие тегов придает данным свойство самоопределяемости, т.е. данные описывают себя самостоятельно.

Рассмотрим пример: в обычной ЭВМ тип данных определяется контекстно задаче, т.е. непосредственно командой, которая использует эти данные:

ADD B	ADD BX	ADD EBX
8 разр.	16 разр.	32 разр.

В этом случае код команды задает вид (тип) данных. Таким образом, команд сложения может быть очень много: с ФТ, с ПТ, с разрядностью 8, 16, 32 бита. Теговая же организация позволяет достигнуть инвариантности команд относительно данных, что приводит к значительному сокращению набора команд машины (в нашем случае остается всего одна команда сложения). Это упрощает структуру процессора, облегчает работу программиста, облегчает обнаружение ошибок при операциях с некорректными данными.

Достоинства:

- теговая организация памяти способствует реализации принципа независимости программ от данных;
- приводит к экономии памяти, так как отсутствует информационная избыточность на задание типов и размеров операндов при их использовании несколькими командами.

Недостаток: уменьшается скорость работы МП из-за установки соответствия типа команд типам данных, происходящей не на этапе компиляции, а при выполнении программы.

Дескрипторы. Дескрипторы — служебные слова, содержащие информацию (описание) массивов данных и команд. Дескрипторы могут применяться как с тегами, так и без них.

Дескриптор содержит сведения:

- о размере массива данных;
- местоположении массива в ОП или во внешней памяти;
- адресе начала массива;
- режиме защиты данных.

На рис. 4.17 приведен пример описания двухмерного массива дескрипторами древовидной структуры.

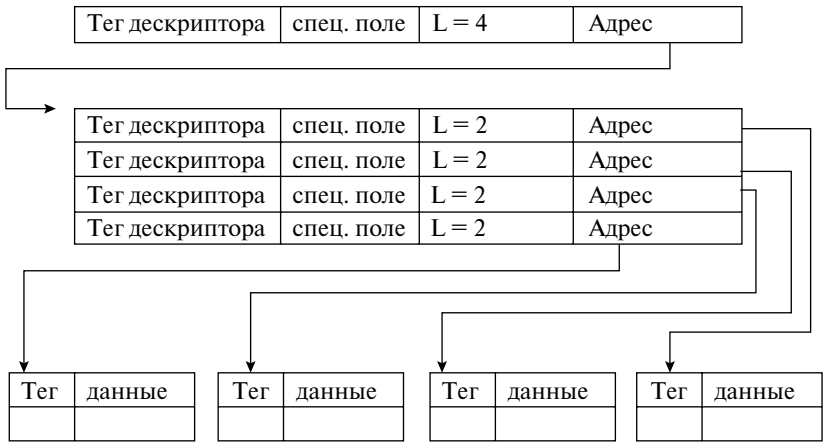


Рис. 4.17. Описание двухмерного массива

Наличие в архитектуре ЭВМ дескрипторов подразумевает, что обращение к информации в памяти производится через дескрипторы, которые можно рассматривать как дальнейшее развитие аппаратно-косвенной адресации.

4.2. Организация процессора и основной памяти

Рассмотрим машины с контроллерным управлением, в которых порядок выполнения команд явно задается программой.

Процессор выполняет две функции:

- обработку данных в соответствии с заданной программой;
- управление всеми устройствами машины.

Управление в соответствии с заданной программой представляется в виде последовательности команд в цифровой форме. Как мы уже знаем, каждая из команд имеет две части: операционную и адресную.

Операционная часть задает код операции и режим ее выполнения. Адресная часть содержит сведения о размещении операндов (данных): непосредственно сами значения данных, адреса данных в памяти или сведения для определения адресов размещения данных в памяти. Формирование исполнительного адреса — этап перехода от сведений об адресе к самому адресу. В адресной части могут быть сведения об отсутствии операндов (нуль-адресная или безадресная команда) и от одного (одноадресная команда) до трех операндов (трехадресная команда).

Типовая структура процессора и основной памяти

Рассмотрим типовую структуру процессора и основной памяти, представленную на рис. 4.18.

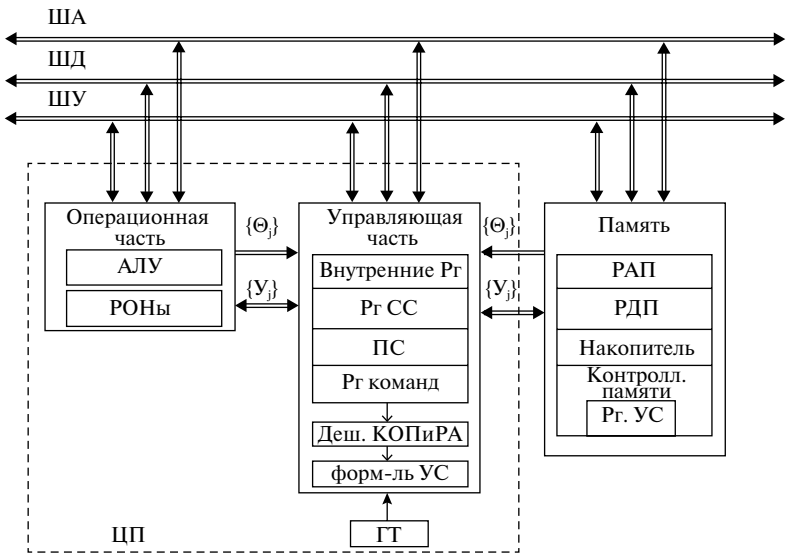


Рис. 4.18. Типовая структура процессора и основной памяти

АЛУ — арифметико-логическое устройство.

РОНы — регистры общего назначения.

Рг СС — регистр слова состояния. Содержит текущее состояние процессора, в который входит уровень приоритета текущей программы, биты условий завершения последней команды, режим обработки текущей команды.

Возможны следующие режимы обработки (в порядке возрастания уровня):

- User Mode — режим пользователя; в этом режиме не могут выполняться системные команды (команды изменения состояния процессора и команды ввода-вывода);
- SuperVisor Mode — режим супервизора; обеспечивается выполнение всех команд ввода-вывода;
- Kernel Mode — режим ядра; возможно выполнение всех команд процессора.

ПС — программный счетчик. Содержит адрес текущей команды и автоматически наращивается для подготовки адреса следующей команды (исключение составляет команда перехода);

Рг команд — регистр команд. Содержит код исполняемой в данный момент команды.

ДешКОПиРА — дешифратор кода операции и режимов адресации.

Форм-льУС — формирователь управляющих сигналов {У*i*}.

РАП — регистр адреса памяти.

РДП — регистр данных памяти.

Рг УС — регистр управляющего слова контроллера памяти.

Основной цикл работы процессора

Этапы:

- выборка команды (IF);
- формирование исполнительных адресов операндов, если требуется (AM);
- выборка операндов из памяти (OF);
- исполнение операции (EX);
- запоминание результата (ST);
- проверка запроса программного прерывания (IRQ).

Каждый этап выполняется за один цикл памяти. При обработке этих этапов используется конвейерный способ (при выполнении очередного этапа одновременно происходит выполнение предыдущего этапа следующей команды) (рис. 4.19). Разумеется, если одна команда изменяет содержимое какой-либо ячейки памяти, а следующая использует ее новое значение, то последняя не может начать исполняться, пока результат не будет сохранен.

IF	AM	OF	EX	ST	IRO		
	IF	AM	OF	EX	ST	IRQ	
		IF	AM	OF	EX	ST	IRQ

Рис. 4.19. Конвейерный способ выполнения команд микропроцессора

Возможны два вида прерываний:

1) программное, которое обрабатывается путем выполнения процессором специальной программы — обработчика прерываний;

2) аппаратное, реализуется без участия программы процессора с помощью аппаратных средств, имеет высокий приоритет и может поступать в любой момент времени, прерывая выполнение текущей команды.

Подробнее о прерываниях будет рассказано позже.

Организация процессора и памяти в микропроцессоре Intel 8086

В данном процессоре длина слова составляет 16 бит, что равно 2 байтам. Минимально адресуемой и обрабатываемой единицей информации является байт, при этом адрес слова совпадает с адресом младшего байта и является четным. Максимальная емкость памяти $2^{16} = 64$ Кб. Для расширения адресного пространства используется сегментирование памяти. Максимальная длина каждого сегмента равна 64 Кб. Адрес образуется парой — сегмент (16 бит) и смещение (16 бит) в пределах сегмента:

Segment: Offset.

Таблица 4.3

Распределение оперативной памяти в i8086

ГРАНИЦЫ УЧАЧСТКА (Кб)	СЕГМЕНТ: СМЕЩЕНИЕ	НАЗНАЧЕНИЕ УЧАСТКА
0..1	0000:0000(03FF)	256 векторов прерываний
1..60	0010:0000	Область данных и резидентная часть DOS IO.SYS — расширитель BIOS: • настройка на конфигурацию системы; • установка новых драйверов; • исправление ошибок и неточностей. BIOS применительно к данной системе. MSDOS.COM — обработчик прерываний операционной системы: • прерывания DOS; • функция DOS (21h). Резидентная часть COMMAND.COM: • обработка командных файлов; • инициирование запуска остальных исполняемых файлов
60..640		Данные, программы пользователя
640..768	A000:0000	Область видеоадаптеров
768..1016	C000:0000 FE00:0000	Область ПЗУ; Область BIOS

Смещение сдвигается на 4 бита влево и суммируется со значением сегмента, результатом является 20-битный физический адрес, чем обеспечивается адресация 1 Мб памяти. Пример распределения оперативной памяти в микропроцессоре Intel 8086 приведен в табл. 4.3.

Программно-доступные регистры процессора

Начиная с 80386, процессоры Intel предоставляют 16 основных регистров для пользовательских программ плюс еще 11 регистров для работы с числами с плавающей запятой (FPU/NPX) и мультимедийными приложениями (MMX). Все команды так или иначе изменяют значения регистров, и всегда быстрее и удобнее обращаться к регистру, чем к памяти.

Регистры общего назначения. 16-битные регистры AX (аккумулятор), BX (база), CX (счетчик), DX (регистр данных) могут использоваться без ограничений для любых целей — временного хранения данных, аргументов или результатов различных операций. На самом деле, начиная с процессора 80386, все эти регистры имеют размер 32 бита и называются они EAX, EBX, ECX, EDX. Кроме этого, отдельные байты в 16-битных регистрах AX—DX тоже имеют свои имена и могут использоваться как 8-битные регистры. Старшие байты этих регистров называются AH, BH, CH, DH, а младшие — AL, DL, CL, DL.

Другие четыре регистра общего назначения: SI (индекс источника), DI (индекс приемника), BP (указатель базы), SP (указатель стека) — имеют более конкретное назначение и могут применяться для хранения всевозможных временных переменных, только когда они не используются по назначению. Регистры SI и DI используются в строковых операциях, BP и SP используются при работе со стеком. Так же как и с регистрами AX—DX, начиная с процессора 80386, эти четыре регистра являются 32-битными и называются ESI, EDI, EBP и ESP соответственно.

Сегментные регистры. При использовании памяти для формирования любого адреса применяются два числа — адрес начала сегмента и смещение искомого байта относительно этого начала. В процессорах Intel предусмотрены следующие сегментные регистры: CS (сегмент кода), DS (сегмент данных), ES (дополнительный сегмент), SS (сегмент стека). Начиная с 80286, появились регистры FS и GS.

Смещение следующей выполняемой команды всегда хранится в специальном регистре — IP (указатель инструкции), запись в который приведет к тому, что следующей будет исполнена какая-нибудь другая команда, а не команда, расположенная сразу за данной. На самом деле все команды передачи управления: перехода, условного перехода, цикла, вызова подпрограммы и т.п. — осуществляют запись в CS и EIP.

Регистр флагов.

□	□	□	□	OF	DF	IF	TF	SF	ZF	0	AF	0	PF	1	CF
15							8	7							0

CF — флаг переноса (CARRY);
PF — флаг четности (PARITY);
AF — дополнительный флаг переноса (AUXILARY);
ZF — флаг нуля (ZERO);
SF — знаковый флаг (SIGN);
TF — флаг слежения, ловушка (TRAP);
IF — флаг прерываний (INTERRUPTION);
DF — флаг направления (DIRECTION);
OF — флаг переполнения (OVERFLOW).

Организация стека процессора

Стек может работать только со словами, заполнение стека происходит в сторону уменьшения адресов.

Команда PUSH действует так: сначала значение регистра SP уменьшает на 2 (вычитание происходит по модулю 2^{16}), т.е. SP сдвигается вверх и теперь указывает на свободную ячейку области стека, а затем в нее записывается операнд. Команда POP считывает («выталкивает» слово из вершины стека, а затем SP увеличивает на 2 (сложение происходит по модулю 2^{16}), т.е. сдвигается вниз. Более подробно это будет рассмотрено в главе «Уровень ассемблера».

PUSHAX	POPAX
(SP)-2→SP	[SP]→AX
(AX)→[SP]	(SP)+2→SP

Использование:

- промежуточное хранение содержимого регистра;
- обмен содержимого регистров;
- сохранение адресов возврата при вызове подпрограмм;
- передача параметров между вызываемой и вызывающей программами;
- для обработки прерываний и сохранения векторов прерываний.

4.3. Организация прерываний в процессоре

Система прерываний

Для обработки событий, происходящих асинхронно по отношению к выполнению программы, лучше всего подходит механизм

прерываний. Прерывание можно рассматривать как некоторое особое событие в системе, требующее моментальной реакции. Например, хорошо спроектированные системы повышенной надежности используют прерывание по аварии в питающей сети для выполнения процедур записи содержимого регистров и оперативной памяти на магнитный носитель, с тем чтобы после восстановления питания можно было продолжить работу с того же места.

Кажется очевидным, что возможны самые разнообразные прерывания по самым различным причинам. Поэтому прерывание рассматривается не просто как таковое, с ним связывают число, называемое номером типа прерывания или просто номером прерывания. С каждым номером прерывания связывается то или иное событие. Система умеет распознавать, какое прерывание, с каким номером произошло, и запускает соответствующую этому номеру процедуру.

Программы могут сами вызывать прерывания с заданным номером. Для этого они используют команду INT. Это так называемые программные прерывания. Программные прерывания не являются асинхронными, так как вызываются из программы (а она-то знает, когда она вызывает прерывание!).

Программные прерывания удобно использовать для организации доступа к отдельным, общим для всех программ модулям. Например, программные модули операционной системы доступны прикладным программам именно через прерывания, и нет необходимости при вызове этих модулей знать их текущий адрес в памяти. Прикладные программы могут сами устанавливать свои обработчики прерываний для их последующего использования другими программами. Для этого встраиваемые обработчики прерываний должны быть резидентными в памяти.

Аппаратные прерывания вызываются физическими устройствами и приходят асинхронно. Эти прерывания информируют систему о событиях, связанных с работой устройств, например о том, что наконец-то завершилась печать символа на принтере и неплохо было бы выдать следующий символ, или о том, что требуемый сектор диска уже прочитан, его содержимое доступно программе. Использование прерываний при работе с медленными внешними устройствами позволяет совместить ввод-вывод с обработкой данных в центральном процессоре и в результате повышает общую производительность системы. Некоторые прерывания (первые пять в порядке номеров) зарезервированы для использования самим центральным процессором на случай каких-либо особых событий вроде

попытки деления на ноль, переполнения и т.п. Иногда желательно сделать систему нечувствительной ко всем или отдельным прерываниям. Для этого используют так называемое маскирование прерываний. Но некоторые прерывания замаскировать нельзя, это немаскируемые прерывания.

Заметим еще, что обработчики прерываний могут сами вызывать программные прерывания, например для получения доступа к сервису BIOS или DOS (сервис BIOS также доступен через механизм программных прерываний).

Составление собственных программ обработки прерываний и замена стандартных обработчиков DOS и BIOS — ответственная и сложная работа. Необходимо учитывать все тонкости работы аппаратуры и взаимодействия программного и аппаратного обеспечения. При отладке возможно разрушение операционной системы с непредсказуемыми последствиями, поэтому надо очень внимательно следить за тем, что делает ваша программа.

Для того чтобы связать адрес обработчика прерывания с номером прерывания, используется таблица векторов прерываний, занимающая первый килобайт оперативной памяти — адреса от 0000:0000 до 0000:03FF. Таблица состоит из 256 элементов — FAR-адресов обработчиков прерываний. Эти элементы называются *векторами прерываний*. В первом слове элемента таблицы записано смещение, а во втором — адрес сегмента обработчика прерывания.

Прерыванию с номером 0 соответствует адрес 0000:0000, прерыванию с номером 1—0000:0004 и т.д.

Инициализация таблицы происходит частично BIOS после тестирования аппаратуры и перед началом загрузки операционной системой, частично при загрузке DOS. DOS может переключить на себя некоторые прерывания BIOS.

Рассмотрим содержимое таблицы векторов прерываний (табл. 4.4). Приведем назначение некоторых наиболее важных векторов.

Аппаратные прерывания вырабатываются устройствами компьютера, когда возникает необходимость их обслуживания. Например, по прерыванию таймера соответствующий обработчик прерывания увеличивает содержимое ячеек памяти, используемых для хранения времени. В отличие от программных прерываний, вызываемых запланированно самой прикладной программой, аппаратные прерывания всегда происходят асинхронно по отношению к выполняющимся программам. Кроме того, может возникнуть одновременно сразу несколько прерываний.

Таблица 4.4

Таблица векторов прерываний

Номер	Описание
0	Ошибка деления. Вызывается автоматически после выполнения команд DIV или IDIV, если в результате деления происходит переполнение (например, при делении на ноль). DOS обычно при обработке этого прерывания выводит сообщение об ошибке и останавливает выполнение программы. Для процессора 8086 при этом адрес возврата указывает на следующую после команды деления команду, а в процессоре 80286 — на первый байт команды, вызвавшей прерывание
1	Прерывание пошагового режима. Вырабатывается после выполнения каждой машинной команды, если в слове флагов установлен бит пошаговой трассировки TF. Используется для отладки программ. Это прерывание не вырабатывается после выполнения команды MOV в сегментные регистры или после загрузки сегментных регистров командой POP
2	Аппаратное немаскируемое прерывание. Это прерывание может использоваться по-разному в разных машинах. Обычно вырабатывается при ошибке четности в оперативной памяти и при запросе прерывания от сопроцессора
3	Прерывание для трассировки. Это прерывание генерируется при выполнении однобайтовой машинной команды с кодом CCh и обычно используется отладчиками для установки точки прерывания
4	Переполнение. Генерируется машинной командой INTO, если установлен флаг OF. Если флаг не установлен, то команда INTO выполняется как NOP. Это прерывание используется для обработки ошибок при выполнении арифметических операций
5	Печать копии экрана. Генерируется при нажатии на клавиатуре клавиши PrtScr. Обычно используется для печати образа экрана. Для процессора 80286 генерируется при выполнении машинной команды BOUND, если проверяемое значение вышло за пределы заданного диапазона
6	Неопределенный код операции или длина команды больше 10 байт (для процессора 80286)
7	Особый случай отсутствия математического сопроцессора (процессор 80286)
8	IRQ0 — прерывание интервального таймера, возникает 18,2 раза в секунду
9	IRQ1 — прерывание от клавиатуры. Генерируется при нажатии и при отжатии клавиши. Используется для чтения данных от клавиатуры

Продолжение табл. 4.4

Номер	Описание
A	IRQ2 — используется для каскадирования аппаратных прерываний в машинах класса AT
B	IRQ3 — прерывание асинхронного порта COM2
C	IRQ4 — прерывание асинхронного порта COM1
D	IRQ5 — прерывание от контроллера жесткого диска для XT
E	IRQ6 — прерывание генерируется контроллером флоппи-диска после завершения операции
F	IRQ7 — прерывание принтера. Генерируется принтером, когда он готов к выполнению очередной операции. Многие адаптеры принтера не используют это прерывание
10	Обслуживание видеоадаптера
11	Определение конфигурации устройств в системе
12	Определение размера оперативной памяти в системе
13	Обслуживание дисковой системы
14	Последовательный ввод/вывод
15	Расширенный сервис для AT-компьютеров
16	Обслуживание клавиатуры
17	Обслуживание принтера
18	Запуск BASIC в ПЗУ, если он есть
19	Загрузка операционной системы
1A	Обслуживание часов
1B	Обработчик прерывания Ctrl-Break
1C	Прерывание возникает 18,2 раза в секунду, вызывается программно обработчиком прерывания таймера
1D	Адрес видеотаблицы для контроллера видеоадаптера 6845
1E	Указатель на таблицу параметров дискеты
1F	Указатель на графическую таблицу для символов с кодами ASCII 128—255
20—5F	Используется DOS или зарезервировано для DOS
60—67	Прерывания, зарезервированные для пользователя
68—6F	Не используются
70	IRQ8 — прерывание от часов реального времени
71	IRQ9 — прерывание от контроллера EGA
72	IRQ10 — зарезервировано
73	IRQ11 — зарезервировано
74	IRQ12 — зарезервировано
75	IRQ13 — прерывание от математического сопроцессора

Окончание табл. 4.4

Номер	Описание
76	IRQ14 — прерывание от контроллера жесткого диска
77	IRQ15 — зарезервировано
78—7F	Не используются
80—85	Зарезервированы для BASIC
86-F0	Используются интерпретатором BASIC
F1-FF	Не используются

IRQ0—IRQ15 — аппаратные прерывания.

Для того чтобы система «не растерялась», решая, какое прерывание обслуживать в первую очередь, существует специальная схема приоритетов. Каждому прерыванию назначается свой уникальный приоритет. Если происходит одновременно несколько прерываний, то система отдает предпочтение самому высокоприоритетному, откладывая на время обработку остальных прерываний.

Уровни приоритетов обозначаются сокращенно IRQ0 — IRQ15 (для машин класса XT существуют только уровни IRQ0 — IRQ7).

Для машин XT приоритеты линейно зависели от номера уровня прерывания. IRQ0 соответствовало самому высокому приоритету, за ним шли IRQ1, IRQ2, IRQ3 и т.д. Уровень IRQ2 в машинах класса XT был зарезервирован для дальнейшего расширения системы и, начиная с машин класса AT IRQ2, стал использоваться для каскадирования контроллеров прерывания 8259. Добавленные приоритетные уровни IRQ8—IRQ15 в этих машинах располагаются по приоритету между IRQ1 и IRQ3 (табл. 4.5).

Из табл. 4.5 видно, что самый высокий приоритет — у прерываний от интервального таймера, затем идет прерывание от клавиатуры.

Для управления схемами приоритетов необходимо знать внутреннее устройство контроллера прерываний 8259. Поступающие прерывания запоминаются в регистре запроса на прерывание IRR. Каждый бит из восьми в этом регистре соответствует прерыванию. После проверки на обработку в настоящий момент другого прерывания запрашивается информация из регистра обслуживания ISR. Перед выдачей запроса на прерывание в процессор проверяется содержимое 8-битового регистра маски прерываний IMR. Если прерывание данного уровня не замаскировано, то выдается запрос на прерывание.

Таблица 4.5

Аппаратные прерывания, расположенные в порядке приоритета

Номер	Описание
8	IRQ0 — прерывание интервального таймера, возникает 18,2 раза в секунду
9	IRQ1 — прерывание от клавиатуры. Генерируется при нажатии и при отжатии клавиши. Используется для чтения данных с клавиатуры
A	IRQ2 — используется для каскадирования аппаратных прерываний в машинах класса AT
70	IRQ8 — прерывание от часов реального времени
71	IRQ9 — прерывание от контроллера EGA
72	IRQ10 — зарезервировано
73	IRQ11 — зарезервировано
74	IRQ12 — зарезервировано
75	IRQ13 — прерывание от математического сопроцессора
76	IRQ14 — прерывание от контроллера жесткого диска
77	IRQ15 — зарезервировано
B	IRQ3 — прерывание асинхронного порта COM2
C	IRQ4 — прерывание асинхронного порта COM1
D	IRQ5 — прерывание от контроллера жесткого диска для XT
E	IRQ6 — прерывание генерируется контроллером флоппи диска после завершения операции
F	IRQ7 — прерывание принтера. Генерируется принтером, когда он готов к выполнению очередной операции. Многие адаптеры принтера не используют это прерывание

Наиболее интересными с точки зрения программирования контроллера прерываний являются регистры маски прерываний IMR и управляющий регистр прерываний.

В машинах класса XT регистр маски прерываний имеет адрес 21h, управляющий регистр прерываний — 20h. Для машин AT первый контроллер 8259 имеет такие же адреса, что и в машинах XT, регистр маски прерываний второго контроллера имеет адрес A1h, управляющий регистр прерываний — A0h.

Разряды регистра маски прерываний соответствуют номерам IRQ. Для того чтобы замаскировать аппаратное прерывание какого-либо уровня, надо заслать в регистр маски байт, в котором бит, соответствующий этому уровню, установлен в 1. Например, для маскирования прерываний от НГМД в порт 21h надо заслать двоичное число 01000000.

Программируемый контроллер прерываний 8259 предназначен для обработки до восьми приоритетных уровней прерываний. Возможно каскадирование микросхем, при этом общее число уровней прерываний будет достигать 64.

Контроллер 8259 имеет несколько режимов работы, которые устанавливаются программным путем. В персональных компьютерах XT и AT за первоначальную установку режимов работы микросхем 8259 отвечает BIOS.

Каждому приоритетному уровню прерывания микросхема ставит в соответствие определенный, задаваемый программно номер прерывания. В разделе книги, посвященном особенностям обработки аппаратных прерываний, приводится такое соответствие для машин типа XT и AT.

Для обработки прерываний контроллер имеет несколько внутренних регистров. Это регистр запросов прерываний IRR, регистр обслуживания прерываний ISR, регистр маски прерываний IMR. В регистре IRR хранятся запросы на обслуживание прерываний от аппаратуры. После выработки сигнала прерывания центральному процессору соответствующий разряд регистра ISR устанавливается в единичное состояние, что блокирует обслуживание всех запросов с равным или более низким приоритетом. Устранить эту блокировку можно либо сбросом соответствующего бита в ISR, либо командой специального маскирования.

Имеется два типа команд, посылаемых программой в контроллер 8259: команды инициализации и команды операции. Возможны следующие операции:

- индивидуальное маскирование запросов прерывания;
- специальное маскирование обслуженных запросов;
- установка статуса уровней приоритета (по установке исходного состояния, по обслуженному запросу, по указанию);
- операции конца прерывания (обычный конец прерывания, специальный конец прерывания, автоматический конец прерывания);
- чтение регистров IRR, ISR, IMR.

Организация прерываний в процессоре Intel 80X86

Механизм асинхронного взаимодействия процессов, одновременно выполняемых в вычислительной системе, и — еще один способ вызова подпрограмм. Весь программный интерфейс с операционной системой реализуется именно на основе прерываний.

Прерывания	
1. Программные: • исключения • прерывания от внешних устройств	2. Аппаратные: • прерывания от внешних устройств

Исключения возникают при выполнении некой команды в процессоре; они подразделяются на ошибки, ловушки и остановы.

Ошибка появляется до выполнения команды (например, если такой команды не существует или происходит обращение к некоторой привилегированной функции или области данных). Адрес возврата в этом случае указывает на ошибочную команду.

Ловушка — прерывание, возникающее после выполнения команды (например, для организации пошагового выполнения команд). Адрес возврата указывает на следующую команду.

Останов — ситуация с неопределенным результатом. Возврат может вообще не происходить.

Большинство программных прерываний типа INTn являются ловушками.

Аппаратные прерывания — прерывания, возникающие асинхронно от внешних устройств, которые обслуживаются в зависимости от приоритета и обрабатываются контроллером прерывания (микросхема 8259).

Прерывания низкого уровня. Диапазон значений номеров вектора прерывания NВП=0..1Fh.

1. Прерывания от схем процессора 00–07 (0 — деление на ноль; 1 — пошаговый режим; 2 — немаскируемое прерывание; 3 — точка останова; 4 — прерывание по переполнению; 5 — печать содержимого экрана либо прерывание по команде BAUND; 6 — прерывание по отсутствию команды; 7 — прерывание по отсутствию FPU).

2. Прерывания от контроллера 8259 8–0Fh (8 — системный таймер; 9 — клавиатура; 0Ah — обслуживание видеоадаптера; 0Eh — обслуживание жесткого диска).

3. Прерывания BIOS 0Ah–1Fh (0Ah — обмен данными с дисплеем; 0Bh — возвращение объема памяти; 0Ch — обмен данными с диском; 0Dh — последовательный порт ввода-вывода и т.д.).

Прерывания среднего уровня. $N_{ВП}$=20h..5Fh (21h — большая часть работы с файлами, управление задачами, выделение и освобождение памяти, работа с виртуальной памятью).

Прерывания пользователей. $N_{ВП}$=60h..7Fh (от 70h используются для различных устройств).

Прерывания языков высокого уровня. $N_{ВП}$=80h... и более.

Уровни приоритета. IRQ0—IRQ15 — контролируются и выполняются с помощью контроллера 8259. Самый высокий уровень приоритета у IRQ0.

IRQ0 — системный таймер;
IRQ1 — клавиатура;
IRQ2 — размножитель приоритетных уровней;
8 — часы реального времени;
9 — прерывание обратного хода луча и звуковой карты;
10—12 — резерв;
13 — ошибка FPU;
14 и 15 — от контроллеров жесткого диска IDE1, IDE2;
IRQ3 — COM2 (INT 0Bh);
IRQ4 — COM1 (INT 0Ch);
IRQ5 — не используется (INT 0Dh);
IRQ6 — прерывание от магнитного диска (INT 0Eh);
IRQ7 — LPT1 (INT 0Fh).

Маскируемые прерывания. Запрет от прерывания выполнения критичной части программы или для маскирования долго выполняемых прерываний.

Способы реализации:

1. Общее маскирование.

CLI, STI — устанавливают флаг прерывания IF в 0 и единицу соответственно.

2. Выборочное маскирование.

Засылка определенного кода в регистр маски контроллера 8259. В этом контроллере три основных регистра:

- IRR — регистр запроса прерывания;
- ISR — регистр обслуживания прерывания (порт 20h);
- IMR — регистр маскирования прерывания (порт 21h).

Например,

```
moval, 01000000b; маскируются запросы прерывания от жесткого диска
out 21h
------
moval, 0
out 21h
```

К IRR подключены все линии IRQ0—IRQ15. ISR хранит приоритет текущего обслуживаемого процесса. Происходит сравнение PR_{IRR} и PR_{ISR}, запрос IRR не должен быть маскирован.

Разработка собственных прерываний

Причины: необходимость создания собственной подпрограммы резидентной в памяти и доступной из любой программы; необходимость дополнения существующих прерываний; использование холостых прерываний, телом которых является IRET.

Существует два способа заполнения вектора прерывания адресом своего обработчика:

1) низкоуровневый: командой mov записать по адресу вектора сегмента и смещения обработчика;

2) используя средства операционной системы: функции 25 и 35 int 21h позволяют устанавливать новое и получать старое значение адреса обработчика вектора прерывания.

Пример 4.2. N=60

```
Data Segment
        old_csdw 0; буфер для хранения
        old_ip dw 0; старый ВП
------
        old_vp dd 0;
Data ENDS
Code Segment
; сохранение адреса старого обработчика
        mov ax, 3560h
        int 21; ВП_cs→es, ВП_ir→bx
        mov old_cs, es
        mov old_ip, bx
; задание адреса нового обработчика в ВП 60h
        push ds
        mov dx, offset    New_sub
        mov ax, seg       New_sub
        mov ds, ax
        mov ax, 2560h
        int 21h
        pop ds
; новый обработчик прерывания 60h
New_sub proc far
        push ax
; тело обработчика
        pop ax
        mov al, 20
        out 20h
        iret
```

———-
Final EQU $
———-
New_sub ENDP
; восстановление старого ВП
lds dx, DWORD PTR OLD_CS
mov ax, 2560h
int 21h

Возможные проблемы:

- если данные передаются через память, нужно тщательно следить за содержимым регистра ds. Лучше данные передавать через регистры или стек;
- если возможно прерывание обработчика через Ctrl+Break, необходимо предусмотреть восстановление адреса старого обработчика;
- требуется минимизировать код обработчика прерывания, так как на время его выполнения может быть запрещено выполнение других прерываний.

Перекрытие обработчика прерываний

Можно записать существующий обработчик по новому адресу, а на старый номер — свой, который может вызывать старый. Существующие обработчики прерываний DOS и BIOS сложно поддаются модификации на уровне исходных кодов, и для добавления новых функций требуется реализовать следующий механизм.

1. Создать новый обработчик прерываний (реализующий дополнительные функции), который вызывает старый (системный), размещенный по новому неиспользованному вектору в диапазоне 60h—70h.

2. Перенести старый обработчик прерываний в новый вектор прерывания.

3. Изменить вектор прерывания с системным номером таким образом, чтобы он указывал на новый обработчик прерывания.

4. Завершить программу установки нового обработчика и оставить ее резидентной в памяти.

Возврат после завершения старого обработчика может происходить либо в новый обработчик:

push f

call old_handler; дополнительные функции после старого обработчика

→

либо в вызывающую процедуру:

jmp cs: old_handler; дополнительные функции до старого обработчика

TSR (TerminateandStayResident):
int 27h — более старая версия;
int 21h — более новая.

Резидентный обработчик обычно пишется в виде модуля типа com, и для его разработки необходимо:

- наличие свободного вектора прерывания и указание метки конца обработчика для int 27h (определение длины обработчика +100PSP);
- для int 31h длина обработчика задается параграфами.

Для минимизации длины кода обработчика инициализирующую его часть выносят за его предел:

```
Cod_s Segment
Begin: jmp short set_up; переход на инициализацию
Rezid_h: proc far
        push ds
; тело процедуры обработчика прерывания
        pop ds
        iret
Final EQU $;            текущее значение счетчика размещения
Rezid_h ENDP
Set_up: mov dx, offset Rezid_h
        mov ax, 2568h; задание свободного ВП с № 68h
        int 21h
; завершение с оставлением в памяти
        lea dx, Final
        int 27h
        ret
Cod_s ENDS
```

Первый способ несовместим с резидентными программами (так как одни и те же прерывания используются для эмулятора и резидентных программ). Поэтому для таких программ надо использовать третий способ. Второй способ — самый быстрый.

Контрольные вопросы и задания

1. Опишите типовую структуру процессора и основной памяти.
2. Опишите основной цикл работы процессора.
3. Опишите процедуру организации прерываний в процессоре Intel 80X86.
4. Определите физический адрес памяти, если логические адреса в форме сегмент:смешения равны 31A0:4C6716 и 31A1:4C5716. Чем отличаются логические адреса между собой?

5. Уровень архитектуры набора команд. Отличие от уровня микроархитектуры.
6. Трансляция. Виды трансляции.
7. Какие типы данных вы знаете?
8. Структуры команд. Понятия «адресный код в команде AR» и «исполнительный (физический адрес) адрес АИ».
9. Какие типы прерываний вы знаете?
10. Маскируемые прерывания. Способы реализации.
11. Что произойдет с системой, если одновременно поступили два прерывания?
12. Резидентный обработчик прерываний. Требования для его разработки.

Глава 5
УРОВЕНЬ ОПЕРАЦИОННОЙ СИСТЕМЫ

Рассмотрим уровень операционной системы. В логической структуре типичной вычислительной системы операционная система занимает положение между устройствами с их микроархитектурой, машинным языком и, возможно, собственными (встроенными) микропрограммами (драйверами), с одной стороны, и прикладными программами — с другой.

Разработчикам программного обеспечения операционная система позволяет абстрагироваться от деталей реализации и функционирования устройств, предоставляя минимально необходимый набор функций.

5.1. Назначение, структура и функции операционной системы

Операционная система — это комплекс взаимосвязанных программ, кода и графического интерфейса, предназначенных для управления ресурсами компьютера и взаимодействия машины с пользователем. Любая операционная система представляет собой разделительный барьер, посредника между разработчиками программного обеспечения и микроархитектуры.

В большинстве вычислительных систем операционная система является основной, наиболее важной (а иногда и единственной) частью системного программного обеспечения. С 1990-х гг. наиболее распространенными операционными системами являются системы семейства Windows, UNIXиUNIX-подобные системы.

Хотя и уровень операционной системы, и уровень архитектуры команд абстрактны (в том смысле, что не являются реальными устройствами), между ними есть важное различие. Все команды уровня операционной системы доступны для прикладных программистов. Это практически все команды более низкого уровня, а также новые команды, добавленные операционной системой. Новые команды называются системными вызовами. Они вызывают предопределенную службу операционной системы, в частности одну из ее команд. Обычный системный вызов считывает какие-нибудь данные из файла. Уровень операционной системы всегда интерпретируется. Когда пользовательская программа вызывает команду операционной

системы, например чтение данных из файла, операционная система выполняет эту команду шаг за шагом точно так же, как микропрограмма выполняет команду ADD. Однако, когда программа вызывает команду уровня архитектуры команд, эта команда выполняется непосредственно уровнем микроархитектуры без участия операционной системы.

С точки зрения программиста, операционная система — это программа, добавляющая ряд команд и функций к командам и функциям, предлагаемым уровнем архитектуры команд [1].

Операционную систему определяют также как «набор программ, управляющих оборудованием», и как «набор программ, управляющих другими программами». Данные определения имеют свой точный технический смысл, который связан с вопросом, в каких случаях требуется операционная система.

Структура операционных систем

Все компоненты операционной системы можно разделить на две группы — работающие в привилегированном режиме и работающие в пользовательском режиме, причем состав этих групп меняется от системы к системе.

Современные процессоры имеют минимум два режима работы — *привилегированный* (supervisor mode) и *пользовательский* (user mode).

Отличие между ними заключается в том, что в пользовательском режиме недоступны команды процессора, связанные с управлением аппаратным обеспечением, защитой оперативной памяти, переключением режимов работы процессора. В привилегированном режиме процессор может выполнять все возможные команды.

Приложения, выполняемые в пользовательском режиме, не могут напрямую обращаться к адресным пространствам друг друга — только посредством системных вызовов.

Основным компонентом операционной системы является *ядро* (kernel). Функции ядра могут существенно отличаться в разных системах, но во всех системах ядро работает в привилегированном режиме (который часто называется режим ядра, kernel mode).

Термин «ядро» также используется в разных смыслах. Например, в Windows термин «ядро» (NTOS kernel) обозначает совокупность двух компонентов — исполнительной системы (executive layer) и собственно ядра (kernel layer).

Существует два основных вида ядер — монолитные ядра (monolithic kernel) и микроядра (microkernel). В *монолитном ядре* реализуются все основные функции операционной системы, и оно является,

по сути, единой программой, представляющей собой совокупность процедур [5]. В *микроядре* остается лишь минимум функций, который должен быть реализован в привилегированном режиме: планирование потоков, обработка прерываний, межпроцессное взаимодействие. Остальные функции операционной системы по управлению приложениями, памятью, безопасностью и пр. реализуются в виде отдельных модулей в пользовательском режиме.

Ядра, которые занимают промежуточное положение между монолитными и микроядрами, называют *гибридными* (hybrid kernel).

Кроме ядра, в привилегированном режиме (в большинстве операционных систем) работают *драйверы* (driver) — программные модули, управляющие устройствами.

В состав операционной системы также входят:

- системные библиотеки (system DLL — Dynamic Link Library, динамически подключаемая библиотека), преобразующие системные вызовы приложений в системные вызовы ядра;
- пользовательские оболочки (shell), предоставляющие пользователю интерфейс — удобный способ работы с операционной системой.

Пользовательские оболочки реализуют один из двух основных видов пользовательского интерфейса:

- текстовый интерфейс (Text User Interface, TUI), другие названия — консольный интерфейс (Console User Interface, CUI), интерфейс командной строки (Command Line Interface, CLI);
- графический интерфейс (Graphic User Interface, GUI).

Пример реализации текстового интерфейса в Windows — интерпретатор командной строки cmd.exe; пример графического интерфейса — Проводник Windows (explorer.exe).

Функции операционной системы

В зависимости от назначения операционную систему можно представить следующим образом.

1. **Операционная система как виртуальная машина.** Операционная система предоставляет пользователю виртуальную машину, которую легче программировать и с которой легче работать, чем непосредственно с аппаратурой, составляющей реальную машину.

Использование вычислительных машин на уровне машинного языка затруднительно, особенно это касается ввода-вывода. Например, чтобы считать или записать информацию на дискету, надо:

- запустить двигатель вращения дискеты;
- управлять шаговым двигателем перемещения головки;

- следить за индикатором присутствия дискеты;
- выбрать номер блока на диске;
- выбрать дорожку;
- выбрать номер сектора на дорожке и т.д.

Все эти функции берет на себя операционная система. Операционная система предоставляет простой файловый интерфейс, освобождает программиста от аппаратных проблем, связанных с организацией ввода-вывода, обработкой прерываний, управлением таймерами и оперативной памятью, а также других низкоуровневых проблем.

2. **Операционная система как система управления ресурсами.** В соответствии со вторым подходом функцией ОС является распределение процессоров, памяти, устройств и данных между процессами, конкурирующими за эти ресурсы. ОС должна управлять всеми ресурсами вычислительной машины таким образом, чтобы обеспечить максимальную эффективность ее функционирования. Критерием эффективности может быть, например, пропускная способность или реактивность системы. Управление ресурсами включает решение двух общих, не зависящих от типа ресурса задач:

- планирование ресурса, т.е. определение, кому, когда, а для делимых ресурсов и в каком количестве необходимо выделить данный ресурс;
- отслеживание состояния ресурса, т.е. поддержание оперативной информации о том, занят или не занят ресурс, а для делимых ресурсов — какое количество ресурса уже распределено, а какое свободно.

Для решения этих общих задач управления ресурсами разные ОС используют различные алгоритмы, что в конечном счете и определяет их облик в целом, включая характеристики производительности, область применения и даже пользовательский интерфейс. Так, например, алгоритм управления процессором в значительной степени определяет, является ли ОС системой разделения времени, системой пакетной обработки или системой реального времени.

3. **Операционная система как API** (Application Programming Interface) — интерфейс прикладного программирования.Интерфейс между операционной системой и программами определяется набором системных вызовов. Например, если пользовательскому процессу необходимо считать данные из файла, он должен выполнить команду системного вызова, т.е. выполнить прерывание с переключением в режим ядра и активизировать функцию операционной системы для считывания данных из файла.

Функционирование компьютера после включения питания

Функционирование компьютера после включения питания начинается с запуска программы первоначальной загрузки — Boot Track. Программа Boot Track инициализирует основные аппаратные блоки компьютера и регистры процессора (CPU), накопитель памяти, контроллеры периферийного оборудования. Затем загружается ядро ОС, т.е. Operating System Kernel. Дальнейшее функционирование ОС осуществляется как реакция на события, происходящие в компьютере. Наступление того или иного события сигнализируется прерываниями — Interrupt. Источниками прерываний могут быть как аппаратура (HardWare), так и программы (SoftWare).

Аппаратура «сообщает» о прерывании асинхронно (в любой момент времени) путем пересылки в CPU через общую шину сигналов прерываний. Программа «сообщает» о прерывании путем выполнения операции System Call. Примеры событий, вызывающих прерывания:

- попытка деления на 0;
- запрос на системное обслуживание;
- завершение операции ввода-вывода;
- неправильное обращение к памяти.

Каждое прерывание обрабатывается соответственно обработчиком прерываний (Interrupt handler), входящим в состав ОС.

Главные функции механизма прерываний:

- распознавание или классификация прерываний;
- передача управления соответственно обработчику прерываний;
- корректное возвращение к прерванной программе.

Переход от прерываемой программы к обработчику и обратно должен выполняться как можно быстрей. Одним из быстрых методов является использование таблицы, содержащей перечень всех допустимых для компьютера прерываний и адреса соответствующих обработчиков. Такая таблица называется вектором прерываний (Interrupt vector) и хранится в начале адресного пространства основной памяти (UNIX/MS DOS).

Для корректного возвращения к прерванной программе перед передачей управления обработчику прерываний содержимое регистров процессора запоминается либо в памяти с прямым доступом, либо в системном стеке — System Stack.

Обычно запрещаются прерывания обработчика прерываний. Однако в некоторых ОС прерывания снабжаются приоритетами, т.е.

работа обработчика прерывания с более низким приоритетом может быть прервана, если произошло прерывание с более высоким приоритетом.

Немного истории

Предшественником операционных систем следует считать служебные программы (загрузчики и мониторы), а также библиотеки часто используемых подпрограмм, начавшие разрабатываться с появлением универсальных компьютеров 1-го поколения (конец 1940-х гг.). Служебные программы минимизировали физические манипуляции оператора с оборудованием, а библиотеки позволяли избежать многократного программирования одних и тех же действий (осуществления операций ввода-вывода, вычисления математических функций и т.п.).

В 1950–1960-х гг. сформировались и были реализованы основные идеи, определяющие функциональность ОС: пакетный режим, разделение времени и многозадачность, разделение полномочий, реальный масштаб времени, файловые структуры и файловые системы.

Пакетный режим. Необходимость оптимального использования дорогостоящих вычислительных ресурсов привела к появлению концепции «пакетного режима» исполнения программ. Пакетный режим предполагает наличие очереди программ на исполнение, причем система может обеспечивать загрузку программы с внешних носителей данных в оперативную память, не дожидаясь завершения исполнения предыдущей программы, что позволяет избежать простоя процессора.

Разделение времени и многозадачность. Уже пакетный режим в своем развитом варианте требует разделения процессорного времени между выполнением нескольких программ.

Необходимость в разделении времени (многозадачности, мультипрограммировании) проявилась еще сильнее при распространении в качестве устройств ввода-вывода телетайпов (а позднее — терминалов с электронно-лучевыми дисплеями) (1960-е гг.). Поскольку скорость клавиатурного ввода (и даже чтения с экрана) данных оператором много ниже, чем скорость обработки этих данных компьютером, использование компьютера в «монопольном» режиме (с одним оператором) могло привести к простою дорогостоящих вычислительных ресурсов.

Разделение времени позволило создать «многопользовательские» системы, в которых один (как правило) центральный процессор

и блок оперативной памяти соединялся с многочисленными терминалами. При этом часть задач (таких как ввод или редактирование данных оператором) могла исполняться в режиме диалога, а другие задачи (такие как массивные вычисления) — в пакетном режиме.

Разделение полномочий. Распространение многопользовательских систем потребовало решения задачи разделения полномочий, позволяющей избежать возможности изменения исполняемой программы или данных одной программы в памяти компьютера другой программой (намеренно или по ошибке), а также изменения самой системы прикладной программой.

Реализация разделения полномочий в операционных системах была поддержана разработчиками процессоров, предложивших архитектуры с двумя режимами работы процессора — «реальным» (в котором исполняемой программе доступно все адресное пространство компьютера) и «защищенным» (в котором доступность адресного пространства ограничена диапазоном, выделенным при запуске программы на исполнение).

Масштаб реального времени. Применение универсальных компьютеров для управления производственными процессами потребовало реализации «масштаба реального времени» («реального времени») — синхронизации исполнения программ с внешними физическими процессами.

Включение функции масштаба реального времени позволило создавать решения, одновременно обслуживающие производственные процессы и решающие другие задачи (в пакетном режиме и/или в режиме разделения времени).

Файловые системы и структуры. Постепенная замена носителей с последовательным доступом (перфолент, перфокарт и магнитных лент) накопителями произвольного доступа (на магнитных дисках).

Файловая система — способ хранения данных на внешних запоминающих устройствах.

5.2. Операционная система как система управления ресурсами

Понятия «процессы», «потоки» и «файберы»

Не следует смешивать понятия «процесс» и «программа». Программа — это план действий, а процесс — это само действие.

Например, когда вы открываете приложение MS Word, то запускаете процесс, исполняющий программу MS Word. Поэтому можно

сказать, что **процесс** — это выполнение вычислительной системой некоторой системной или прикладной программы или их фрагмента.

Каждому процессу в операционной системе соответствует контекст процесса. Этот контекст включает в себя:

- пользовательский контекст (соответствующий программный код, данные, размер виртуальной памяти, дескрипторы открытых файлов и пр.);
- аппаратный контекст (содержимое регистра счетчика команд, регистра состояния процессора, регистр указателя стека, а также содержимое регистров общего назначения);
- системный контекст (состояние процесса, идентификатор соответствующего пользователя, идентификатор процесса и пр.).

Важно, что из-за большого объема данных контекста процесса переключение процессора системы с выполнения одного процесса на выполнение другого процесса (смена контекста процесса) является относительно дорогостоящей операцией.

Для уменьшения времени смены контекста процесса в современных ОС (например, в ОС UNIX) наряду с понятием процесса широко используется понятие легковесного процесса «light-weight process» или понятие потока, нити «thread». Легковесный процесс можно определить как подпроцесс некоторого процесса, выполняемый в контексте этого процесса. Контекст процесса содержит общую для всех его легковесных процессов информацию — виртуальную память, дескрипторы открытых файлов и т.д. Остальная информация из контекста процесса переходит в контексты его легковесных процессов.

Простейшим процессом является процесс, состоящий из одного легковесного процесса.

Поток — отдельное исполняемое задание внутри процесса. Процесс может содержать множество исполняемых потоков. После запуска приложения исполняется главный поток, который далее может порождать другие потоки.

Принципиальным является то обстоятельство, что нити одного процесса выполняются в общей виртуальной памяти, т.е. имеют равные права доступа к любым частям виртуальной памяти процесса. Операционной системой основной ресурс вычислительной системы — процессорное время — выделяется не процессу, а легковесному процессу.

На основе сказанного процесс можно определить как некоторый контекст, включающий виртуальную память и другие системные ре-

сурсы, в котором выполняется по крайней мере один легковесный процесс, обладающий своим собственным (более простым) контекстом. ОС «знает» о существовании двух указанных уровней контекстов и способна сравнительно быстро изменять контекст легковесного процесса, не изменяя общего контекста процесса. Заметим, что для синхронизации легковесных процессов, работающих в общем контексте процесса, можно использовать более дешевые средства, чем для синхронизации процессов.

Понятие легковесного процесса направлено на организацию вычислений в многопроцессорной вычислительной системе, в случае когда приложение, выполняемое в рамках одного процесса, обладает внутренним параллелизмом. Разумеется, параллельное выполнение приложения можно организовать и на пользовательском уровне — путем создания для одного приложения нескольких процессов для каждой из параллельных работ. Однако при этом не учитывается тот факт, что эти процессы решают общую задачу, а значит, могут иметь много общего — общие данные, программные коды, права доступа к ресурсам системы и пр. Кроме того, как отмечалось выше, каждый процесс требует значительных системных ресурсов, которые при такой организации параллельных вычислений неоправданно дублируются.

На одном процессоре многопоточность обычно происходит путем временного мультиплексирования (как и в случае многозадачности): процессор переключается между разными потоками выполнения. Это переключение контекста обычно происходит достаточно часто, чтобы пользователь воспринимал выполнение потоков или задач как одновременное. В многопроцессорных и многоядерных системах потоки или задачи могут реально выполняться одновременно, при этом каждый процессор или ядро обрабатывает отдельный поток или задачу.

Многие современные операционные системы поддерживают как временные нарезки от планировщика процессов, так и многопроцессорные потоки выполнения. Ядро операционной системы позволяет программистам управлять потоками выполнения через интерфейс системных вызовов. Некоторые реализации ядра называют потоком ядра, другие же — легковесным процессом (англ. light-weight process, LWP), представляющим собой особый тип потока выполнения ядра, который совместно использует одни и те же состояния и данные.

Файберы являются еще более «легкими» блоками планирования, относящимися к кооперативной многозадачности: выполняющийся файбер должен явно «уступить» право другим файберам на выполнение, что делает их реализацию гораздо легче, чем реализацию потоков выполнения ядра или пользовательских потоков выполнения. Файберы могут быть запланированы для запуска в любом потоке выполнения внутри того же процесса. Это позволяет приложениям получить повышение производительности за счет управления планированием самого себя, вместо того чтобы полагаться на планировщика ядра, который может быть не настроен на такое применение. Параллельные среды программирования, такие как OpenMP, обычно реализуют свои задачи посредством файберов.

Системный вызов (англ. system call) в программировании и вычислительной технике — обращение прикладной программы к ядру операционной системы для выполнения какой-либо операции.

Современные ОС предусматривают разделение времени между выполняющимися вычислительными процессами (многозадачность) и разделение полномочий, препятствующее обращению исполняемых программ к данным других программ и оборудованию. Ядро ОС исполняется в привилегированном режиме работы процессора. Для выполнения межпроцессной операции или операции, требующей доступа к оборудованию, программа обращается к ядру, которое, в зависимости от полномочий вызывающего процесса, исполняет либо отказывает в исполнении такого вызова.

С точки зрения программиста, системный вызов обычно выглядит как вызов подпрограммы или функции из системной библиотеки. Однако системный вызов как частный случай вызова такой функции или подпрограммы следует отличать от более общего обращения к системной библиотеке, поскольку последнее может и не требовать выполнения привилегированных операций.

Большинство современных операционных систем позволяют формировать и прерывать процессы динамически. Для формирования нового процесса требуется системный вызов. Этот системный вызов может просто создать клон вызывающей программы или позволить исходному процессу задать начальное состояние нового процесса, включая его программу, данные и начальный адрес. В одних случаях исходный (родительский) процесс может сохранять частичный или даже полный контроль над порожденным (дочерним) процессом. Виртуальные команды позволяют родительскому процессу

останавливать и снова запускать, проверять и завершать дочерние процессы. В других случаях исходный процесс никак не контролирует дочерний процесс: после того как новый процесс сформирован, исходный процесс не может его остановить, запустить заново, проверить или завершить. Таким образом, эти два процесса работают независимо друг от друга.

Управление процессами

Операционная система контролирует следующую деятельность, связанную с процессами:

1. Создание и удаление процессов.
2. Планирование процессов.
3. Синхронизация процессов.
4. Коммуникация процессов.
5. Разрешение тупиковых ситуаций.

В одиночном режиме в системе существует только один прикладной процесс. Для выполнения необходимо минимум информации.

В мультипрограммном режиме ОС должна иметь информацию о каждом процессе. Создаются управляющие структуры — Control Block — таблицы, содержащие информацию, необходимую операционной системе для управления вычислительным процессом.

Таким образом, для одной программы могут быть созданы несколько процессов в том случае, если с помощью одной программы в компьютере выполняется несколько несовпадающих последовательностей команд. За время существования процесс многократно изменяет свое состояние.

Различают следующие состояния процесса:

1) новый (new, процесс только что создан);

2) выполняемый (running, команды программы выполняются в CPU);

3) ожидающий (waiting, процесс ожидает завершения некоторого события, чаще всего операции ввода-вывода);

4) готовый (ready, процесс ожидает освобождения CPU);

5) завершенный (terminated, процесс завершил свою работу).

Переход из одного состояния в другое не может выполняться произвольным образом.

Во многих операционных системах вся информация о каждом процессе, дополнительная к содержимому его собственного адресного пространства, хранится в *таблице процессов* операционной системы.

Приведем некоторые поля таблицы.

Управление процессом	Управление памятью	Управление файлами
Регистры Счетчик команд Указатель стека Состояние процесса Приоритет Параметры планирования Идентификатор процесса Родительский процесс Группа процесса Время начала процесса Использованное процессорное время	Указатель на текстовый сегмент Указатель на сегмент данных Указатель на сегмент стека	Корневой каталог Рабочий каталог Дескрипторы файла Идентификатор пользователя Идентификатор группы

Каждый процесс представлен в операционной системе набором данных, называемых process control block. В process control block процесс описывается набором значений, параметров, характеризующих его текущее состояние и используемых операционной системой для управления прохождением процесса через компьютер.

Вычислительный процесс создается либо непосредственно введенной с клавиатуры командой, либо через командный файл. Создание вычислительного процесса включает в себя как минимум загрузку программы и создание управляющих блоков.

В результате создания в системе появляется новый процесс, который включается в мультипрограммную смесь и ОС начинает его видеть. Таким образом, процесс находится в состоянии готовности.

Процесс может быть диспетчеризован.

Диспетчеризация — это действие, в результате которого один из готовых процессов становится активным. Выбор одного процесса, который станет активным, из нескольких основан на правиле, называемом *дисциплиной диспетчеризации*. Диспетчер просматривает список готовых процессов и применяет алгоритм:

- LIFO — Last In First Out;
- FIFO — First In First Out;
- SJF — Short Job First.

Приоритеты бывают:

- относительными;
- абсолютными.

Эти дисциплины особенно важны в системах оперативной обработки, когда критерием является время ответа.

Все дисциплины можно использовать с квантованием и без квантования. *Квантование* — прием, при котором время обслуживания делят на части постоянной или переменной длины, называемые квантами. Если обслуживание за один квант не закончено, то оно искусственно прерывается и ресурс отдается следующей работе. Такая дисциплина называется *циклическим обслуживанием*.

Диспетчеризация является составной частью каждого цикла переключения при реализации процессов.

Когда процесс находится в активном состоянии, выполняются команды процесса.

Фундаментальная проблема — взаимная блокировка.

Существует два подхода:

1) файлы распределяются предварительно;

2) ресурсы распределяются динамически (в момент, когда они нужны).

Тупиковая ситуация, взаимная блокировка, тупик (deadlock) — ситуация, возникающая при параллельной обработке, когда несколько процессов, использующих общие ресурсы, не позволяют друг другу продолжить работу. Например, пусть есть два процесса А и Б, которым требуются ресурсы *X* и *Y*. Пусть процесс А сначала запрашивает ресурс *X* и захватывает его, а процесс Б параллельно захватывает ресурс *Y*, который в это время еще свободен. В результате возникает ситуация, когда ни один из двух процессов не может продолжать работу: процесс А ждет, когда процесс Б закончит свою работу и освободит ресурс *Y*, а процесс Б ждет, когда процесс Б закончит свою работу и освободит ресурс *X*.

Если процессы, развивающиеся динамически, например, пытаются обратиться к одним и тем же файлам, то они блокируют друг друга — это и называется полной блокировкой или тупиком.

Активное состояние может быть прервано в следующих случаях:

1. *По особому состоянию*. Особое состояние — ошибка в программе. Происходит аварийное завершение. При этом ОС пытается сохранить информацию для последующего анализа причины особого состояния. Объем этой информации может быть разным (dump). Dump — распечатка содержимого оперативной память (ОП) или файла, обычно без учета внутренней структуры данных. Копия содержимого регистров, нужного участка ОП, блока данных или файла выводится на печать последовательно в форме двоичного, восьме-

ричного или шестнадцатеричного кодов. Дамп применяется для анализа работы программных средств. Например, дамп, полученный после аварийного завершения программы, служит материалом для выявления причин аварийного завершения.

2. *Естественное завершение*. Процесс остается в памяти и получает флаг — завершенный процесс.

3. *По прерыванию*. При этом процесс переводится в состояние ожидания.

Логика обработки прерывания зависит от архитектуры процессора. При этом учитывается два обстоятельства.

А. Необходим некий механизм связи между процессом, находящимся в состоянии ожидания и ОС, который должен устанавливать тот факт, что событие наступило. Для связи с процессом создается таблица ожиданий.

Б. Обращение к ОС не всегда может быть реализовано немедленно. ОС пытается выполнить функцию, которую у нее запросили, например обращение к устройству. Если оно свободно, то это возможно. Если же устройство занято, то должна возникнуть очередь к устройству. Процесс уже попал в состояние ожидания, и повторное обращение со стороны процесса невозможно. Поэтому необходим механизм ведения очередей отложенных запросов. При выполнении запроса ОС создает соответствующую структуру, которая описывает этот запрос (Request Control Block). Эту структуру и ставят в очередь.

Очередь к внешним устройствам (ВУ) обслуживается по определенным дисциплинам (*диспетчеризация ввода-вывода*).

Из состояния ожидания процесс выходит по прерыванию от ВУ. Можно организовать синхронное ожидание. Для этого должен работать таймер.

В состоянии завершения может находиться много процессов. Если завершение аварийное, то надо сделать дамп. Необходимо освободить ресурсы, имеющиеся у процессора. В этот момент:

- файлы становятся доступными (после CLOSE);
- уничтожаются временные файлы;
- освобождается физическая память;
- уничтожаются управляющие структуры, предварительно все системы формируют файлы хронологии.

Для выполнения отложенного ввода-вывода существует два способа:

- использование выделенного ВУ;
- спулинг.

В многозадачной системе реальный процессор переключается с процесса на процесс, но для упрощения модели рассматривается набор процессов, идущих параллельно (псевдопараллельно).

Рассмотрим схему с четырьмя работающими процессами (рис. 5.1).

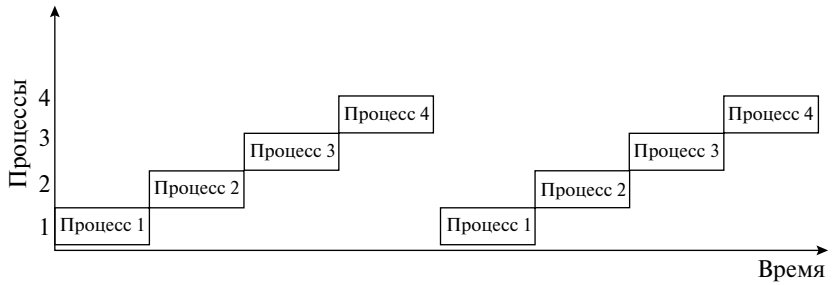


Рис. 5.1. Схема с четырьмя работающими процессами, идущими параллельно

В каждый момент времени активен только один процесс. На рис. 5.2 приведен пример параллельно работающих процессоров со счетчиком команд.

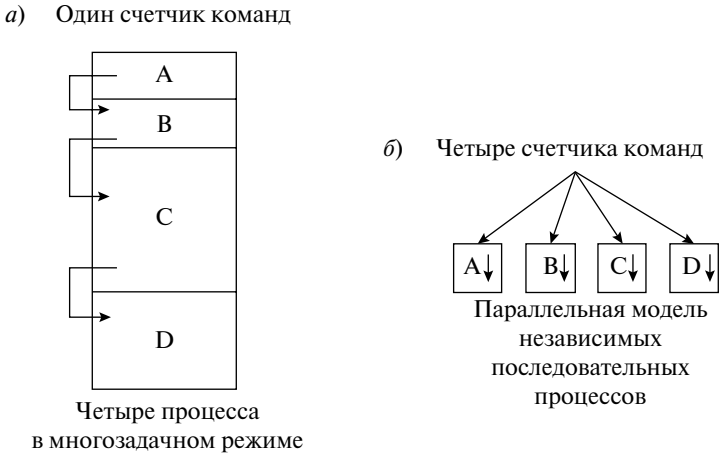


Рис. 5.2. Схема параллельно работающих процессов со счетчиком команд: *а* — с одним; *б* — с четырьмя

Справа представлены параллельно работающие процессы, каждый со своим счетчиком команд. Разумеется, на самом деле существует

только один физический счетчик команд, в который загружается логический счетчик команд текущего процесса. Когда время, отведенное текущему процессу, заканчивается, физический счетчик команд сохраняется в памяти, в логическом счетчике команд процесса.

Средства создания и завершения процессов

Рассмотрим основные средства создания и завершения процессов на примере операционной системы UNIX.

Системный вызов fork. Для создания нового процесса используется системный вызов fork. Все процессы ОС UNIX, кроме начального, запускаемого при «раскрутке» системы, образуются при помощи системного вызова fork. После создания процесса-потомка процесс-предок и процесс-потомок начинают «жить» своей собственной жизнью, произвольным образом изменяя свой контекст. Например, и процесс-предок, и процесс-потомок могут выполнить системный вызов exec (см. ниже), приводящий к полному изменению контекста процесса.

Системный вызов wait. Системный вызов wait используется для синхронизации процесса-предка и процессов-потомков. Выполнение этого системного вызова приводит к приостановке выполнения процесса-предка до тех пор, пока не завершится выполнение какого-либо процесса, являющегося его потомком.

Сигналы. Сигнал — это способ информирования процесса со стороны ядра операционной системы о происшествии некоторого события (event) в системе, например:

- исключительная ситуация (выход за допустимые границы виртуальной памяти, попытка записи в область виртуальной памяти, которая доступна только для чтения, и т.д.);
- ошибка в системном вызове (несуществующий системный вызов, ошибки в параметрах системного вызова и т.д.);
- прием сообщения от другого процесса;
- нажатие пользователем определенных клавиш на клавиатуре терминала, связанного с процессом.

Все возможные в системе сигналы имеют уникальные номера и идентификаторы.

С помощью системного вызова signal пользовательская программа может осуществить «перехват» указанного в вызове сигнала — вызвать соответствующую функцию, которая выполнит обработку этого сигнала. Например, вызов signal (SIGFPE, error) вызовет выполнение функции error при переполнении или делении на ноль во время выполнения операции с плавающей запятой.

ОС Unix предоставляет возможность пользовательским процессам направлять сигналы другим процессам. Например, системный вызов kill (PID, signum) посылает процессу с идентификатором PID сигнал с номером signum.

Системный вызов exec. При выполнении системного вызова exec (filename,...), где filename — имя выполняемого файла, операционная система производит реорганизацию виртуальной памяти вызывающего процесса, уничтожая в ней сегменты старого программного кода и образуя новые сегменты, в которые загружается программный код из файла filename.

Управление памятью

Память является важнейшим ресурсом, требующим тщательного управления со стороны мультипрограммной ОС. Особая роль памяти объясняется тем, что процессор может выполнять инструкции программы только в том случае, если они находятся в памяти.

В ранних ОС управление памятью сводилось просто к загрузке программы и ее данных из некоторого внешнего накопителя (перфоленты, магнитной ленты или магнитного диска) в память. С появлением мультипрограммирования перед ОС были поставлены новые задачи, связанные с распределением имеющейся памяти между несколькими работающими программами.

Функции ОС по управлению памятью в мультипрограммной системе:

- отслеживание свободной и занятой памяти;
- выделение памяти процессам и освобождение памяти при завершении процессов;
- вытеснение кодов и данных процессов из ОП на диск (полное или частичное), когда размеры основной памяти недостаточны для размещения в ней всех процессов, и возвращение их в ОП, когда в ней освобождается место;
- настройка адресов программы на конкретную область физической памяти.

Дополнительные функции:

- динамическое распределение памяти, т.е. выполнение запросов приложений на выделение им дополнительной памяти на время выполнения;
- создание новых служебных информационных структур (описателей процессов и потоков, буферов и др.);
- защита памяти, которая состоит в том, чтобы не позволить выполняемому процессу записывать или читать данные из памяти, назначенной другому процессу.

В связи с тем что оперативной памяти часто не хватает, для выполнения процессов часто приходится использовать внешнюю память (дисковую).

Основные способы использования диска:

- свопинг (подкачка) — процесс целиком загружается в память для работы;
- виртуальная память — процесс может быть частично загружен в память для работы.

Свопинг (подкачка). При нехватке памяти процессы могут быть выгружены на диск (рис. 5.3).

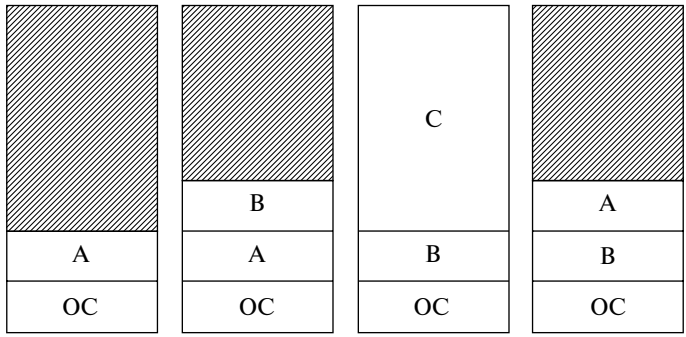


Рис. 5.3. Свопинг (подкачка) — процесс целиком загружается в память

Так как процесс *C* очень большой, процесс *A* был выгружен временно на диск, после завершения процесса *C* он снова был загружен в память.

Как мы видим, процесс *A* второй раз загрузился в другое адресное пространство; должны создаваться такие условия, которые не повлияют на работу процесса.

Свопер — планировщик, управляющий перемещением данных между памятью и диском.

Этот метод был основным для UNIX до версии 3BSD.

Виртуальная память. Одним из наиболее популярных способов управления памятью в современных ОС является так называемая виртуальная память. Наличие в ОС виртуальной памяти позволяет программисту писать программу так, как будто в его распоряжении имеется однородная ОП большого объема, часто существенно превышающего объем имеющейся физической памяти. В действительности все данные, используемые программой, хранятся на диске и при необходимости частями (сегментами или страницами) отображаются в физические адреса ячеек ОП.

Типы адресов

Для идентификации переменных и команд на разных этапах цикла программы используются символьные номера (метки), виртуальные адреса и физические адреса (рис. 5.4).

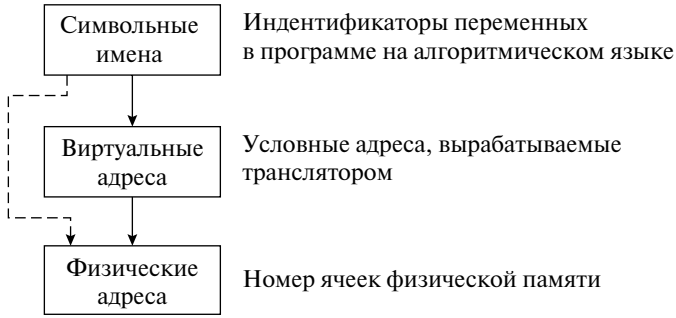


Рис. 5.4. Идентификация переменных и команд

Символьные номера присваивает пользователь при написании программы на алгоритмическом языке или ассемблере (имена переменных и входных точек программных модулей).

Физические адреса соответствуют номерам ячеек ОП, где в действительности будут расположены переменные и команды. Физическая память представляет собой упорядоченное множество ячеек, и все они пронумерованы, т.е. к каждой из них можно обратиться, указав ее порядковый номер (адрес). Количество ячеек физической памяти ограничено и фиксировано.

Виртуальные адреса (математические или логические) вырабатывает транслятор, переводящий программу на математический язык. Поскольку во время трансляции в общем случае неизвестно, в какое место ОП будет загружена программа, то транслятор присваивает переменным и командам виртуальные (условные) адреса, обычно считая по умолчанию, что начальным адресом будет нулевой адрес.

ОС должна связать каждое указанное пользователем имя с физической ячейкой памяти, т.е. осуществить отображение пространства имен на физическую память компьютера. В общем случае это отображение осуществляется в два этапа: сначала системой программирования, а затем ОС (с помощью специальных программных модулей управления памятью и использования соответствующих аппаратных средств вычислительной системы). Между этими этапами обращения к памяти имеют форму виртуального адреса.

Переход от виртуальных адресов к физическим может осуществляться двумя способами:

Первый способ. Замену виртуальных адресов на физические делает специальная системная программа — перемещающий загрузчик. Перемещающий загрузчик на основании имеющихся у него исходных данных о начальном адресе физической памяти, в которую предстоит загружать программу, и информации, предоставленной транслятором об адресно-зависимых константах программы, выполняет загрузку программы и одновременно заменяет виртуальные адреса на физические.

Второй способ. Программа загружается в память в неизмененном виде в виртуальных адресах, при этом операционная система фиксирует смещение действительного расположения программного кода относительно виртуального адресного пространства. Во время выполнения программы при каждом обращении к оперативной памяти выполняется преобразование виртуального адреса в физический. Второй способ является более гибким, он допускает перемещение программы во время ее выполнения, в то время как перемещающий загрузчик жестко привязывает программу к первоначально выделенному ей участку памяти. Вместе с тем использование перемещающего загрузчика уменьшает накладные расходы, так как преобразование каждого виртуального адреса происходит только один раз во время загрузки, а во втором случае — каждый раз при обращении по данному адресу.

В некоторых случаях (обычно в специализированных системах), когда заранее точно известно, в какой области оперативной памяти будет выполняться программа, транслятор выдает исполняемый код сразу в физических адресах.

Виртуальное адресное пространство

Диапазон виртуальных адресов определяется программно-аппаратным обеспечением компьютера, в частности разрядностью его схем адресации. Совокупность всех возможных адресов из этого диапазона называется *виртуальным адресным пространством* (ВАП).

Так, 32-разрядный процессор семейства x86 дает возможность адресовать до 2^{32} байт, т.е. до 4 Гбайт памяти, с диапазоном виртуальных адресов от 00000000_h до $FFFFFFFF_h$.

Реальные процессы используют только часть доступного виртуального пространства (на 1–2 порядка меньше максимума).

Совпадение виртуальных адресов переменных и команд различных программ не приводит к конфликтам, так как, в случае когда эти

переменные или команды одновременно присутствуют в памяти, операционная система отображает совпадающие виртуальные адреса на разные физические (если эти переменные или команды не должны разделяться соответствующими процессами) адреса.

Образ процесса — термин, обозначающий содержимое назначенного процессу виртуального адресного пространства, т.е. коды команд и данные (исходные, промежуточные и результаты).

Таким образом, основная идея виртуальной памяти заключается в разбиении программы на части, и в память эти части загружаются по очереди. Программа при этом общается с виртуальной памятью, а не с физической. Диспетчер памяти преобразует виртуальные адреса в физические.

Страничная организация памяти

Страницы — это части, на которые разбивается пространство виртуальных адресов.

Страничные блоки — единицы физической памяти.

Страницы всегда имеют фиксированный размер. Передача данных между ОЗУ и диском всегда происходит в страницах (рис. 5.5).

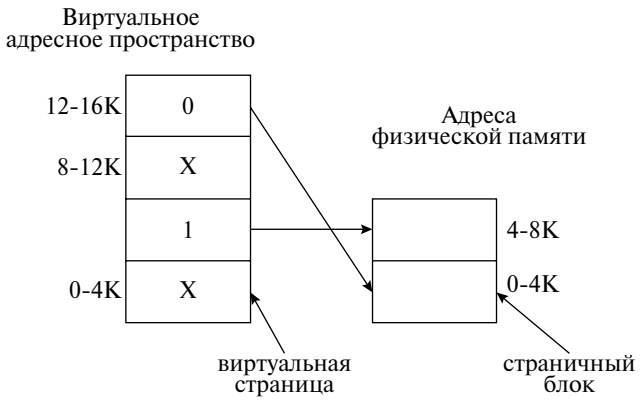


Рис. 5.5. Страничная организация памяти

X — обозначает неотображаемую страницу в физической памяти.

Страничное прерывание происходит, если процесс обратился к странице, которая не загружена в ОЗУ (т.е. X). Процессор передается другому процессу, и параллельно страница загружается в память.

Таблица страниц используется для хранения соответствия адресов виртуальной страницы и страничного блока (рис. 5.6).

Таблица может быть размещена:

- в аппаратных регистрах (преимущество: более высокое быстродействие; недостаток — высокая стоимость);
- в ОЗУ.

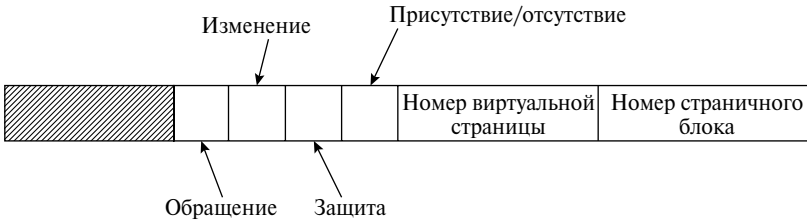


Рис. 5.6. Типичная запись в таблице страниц

Присутствие/отсутствие — загружена или не загружена в память.

Защита — виды доступа, например чтение/запись.

Изменение — изменилась ли страница; если да, то при выгрузке записывается на диск, если нет, просто уничтожается.

Обращение — было ли обращение к странице; если нет, то это лучший кандидат на освобождение памяти.

Информация об адресе страницы, когда она хранится на диске, в таблице не размещается.

Для ускорения доступа к страницам в диспетчере памяти создают *буфер быстрого преобразования адреса*, в котором хранится информация о наиболее часто используемых страницах.

Страничная организация памяти используется и в UNIX, и в Windows.

Способы структуризации виртуального адресного пространства в ОС

Структура адреса или модель адресации определяется в совокупности компилятором, операционной системой и аппаратным обеспечением. Компилятор должен обеспечить простоту работы с адресом, но с минимальным разрывом между программистом и ОС. Поэтому в языке программирования отображается та модель, которая используется в ОС. Эта модель, в свою очередь, определяется заложенной в ОС идеей адресации с учетом необходимости реализации этой идеи на конкретной аппаратной платформе. Таким образом, ОС «сверху» должна обеспечить достаточно простую модель адресации для компилятора, а «снизу» уметь преобразовать эту модель в модель, навязанную аппаратурой.

Рассмотрим две наиболее характерных модели структуризации адресного пространства — плоскую и двухуровневую модель «сегмент—смещение». Эти модели представлены на рис. 5.7.

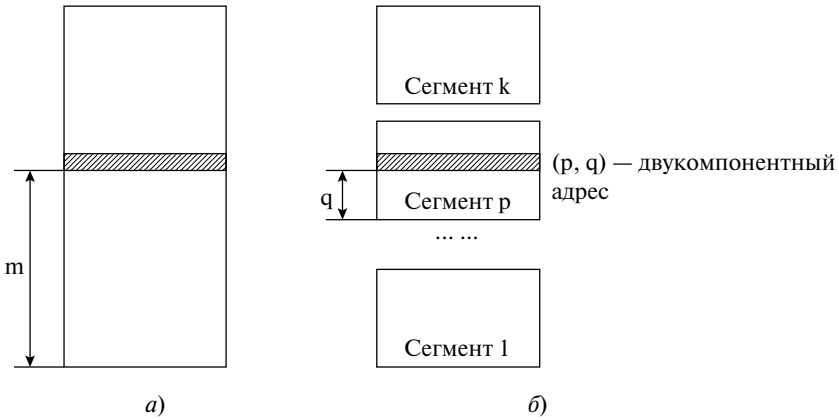


Рис. 5.7. Типы виртуальных адресных пространств:
а — плоское, *б* — сегментированное

Плоская (flat)структура. Виртуальное адресное пространство представлено в виде непрерывной линейной последовательности адресов. Линейный виртуальный адрес — число, представляющее собой смещение относительно начала виртуального адресного пространства (обычно это нулевое значение).

Сегментированная структура. Виртуальное адресное пространство представляется разделенным на сегменты, а адрес любого объекта в памяти определяется номером сегмента и смещением относительно начала этого сегмента, т.е. парой «сегмент—смещение».

Более конкретно способы структуризации виртуального адресного пространства рассмотрены в [5].

Важно отметить следующее. Использование и реализация универсального принципа сегментирования структуры адресного пространства в разные периоды развития вычислительной техники были принципиально различными и менялись по крайней мере трижды.

В ранних ОС на сегменты фиксированного размера делилась физическая память, о чем пользователь должен был знать и что при необходимости учитывал в программе. Необходимость структуризации адреса диктовалась архитектурой процессора и памяти. Так, модель памяти «сегмент—смещение» была реализована в 32-разрядной

архитектуре IBM-360 (объем памяти оказывался меньше потенциально адресуемого, но, тем не менее, была реализована модель «сегмент—смещение») и в 16-разрядной архитектуре x-86 (по причине сугубо аппаратного свойства: процессор использовал 20-разрядную шину адреса, располагая 16-разрядными регистрами, и для формирования адреса использовалось два регистра).

Для программы адресное пространство представляется плоским.

Подходы к преобразованию виртуальных адресов в физические

Загрузка программы с совместной заменой виртуальных адресов физическими адресами. Замена адресов выполняется один раз. Программа «перемещающий загрузчик», имея начальный адрес загрузки (т.е. адрес оперативной памяти, начиная с которого будет размещена программа) и код в относительных (виртуальных) адресах, выполняет загрузку с одновременным увеличением виртуальных адресов на величину начального адреса загрузки.

Динамическое преобразование виртуальных адресов. Программа загружается в память в виртуальных адресах. Начальный адрес загрузки ОС фиксирует в специальном регистре. Преобразование виртуальных адресов в физические (также путем прибавления начального адреса загрузки) производится во время выполнения программы при обращении к памяти. Таким образом, некоторый виртуальный адрес пересчитывается в физический столько раз, сколько обращений по нему производится.

Этот способ более гибок, так как позволяет перемещать программный код процесса во время выполнения, но менее экономичен из-за многократных преобразований одних и тех же адресов.

5.3. Интерфейс прикладного программирования

API (англ. Application Programming Interface — интерфейс прикладного программирования) — набор готовых классов, процедур, функций, структур и констант, предоставляемых приложением (библиотекой, сервисом) или операционной системой для использования во внешних программных продуктах.

Практически все операционные системы (UNIX, Windows, OS X и т.д.) имеют API, с помощью которого программисты могут создавать приложения для этой операционной системы. Главный API операционных систем — это множество системных вызовов.

Например, если пользовательскому процессу необходимо считать данные из файла, он должен выполнить команду системного вызова, т.е. выполнить прерывание с переключением в режим ядра и активизировать функцию операционной системы для считывания данных из файла.

В POSIX существует более 100 системных вызовов. Ниже приведены примеры наиболее часто применяемых системных вызовов стандарта POSIX:

- fork — создание нового процесса;
- exit — завершение процесса;
- open — открывает файл;
- close — закрывает файл;
- read — читает данные из файла в буфер;
- write — пишет данные из буфера в файл;
- stat — получает информацию о состоянии файла;
- mkdir — создает новый каталог;
- rmdir — удаляет каталог;
- link — создает ссылку;
- unlink — удаляет ссылку;
- mount — монтирует файловую систему;
- umount — демонтирует файловую систему;
- chdir — изменяет рабочий каталог.

В UNIX вызовы почти один к одному идентичны библиотечным процедурам, которые используются для обращения к системным вызовам.

Рассмотрим интерфейс прикладного программирования для Windows — **Win32 API.** Win32 API отделен от системных вызовов. Это позволяет в разных версиях менять системные вызовы, не переписывая программы. Поэтому непонятно, является ли вызов системным (выполняется ядром) или он обрабатывается в пространстве пользователя.

В Win32 API существует более 1000 вызовов. Такое количество связано и с тем, что графический интерфейс пользователя UNIX запускается в пользовательском режиме, а в Windows встроен в ядро. Поэтому Win32 API имеет много вызовов для управления окнами, текстом, шрифтами и т.д.

Вызовы Win32 API подобны вызовам стандарта POSIX. Примеры вызовов Win32 API:

- CreatProcess (fork) — создание нового процесса;
- ExitProcess (exit) — завершение процесса;

- CreatFile (open) — открывает файл;
- CloseHandle (close) — закрывает файл;
- ReadFile (read) — читает данные из файла в буфер;
- WriteFile (write) — пишет данные из буфера в файл;
- CreatDirectory (mkdir) — создает новый каталог;
- RemoveDirectory (rmdir) — удаляет каталог;
- SetCurrentDirectory (chdir) — изменяет рабочий каталог.

Интерфейс Win32 API позволяет программам работать почти на всех версиях Windows.

В индустрии программного обеспечения общие стандартные API для стандартной функциональности имеют важную роль, так как они гарантируют, что все программы, использующие общий API, будут работать одинаково хорошо или по крайней мере типичным, привычным образом. В случае API графических интерфейсов это означает, что программы будут иметь похожий пользовательский интерфейс, что облегчает процесс освоения новых программных продуктов.

С другой стороны, отличия в API различных операционных систем существенно затрудняют перенос приложений между платформами. Существуют различные методы обхода этой сложности — написание «промежуточных» API (API графических интерфейсов WxWidgets, Qt, GTK и т.п.), написание библиотек, которые отображают системные вызовы одной ОС в системные вызовы другой ОС (такие среды исполнения, как Wine, cygwin и т.п.), введение стандартов кодирования в языках программирования (например, стандартная библиотека языка C), написание интерпретируемых языков, реализуемых на разных платформах (sh, python, perl, php, tcl, Java и т.д.).

Также необходимо отметить, что в распоряжении программиста часто находится несколько различных API, позволяющих добиться одного и того же результата. При этом каждый API обычно реализован с использованием API программных компонент более низкого уровня абстракции.

Например, для того чтобы увидеть в браузере строчку «Hello, world!», достаточно лишь создать HTML-документ с минимальным заголовком и простейшим телом, содержащим данную строку. Когда браузер откроет этот документ, программа-браузер передаст имя файла (или уже открытый дескриптор файла) библиотеке, обрабатывающей HTML-документы; та, в свою очередь, при помощи API операционной системы прочитает этот файл и разберется в его

устройстве, затем последовательно вызовет через API библиотеки стандартных графических примитивов операции типа «очистить окошко», «написать “Hello, world!” выбранным шрифтом». Во время выполнения этих операций библиотека графических примитивов обратится к библиотеке оконного интерфейса с соответствующими запросами; уже эта библиотека обратится к API операционной системы, чтобы записать данные в буфер видеокарты.

При этом практически на каждом из уровней реально существует несколько возможных альтернативных API. Например, мы могли бы писать исходный документ не на HTML, а на LaTeX, для отображения могли бы использовать любой браузер. Различные браузеры, вообще говоря, используют различные HTML-библиотеки; и кроме того, все это может быть собрано с использованием различных библиотек примитивов и на различных операционных системах.

Основными сложностями существующих многоуровневых систем API, таким образом, являются:

- сложность портирования программного кода с одной системы API на другую (например, при сменеОС);
- потеря функциональности при переходе с более низкого уровня на более высокий. Грубо говоря, каждый «слой» API создается для облегчения выполнения некоторого стандартного набора операций. Но при этом реально затрудняется либо становится принципиально невозможным выполнение некоторых других операций, которые предоставляет более низкий уровень API.

Контрольные вопросы и задания

1. Составьте алгоритм действий для считывания или записи на информационную дискету.
2. Постройте схему работы процессора в многозадачной системе с тремя работающими параллельно процессами.
3. Опишите процедуру загрузки программы с совместной заменой виртуальных адресов физическими адресами.
4. Опишите процедуру динамического преобразования виртуальных адресов.
5. Что такое виртуальный адрес, виртуальное адресное пространство?
6. Назовите функции ОС по управлению памятью в мультипрограммной среде.
7. Перечислите типы виртуальных адресных пространств.
8. Какие виды алгоритмов распределения памяти вы знаете?
9. Назовите подходы к преобразованию виртуальных адресов в физические.
10. Что представляет страничная организация памяти?

11. Опишите динамическое преобразование виртуальных адресов.
12. Каково назначение перемещающего загрузчика?
13. Раскройте понятия процессов, потоков и файберов.
14. Приведите примеры наиболее часто применяемых системных вызовов стандарта POSIX.
15. Какие вы знаете средства создания и завершения процессов?
16. Опишите основные средства создания и завершения процессов на примере операционной системы UNIX.
17. Приведите примеры вызовов Win32 API.
18. Укажите основные сложности существующих многоуровневых систем API.

Глава 6
УРОВЕНЬ АССЕМБЛЕРА

Мы рассмотрели три уровня архитектуры вычислительных машин. В этой главе речь пойдет о еще одном уровне, который также присутствует в архитектуре практически всех современных машин. Это уровень ассемблера, который существенно отличается от трех предыдущих уровней, поскольку он реализуется путем трансляции, а не интерпретации. На этом уровне рассмотрим описание ассемблера и архитектуры MIPS с использованием примеров для разных архитектур ЭВМ, рассмотрим логику работы компьютера с памятью, а также некоторые особенности режимов адресации.

6.1. Язык ассемблера. Начальные сведения

Ассемблер — транслятор исходного текста программы, написанной на языке ассемблера, в программу на машинном языке.

Язык ассемблера (англ. assembly language) — машинно-ориентированный язык низкого уровня с командами, не всегда соответствующими командам машины, который может обеспечить дополнительные возможности вроде макрокоманд; автокод, расширенный конструкциями языков программирования высокого уровня, такими как выражения, макрокоманды, средства обеспечения модульности программ.

Данный тип языков получил свое название от названия транслятора (компилятора) с этих языков — ассемблера (англ. assembler — сборщик). Название обусловлено тем, что программа «автоматически собиралась», а не вводилась вручную покомандно непосредственно в кодах. При этом наблюдается путаница терминов: ассемблером нередко называют не только транслятор, но и соответствующий язык программирования («программа на ассемблере»).

Использование термина «язык ассемблера» также может вызвать ошибочное мнение о существовании некоего единого языка низкого уровня или хотя бы стандартов на такие языки. При именовании языка ассемблера желательно уточнять, ассемблер для какой архитектуры имеется в виду.

Язык ассемблера — система обозначений, используемая для представления в удобочитаемой форме программ, записанных в машинном коде. Язык ассемблера позволяет программисту пользоваться

алфавитными мнемоническими кодами операций, по своему усмотрению присваивать символические имена регистрам ЭВМ и памяти, а также задавать удобные для себя схемы адресации (например, индексную или косвенную). Кроме того, он позволяет использовать различные системы счисления (например, десятичную или шестнадцатеричную) для представления числовых констант и дает возможность помечать строки программы метками с символическими именами, с тем чтобы к ним можно было обращаться (по именам, а не по адресам) из других частей программы (например, для передачи управления).

Перевод программы на языке ассемблера в исполнимый машинный код (вычисление выражений, раскрытие макрокоманд, замена мнемоник собственно машинными кодами и символьных адресов на абсолютные или относительные адреса) производится ассемблером — программой-транслятором, которая и дала языку ассемблера его название.

Команды языка ассемблера один к одному соответствуют командам процессора. Фактически они и представляют собой более удобную для человека символьную форму записи — мнемокоды — команд и их аргументов. При этом одной команде языка ассемблера может соответствовать несколько вариантов команд процессора.

Кроме того, язык ассемблера позволяет использовать символические метки вместо адресов ячеек памяти, которые при ассемблировании заменяются на вычисляемые ассемблером или компоновщиком абсолютные или относительные адреса, а также так называемые директивы (команды ассемблера, не переводимые в машинные команды процессора, а выполняемые самим ассемблером).

Директивы ассемблера позволяют, в частности, включать блоки данных, задать ассемблирование фрагмента программы по условию, задать значения меток, использовать макрокоманды с параметрами.

Каждая модель (или семейство) процессоров имеет свой набор (систему) команд и соответствующий ему язык ассемблера. Наиболее популярные синтаксисы языков ассемблера — Intel-синтаксис и AT&T-синтаксис.

Достоинства языка. Язык ассемблера позволяет писать самый быстрый и компактный код, какой вообще возможен для данного процессора.

Если код программы достаточно большой — данные, которыми он оперирует, не помещаются целиком в регистрах процессора, т.е. частично или полностью находятся в оперативной памяти,— то искусный программист, как правило, способен значительно оптими-

зировать программу по сравнению с транслятором с языка высокого уровня по одному или нескольким параметрам и создать код, близкий к оптимальному по Парето (как правило, быстродействие программы достигается за счет удлинения кода и наоборот):

- скорость работы — за счет оптимизации вычислительного алгоритма и/или более рационального обращения к оперативной памяти (ОП)(например, если все исходные данные хранятся в регистрах процессора, то можно исключить излишние обращения к ОП), перераспределения данных, табличного вычисления функций;
- объем кода (в том числе за счет эффективного использования промежуточных результатов). Сокращение объема кода также нередко повышает скорость выполнения программы;
- обеспечение максимального использования специфических возможностей конкретной платформы, что также позволяет создавать более эффективные программы, в том числе менее ресурсоемкие.

При программировании на языке ассемблера возможен непосредственный доступ к аппаратуре, в частности портам ввода-вывода, регистрам процессора и др. Во многих операционных системах прямое обращение из прикладных программ для записи в регистры периферийного оборудования блокировано для надежности работы системы и исключения «зависаний».

Язык ассемблера часто применяется для создания драйверов оборудования и ядра операционной системы (или машиннозависимых подсистем ядра ОС) тогда, когда важно временно́е согласование работы периферийных устройств с центральным процессором.

Язык ассемблера используется для создания «прошивок» BIOS.

С помощью языка ассемблера часто создаются машиннозависимые подпрограммы компиляторов и интерпретаторы языков высокого уровня, а также реализуется совместимость платформ.

С помощью программы дизассемблера можно понять алгоритмы работы исследуемой программы при отсутствии листинга на высокоуровневом языке, изучая только машинные коды, но в сложных нетривиальных программах это очень и очень трудоемко.

Недостатки языка. В силу машинной ориентации («низкого» уровня) языка ассемблера человеку сложнее читать и понимать программу на нем по сравнению с языками программирования высокого уровня; программа состоит из слишком «мелких» элементов — машинных команд; соответственно, усложняются программирование и отладка, растут трудоемкость и вероятность внесения ошибок.

Требуется повышенная квалификация программиста для получения качественного кода: код, написанный средним программистом на языке ассемблера, обычно оказывается не лучше или даже хуже кода, порождаемого оптимизирующим компилятором для сравнимых программ, написанных на языке высокого уровня [5].

Программа на языке высокого уровня может быть перекомпилирована с автоматической оптимизацией под особенности новой целевой платформы, программа же на языке ассемблера на новой платформе может потерять свое преимущество в скорости без ручного переписывания кода.

Как правило, меньшее количество доступных библиотек по сравнению с современными индустриальными языками программирования.

Отсутствует переносимость программ на компьютеры с другой архитектурой и системой команд.

Применение языка ассемблера

Исторически, если первым поколением языков программирования считать машинные коды, язык ассемблера можно рассматривать как второе поколение языков программирования. Недостатки языка ассемблера, сложность разработки на нем больших программных комплексов привели к появлению языков третьего поколения — языков программирования высокого уровня (таких как Фортран, Лисп, Кобол, Паскаль, Си и др.). Именно языки программирования высокого уровня и их наследники в основном используются в настоящее время в индустрии информационных технологий. Однако языки ассемблера сохраняют свою нишу, обусловленную их уникальными преимуществами в части эффективности и возможности полного использования специфических средств конкретной платформы.

На языке ассемблера пишут программы или их фрагменты в тех случаях, когда критически важны:

- быстродействие (драйверы, игры);
- объем используемой памяти (загрузочные секторы, встраиваемое (англ. embedded) программное обеспечение, программы для микроконтроллеров и процессоров с ограниченными ресурсами, вирусы, программные защиты).

С использованием программирования на языке ассемблера производятся:

- оптимизация критичных к скорости участков программ в программах на языках высокого уровня, таких как C++ или Pascal. Это особенно актуально для игровых приставок, имеющих фиксирован-

ную производительность, и для мультимедийных кодеков, которые стремятся делать менее ресурсоемкими и более быстрыми;

- создание операционных систем (ОС) или их компонентов. В настоящее время подавляющее большинство ОС пишут на более высокоуровневых языках (в основном на Си — языке высокого уровня, который специально был создан для написания одной из первых версий UNIX). Аппаратно зависимые участки кода, такие как загрузчик ОС, уровень абстрагирования от аппаратного обеспечения (hardware abstraction layer) и ядро, часто пишутся на языке ассемблера. Фактически ассемблерного кода в ядрах Windows или Linux совсем немного, поскольку авторы стремятся обеспечить переносимость и надежность, но, тем не менее, он там присутствует. Некоторые любительские ОС, такие как MenuetOS и KolibriOS, целиком написаны на языке ассемблера. При этом MenuetOS и KolibriOS помещаются на дискету и содержат графический многооконный интерфейс;
- программирование микроконтроллеров (МК) и других встраиваемых процессоров. По мнению профессора Таненбаума, развитие МК повторяет историческое развитие компьютеров новейшего времени [9]. Сейчас для программирования МК весьма часто применяют язык ассемблера (хотя и в этой области широкое распространение получают языки вроде Си). В МК приходится перемещать отдельные байты и биты между различными ячейками памяти. Программирование МК весьма важно, так как, по мнению Таненбаума, в автомобиле и квартире современного цивилизованного человека в среднем содержится 50 микроконтроллеров [1];
- создание драйверов. Драйверы (или их некоторые программные модули) программируют на языке ассемблера. Хотя в настоящее время драйверы также стремятся писать на языках высокого уровня (на высокоуровневом языке много проще написать надежный драйвер) в связи с повышенными требованиями к надежности и достаточной производительностью современных процессоров (быстродействие обеспечивает временно́е согласование процессов в устройстве и процессоре) и достаточным совершенством компиляторов с языков высокого уровня (отсутствие ненужных пересылок данных в сгенерированном коде), подавляющая часть современных драйверов пишется на языке ассемблера. Надежность для драйверов играет особую роль, поскольку в Windows NTиUNIX (в том числе в Linux) драйверы работают в режиме ядра системы. Одна тонкая ошибка в драйвере может привести к краху всей системы;

- создание антивирусов и других защитных программ;
- написание трансляторов языков программирования.

Связывание программ на разных языках

Поскольку уже давно на языке ассемблера часто кодируют только фрагменты программ, их необходимо связывать с остальными частями программной системы, написанными на других языках программирования. Это достигается двумя основными способами:

На этапе компиляции — вставка в исходный код программы на языке высокого уровня ассемблерных фрагментов (англ. inline assembler) с помощью специальных директив языка. Способ удобен для несложных преобразований данных, но полноценного ассемблерного кода с данными и подпрограммами, включая подпрограммы со множеством входов и выходов, не поддерживаемых языком высокого уровня, с его помощью сделать невозможно.

На этапе компоновки при раздельной компиляции. Для взаимодействия компонуемых модулей достаточно, чтобы импортируемые функции (определенные в одних модулях и используемые в других) поддерживали определенное соглашение о вызове (англ. calling conventions). Написаны же отдельные модули могут быть на любых языках, в том числе и на языке ассемблера.

6.2. Основы программирования на языке ассемблера

Программируя на аассемблере, необходимо хорошо представлять, как данные размещены в памяти компьютера.

Команды, исполняемые компьютером, могут быть записаны в безадресном (операнд включен в код команды), одноадресном, двухадресном и трехадресном формате. Иными словами, с помощью адресации определяются адреса размещения операндов в памяти компьютера.

Перечислим режимы адресации, которые чаще всего используются при программировании компьютеров семейства Intel x86, а также могут быть применены на платформе MIPS. При программировании важно учитывать, что предлагаемый список доступных режимов требуется уточнять в зависимости от того, какова архитектура целевой вычислительной системы.

В *непосредственном режиме адресации* (immediate addressing mode) сведения, необходимые для доступа к данным, включаются в команду процессора. К примеру, если мы хотим инициализировать регистр, поместив туда ноль, разумнее всего будет не указывать адрес,

откуда можно прочитать ноль, а использовать непосредственный режим адресации, указав ноль в качестве операнда команды.

Иными словами, при непосредственной адресации значение в формате байта, слова или двойного слова (машинное слово — это количество данных, которое процессор может обработать за одну операцию) передается в регистр-приемник или в память. Обычно машинное слово по длине равно размеру регистра общего назначения процессора. На рынке широко доступны процессоры, оперирующие словами по 4 байта (x86-32, MIPS32) и по 8 байтов (x86-64, MIPS64).

Непосредственный операнд может быть задан в различных системах счисления (шестнадцатеричной, двоичной, десятичной). Символы ASCII непосредственно задаются числовым кодом или буквой в кавычках.

В режиме *регистровой адресации* (register addressing mode) команда содержит имя регистра, в котором хранятся числа для обработки. Копия байта, слова или двойного слова из регистра-источника передается в регистр-приемник. Размерность регистров источника и приемника должна быть одинаковой.

Прямой режим адресации (direct addressing mode) подразумевает указание адреса памяти в качестве операнда команды (адрес данных в памяти образуется совокупностью значения, содержащегося в сегментном регистре, и смещения, заданного в команде). Данные пересылаются между памятью и регистром. К примеру, можно сказать: «Компьютер, загрузи, пожалуйста, в регистр данные, содержащиеся в памяти по адресу 2002». Среагировав на наши слова, компьютер перейдет к байту номер 2002 и скопирует его содержимое в нужный регистр.

В *режиме индексной адресации* (indexed addressing mode) команда содержит в качестве операнда базовый адрес, а также имя индексного регистра, содержащего значение смещения от базового адреса. К примеру, если базовый адрес равен 2002, а смещение равно 4, то адрес, по которому обратится команда, будет равен 2006.

С помощью индексной адресации можно организовывать циклы, в которых происходит последовательное обращение к ячейкам памяти, имеющим адрес, находящийся с разным смещением от базового адреса (базово-индексная адресация).

Процессоры x86 поддерживают использование множителя (multiplier), записываемого в индексный регистр (индексная адресация с масштабированием). Это позволяет получать доступ к участкам памяти по байту или по машинному слову за один проход цикла, а также переходить к нужному участку памяти.

Операндами команды могут являться:

- данные, которые явно либо неявно задаются машинной командой и которые должны быть обработаны либо выработаны в ходе исполнения машинной команды;
- явная ссылка в машинной команде на такие данные;
- место, где находятся или должны быть помещены такие данные: позиции в оперативной, внешней, внутренней или управляющей памяти; общие и специальные регистры, флаги, поля признаков, входы и выходы управляющих сигналов.

Работа с регистрами процессора выполняется намного быстрее, чем с ячейками оперативной памяти, поэтому регистры активно используются как в программах на языке ассемблера, так и компиляторами языков высокого уровня.

Регистр — это функциональный узел, осуществляющий прием, хранение и передачу информации. Регистры состоят из группы триггеров, обычно D. По типу приема и выдачи информации различают два типа регистров:

- с последовательным приемом и выдачей информации — сдвиговые регистры;
- с параллельным приемом и выдачей информации — параллельные регистры.

Сдвиговые регистры представляют собой последовательно соединенную цепочку триггеров. Основной режим работы — сдвиг разрядов кода от одного триггера к другому на каждый импульс тактового сигнала.

По назначению регистры разделяются:

- на аккумулятор — используется для хранения промежуточных результатов арифметических и логических операций и инструкций ввода-вывода;
- флаговые — хранят признаки результатов арифметических и логических операций;
- общего назначения — хранят операнды арифметических и логических выражений, индексы и адреса;
- индексные — хранят индексы исходных и целевых элементов массива;
- указательные — хранят указатели на специальные области памяти (указатель текущей операции, указатель базы, указатель стека);
- сегментные — хранят адреса и селекторы сегментов памяти;
- управляющие — хранят информацию, управляющую состоянием процессора, а также адреса системных таблиц.

В табл. 6.1 показано количество регистров общего назначения в нескольких распространенных архитектурах микропроцессоров.

Стоит отметить, что в некоторых архитектурах использование отдельных регистров может быть осложнено. Так, в SPARC и MIPS регистр номер 0 не сохраняет информацию и всегда считывается как ноль, а в процессорах x86 с регистром ESP (указатель на стек) могут работать лишь некоторые команды.

Таблица 6.1

Количество регистров общего назначения в архитектурах микропроцессоров

Архитектура	Целочисленных регистров	FP регистров	Примечания
x86—32	8	8	
x86—64	16	16	
IBM System/360	16	4	
z/Architecture	16	16	
Itanium	128	128	
SPARC	31	32	Регистр 0 (глобальный) всегда занулен
IBM Cell	4~16	1~4	
IBM POWER	32	32	
Power Architecture	32	32	
Alpha	32	32	
6502	3	0	
W65C816S	5	0	
PIC	1	0	
AVR	32	0	
ARM 32-bit[	16	varies	
ARM 64-bit[	31	32	
MIPS	31	32	Регистр 0 всегда занулен

Регистры архитектуры x86:

- IP (англ. Instruction Pointer) — регистр, указывающий на смещение (адрес) инструкций в сегменте кода (1234:0100h сегмент/смещение);
- IP — 16-битный (младшая часть EIP);
- EIP — 32-битный аналог (младшая часть RIP);
- RIP — 64-битный аналог.

Сегментные регистры — регистры, указывающие на сегменты:

CS (англ. Code Segment), DS (англ. Data Segment), SS (англ. Stack Segment), ES (англ. Extra Segment), FS, GS.

В реальном режиме работы процессора сегментные регистры содержат адрес начала 64 Кб сегмента, смещенный вправо на 4 бита.

В защищенном режиме работы процессора сегментные регистры содержат селектор сегмента памяти, выделенного ОС.

CS — указатель на кодовый сегмент. Связка CS: IP (CS: EIP/CS: RIP — в защищенном/64-битном режиме) указывает на адрес в памяти следующей команды.

Регистры данных — служат для хранения промежуточных вычислений:

- RAX, RCX, RDX, RBX, RSP, RBP, RSI, RDI, R8 — R15—64-битные;
- EAX, ECX, EDX, EBX, ESP, EBP, ESI, EDI, R8D — R15D — 32-битные (extended AX);
- AX (англ. Accumulator), CX (англ. Count Register), DX (англ. Data Register), BX (англ. Base Register), SP (англ. Stack Pointer), BP (англ. Base Pointer), SI (англ. Source Index), DI (англ. Destination Index), R8W — R15W — 16-битные;
- AH, AL, CH, CL, DH, DL, BH, BL, SPL, BPL, SIL, DIL, R8B — R15B — 8-битные (половинки 16-битных регистров).

Например, AH — high AX — старшая половинка 8 бит, AL — low AX — младшая половинка 8 бит.

RAX			RCX			RDX			RBX		
	EAX			ECX			EDX			EBX	
	AX			CX			DX			BX	
	AH	AL		CH	CL		DH	DL		BH	BL

RSP			RBP			RSI			RDI			Rx	
	ESP			EBP			ESI			EDI			RxD
	SP			BP			SI			DI			RxW
		SPL			BPL			SIL			DIL		RxB

Регистры RAX, RCX, RDX, RBX, RSP, RBP, RSI, RDI, Rx, RxD, RxW, RxB, SPL, BPL, SIL, DIL доступны только в 64-битном режиме работы процессора.

Регистр флагов FLAGS (16 бит) / EFLAGS (32 бита) / RFLAGS (64 бита) — содержит текущее состояние процессора.

Системные регистры GDTR, LDTR и IDTR введены в процессорах, начиная с Intel286, и предназначены для хранения базовых адресов таблиц дескрипторов — важнейших составляющих системной архитектуры при работе в защищенном режиме.

Регистр GDTR содержит 32-битный (24-битный для Intel286) базовый адрес и 16-битный предел глобальной таблицы дескрипторов (GDT).

Видимая часть регистра LDTR содержит только селектор дескриптора локальной таблицы дескрипторов (LDT). Сам дескриптор LDT автоматически загружается в скрытую часть LDTR из глобальной таблицы дескрипторов.

Регистр IDTR содержит 32-битный (24-битный для Intel286) базовый адрес и 16-битный предел таблицы дескрипторов прерываний (IDT). В реальном режиме может быть использован для изменения местоположения таблицы векторов прерываний.

Видимая часть регистра TR содержит селектор дескриптора сегмента состояния задачи (TSS). Сам дескриптор TSS автоматически загружается в скрытую часть TR из глобальной таблицы дескрипторов.

Счетчик команд. Счетчик команд (также PC = program counter, IP = instruction pointer, IAR = instruction address register, СЧАК = счетчик адресуемых команд) — регистр процессора, содержащий адрес текущей выполняемой команды. В зависимости от архитектуры содержит либо адрес инструкции, которая будет выполняться, либо той, которая выполняется в данный момент.

В большинстве процессоров после выполнения команды, если она не нарушает последовательности команд (например, команда перехода), счетчик автоматически увеличивается (постинкремент). Понятие счетчика команд сильно связано с фон-неймановской архитектурой, одним из принципов которой является выполнение команд друг за другом в определенной последовательности.

IP (англ. Instruction Pointer) — регистр, содержащий адрес-смещение следующей команды, подлежащей исполнению, относительно кодового сегмента CS в процессорах семейства x86.

Регистр IP связан с CS в виде CS: IP, где CS является текущим кодовым сегментом, а IP — текущим смещением относительно этого сегмента.

Регистр IP является 16-разрядным регистром-указателем. Кроме него в состав регистров этого типа входят SP (англ. Stack Pointer — указатель стека) и BP (англ. Base Pointer — базовый указатель).

Принцип работы. Например, CS содержит значение 2CB5[0]H, в регистре IP хранится смещение 123H.

Адрес следующей инструкции, подлежащей исполнению, вычисляется путем суммирования адреса в CS (сегменте кода) со смещением в регистре IP:

2CB50H + 123H = 2CC73H.

Таким образом, адрес следующей инструкции для исполнения равен 2CC73H.

При выполнении текущей инструкции процессор автоматически изменяет значение в регистре IP, в результате чего регистровая пара CS:IP всегда указывает на следующую подлежащую исполнению инструкцию.

EIP. Начиная с процессора 80386, была введена 32-разрядная версия регистра-указателя — EIP (англ. Extended Instruction Pointer). В данном случае IP является младшей частью этого регистра (первые 16 разрядов). Принцип работы EIP в целом схож с работой регистра IP. Основная разница состоит в том, что в защищенном режиме в отличие от реального режима регистр CS является селектором (селектор указывает не на сам сегмент в памяти, а на дескриптор сегмента в таблице дескрипторов).

RIP. В 64-разрядных процессорах используется свой регистр-указатель инструкций — RIP.

Младшей частью этого регистра является регистр EIP.

На основе RIP в 64-разрядных процессорах введен новый метод адресации RIP-relative. В остальном работа RIP аналогична работе регистра EIP.

Основные понятия языка ассемблера

Синтаксис. Синтаксис языка ассемблера определяется системой команд конкретного процессора.

Набор команд. Типичными командами языка ассемблера являются (большинство примеров даны для Intel-синтаксиса архитектуры x86):

- команды пересылки данных (mov и др.);
- арифметические команды (add, sub, imul и др.);
- логические и побитовые операции (or, and, xor, shr и др.);
- команды управления ходом выполнения программы (jmp, loop, ret и др.);
- команды вызова прерываний (иногда относят к командам управления): int;
- команды ввода-вывода в порты (in, out).

Для микроконтроллеров и микрокомпьютеров характерны также команды, выполняющие проверку и переход по условию, например:

- cjne — перейти, если не равно;
- djnz — декрементировать, и если результат ненулевой, то перейти;
- cfsneq — сравнить, и если не равно, пропустить следующую команду.

Инструкции. Типичный формат записи команд:

[метка:] [[префикс] мнемокод [операнд {, операнд}]] [; комментарий],

где мнемокод — непосредственно мнемоника инструкции процессору. К ней могут быть добавлены префиксы (повторения, изменения типа адресации и пр.).

В качестве операндов могут выступать константы, адреса регистров, адреса в оперативной памяти и пр. Различия между синтаксисом Intel и AT&T касаются в основном порядка перечисления операндов и указания различных методов адресации.

Используемые мнемоники обычно одинаковы для всех процессоров одной архитектуры или семейства архитектур (среди широко известных — мнемоники процессоров и контроллеров x86, ARM, SPARC, PowerPC, M68k). Они описываются в спецификации процессоров. Возможные исключения:

- если ассемблер использует кросс-платформенный AT&T-синтаксис (оригинальные мнемоники приводятся к синтаксису AT&T);
- если изначально существовало два стандарта записи мнемоник (система команд была наследована от процессора другого производителя).

Например, процессор Zilog Z80 наследовал систему команд Intel 8080, расширил ее и поменял мнемоники (и обозначения регистров) на свой лад. Процессоры Motorola Fireball наследовали систему команд Z80, несколько ее урезав. Вместе с тем «Motorola» официально вернулась к мнемоникам Intel, и в данный момент половина ассемблеров для Fireball работает с мнемониками Intel, а половина — с мнемониками Zilog.

Директивы. Программа на языке ассемблера может содержать директивы: инструкции, не переводящиеся непосредственно в машинные команды, а управляющие работой компилятора. Набор и синтаксис их значительно разнятся и зависят не от аппаратной платформы, а от используемого транслятора (порождая диалекты языков в пределах одного семейства архитектур). В качестве основного набора директив можно выделить следующие:

- определение данных (констант и переменных);
- управление организацией программы в памяти и параметрами выходного файла;
- задание режима работы компилятора;
- всевозможные абстракции (т.е. элементы языков высокого уровня) — от оформления процедур и функций (для упрощения

реализации парадигмы процедурного программирования) до условных конструкций и циклов (для парадигмы структурного программирования);
- макросы.

Использование стека. Напомним, что стеком называется область памяти, предназначенная для временного хранения данных. Для этой области в программе выделяется сегмент объемом £ 64 Кбайт, который называется сегментом стека. Для работы со стеком предназначены следующие регистры:
- ss — сегментный регистр стека;
- sp — регистр указателя базы.

Доступ к содержимому стека осуществляется также с помощью регистра BP.

Команды работы со стеком организованы в соответствии с принципом LIFO (последним пришел — первым ушел):

```
push операнд        ; запись операнда в стек
pop операнд         ; чтение операнда из стека
```

Команда push сначала вычитает из регистра sp число 2, уменьшая тем самым адрес начала стека на 2, а затем записывает операнд в начало стека. Команда pop, напротив, сначала считывает слово из начала стека и записывает его в операнд, а затем увеличивает значение регистра SP на 2.

Если операндом является двойное слово, то команда push вычитает из SP число 4, а команда pop увеличивает sp на 4.

Пример. Команды

```
push 100            ; записать число 100 в стек
pop bx              ; извлечь из стека
```

установят bx = 100.

При разработке подпрограмм рекомендуется в начале подпрограммы сохранять значения всех используемых в ней регистров в стек, а в конце — восстанавливать. Сохранение делается с помощью команды

```
pusha       ; сохранить в стек AX, CX, DX, BX, SP, BP, SI, DI
```

Регистры сохраняются в стек в указанном порядке.

Восстановление:

```
popa
```

Слова, выбираемые из стека, размещаются в регистрах: DI, SI, BP, *, BX, DX, CX, AX, где символом * обозначено слово, которое игнорируется, вместо того чтобы быть помещенным в регистр SP. Команды работы со стеком можно использовать для установки флагов:

```
pushf       ; запомнить регистр FL в стек
popf        ; слово извлекается из стека и записывается в FL
```

Команда pushf не изменяет разряды регистр FL, а popf, напротив, записывает в FL новое слово и, значит, воздействует на все флаги.

Примеры программы Hello, world! для разных платформ и разных диалектов:

Пример COM-программы для MS-DOS на диалекте TASM

```
.MODEL TINY
CODE SEGMENT
ASSUME CS: CODE, DS: CODE
ORG 100h
START:
    mov ah,9
    mov dx, OFFSET Msg
    int 21h
    int 20h
    Msg DB'Hello World',13,10,'$'
CODE ENDS
END START
```

Пример EXE-программы для MS-DOS на диалекте TASM

```
.MODEL SMALL
.DATA
    msg DB'Hello World',13,10,'$'
.CODE
START:
    mov ax,@DATA
    mov ds, ax
    mov ax,0900h
    lea dx, msg
    int 21h
    mov ax,4C00h
    int21h
END START
```

Пример программы для Linux/x86 на диалекте NASM

```
SECTION.data
msg: db "Hello, world",10
len: equ$-msg

SECTION.text
global_start
_start:   movedx, len
    mov ecx, msg
    mov ebx,1; stdout
    mov eax,4; write(2)
    int0x80
```

```
        mov ebx,0
    mov eax,1; exit(2)
    int 0x80
```

Пример программы для FreeBSD/x86 на диалекте NASM

```
SECTION.data
msg: db"Hello, world",10
len: equ$-msg

SECTION.text
global_start

syscall:  int 0x80
          ret

_start:   push len
          push msg
          push 1        ; stdout
          moveax,4      ; write(2)
          call syscall
          add esp,3*4
          push 0
          mov eax,1     ; exit(2)
          call syscall
```

Пример программы для Microsoft Windows на диалекте MASM

```
.386
.modelflat, stdcall
optioncasemap: none
include\masm32\include\windows.inc
include\masm32\include\kernel32.inc
includelib\masm32\lib\kernel32.lib

.data
    msgdb"Hello, world",13,10
    lenequ$-msg
.data?
    writtendd?
.code
start:
    push   -11
    call   GetStdHandle

    push   0
    push   OFFSETwritten
    push   len
```

```
    push   OFFSETmsg
    push   eax
    call   WriteFile

    push   0
    call   ExitProcess
endstart
```

Пример консольной программы для Windows на диалекте FASM

```
formatPEconsole
entrystart

include'include\win32a.inc'

section'.data'datareadablewriteable
messagedb'Hello, world!',0
section'.code'codereadableexecutable
start:
; CINVOKE макрос в составеFASM.
; Позволяет вызыватьCDECL-функции.
  cinvokeprintf, message
  cinvokegetch
; INVOKE аналогичный макрос для STDCALL-функций.
  invokeExitProcess,0
section'.idata'importdatareadable
librarykernel,'kernel32.dll',\
    msvcrt,'msvcrt.dll'

import kernel,\
    ExitProcess,'ExitProcess'
importmsvcrt,\
    printf,'printf',\

    getch,'_getch'
```

Пример 64-битной программы для Windows на диалекте YASM (с использованием линковщика от Microsoft)

```
; yasm-1.0.0-win32.exe-fwin64HelloWorld_Yasm.asm
; setenv/Release/x64/xp
; linkHelloWorld_Yasm.objKernel32.libUser32.lib/entry:
main/subsystem: windows/LARGEADDRESSAWARE: NO
bits64

globalmain

externMessageBoxA
```

```
externExitProcess

section.data
mytitdb'The64-bitworldofWindows&assembler…',0
mymsgdb'HelloWorld!',0

section.text
main:
movr9d,0; uType=MB_OK
movr8, mytit; LPCSTRlpCaption
movrdx, mymsg; LPCSTRlpText
movrcx,0; hWnd=HWND_DESKTOP
callMessageBoxA
movecx, eax; uExitCode=MessageBox(…)
callExitProcess

ret
```

Пример программы для Solaris и архитектуры SPARC

```
.section".data"
hello:     .asciz"Hello World!\n"

.section".text"

.align     4
.global    main
main:
save       %sp,–96,%sp  !выделяем память

mov 4,%g1  !4=WRITE(системный вызов)
mov 1,%o0  !1=STDOUT
set hello,%o1
mov 14,%o2 !количество символов
ta  8      !вызов системы

!выход из программы
mov 1,%g1  !move1(exit()syscall)into%g1
mov 0,%o0  !move0(return address)into%o0
ta  8      !вызов системы
```

Пример программы, выдающей сообщение при считывании загрузочного сектора дискеты IBM PC-совместимого компьютера

```
org7C00h
use16
```

```
jmp code
nop

db'hellowrd'
SectSize  dw 00200h
ClustSize db 001h
ResSecs   dw 00001h
FatCnt    db 002h
RootSiz   dw 000E0h
TotSecs   dw 00B40h
Media     db 0F0h
FatSize   dw 00009h
TrkSecs   dw 00012h
HeadCnt   dw 00002h
HidnSec   dw 00000h

code:

cli
mov ax, cs
mov ds, ax
mov ss, ax
mov sp,7c00h
sti
mov ax,0b800h
mov es, ax

mov di,200
mov ah,2
mov bx, MessStr
msg_print:
mov al,[cs: bx]
mov [es: di], ax
inc bx
add di,2
cmp bx, MessEnd
jnz msg_print

loo:
jmploo

MessStr equ $
Message db'Hello, World!'
MessEnd equ $
```

Приведем **примеры** программ (с использованием режимов адресации) для процессоров Intel x86 и MIPS.

```
Программа для as (процессор x86)
.section.data
.section.text
.globl _start
_start:
    # Воспользуемся непосредственной адресацией для
указания того,
    # что программа будет осуществлять запись в стан-
дартное
    # устройство ввода-вывода. Логика работы Linux по-
дразумевает,
    # что цифра четыре в регистре eax говорит компью-
теру произвести
    # запись по адресу файлового дескриптора (он нахо-
дится в регистре ebx).
    # Цифра один в регистре ebx сообщает ядру, что за-
пись будет
    # осуществлена на стандартное устройство вывода
(обычно — это экран
    # терминала). Согласно стандарту POSIX цифра 1 со-
ответствует
    # стандартному устройству вывода (stdout).

        movl $4,%eax
        movl $1,%ebx
    # Прямая адресация. Согласно логике работы ядра,
в регистр ecx должна быть
    # помещена переменная, содержащая текст, который
надо записать в стандартное
    # устройство вывода. В нашей программе этот текст
хранится по адресу переменной
    # $text.

        movl $text,%ecx

    # Далее, согласно логике работы Linux (ядра опера-
ционной системы GNU/Linux),
    # в регистр edx необходимо занести длину печатае-
мого сообщения (число байт,
    # которое оно занимает в памяти). Учитывая, что мы
решили вывести на экран
    # надпись на русском языке в кодировке UTF-8, каж-
дый из кириллических символов
```

```
        # требует по два байта для хранения. В свою оче-
редь, пробел, восклицательный
        # знак и символ новой строки занимают по одному
байту. Отсюда
        # получаем: (10 кириллических символов * 2) + 5 = 25
байт.

                movl $25,%edx

        # Теперь можно осуществить системный вызов, то есть
обращение
        # за помощью к ядру операционной системы.

                int $0x80

        # Завершение работы программы также требует зане-
сения в регистры служебной
        # информации. В частности, в eax должна быть еди-
ница (соответствует выходу
        # из программы), а в ebx должен быть внесен код за-
вершения (возврата). Обычно
        # успешному завершению соответствует код возврата
равный нулю, а единица в
        # коде возврата сообщает об ошибке.

                movl $1,%eax

        # Для разнообразия, прибегнув к регистровой адре-
сации,
        # внесем код возврата, равный единице.

                movl %eax,%ebx

        # Снова выполним системный вызов 0x80.

                int $0x80
  text:
                .ascii «Всем привет!!!\n»
```

Теперь произведем компиляцию и компоновку этой программы:

```
as hello.s -o hello.o
ld hello.o -o hello
./hello
```

Убедиться в том, что код возврата нашей программы равен единице, можно, выполнив (сразу после завершения работы программы hello) в терминале следующую команду:

```
echo $?
```

Более подробную информацию о логике работы системных вызовов Linux можно посмотреть на сайте syscalls.kernelgrok.com

Теперь обсудим адресацию памяти в контексте особенностей процессоров MIPS. Напомним, что MIPS — это представитель архитектуры с сокращенным набором команд (RISC).

Представленная ниже программа для spim (процессор MIPS, эмулируемый программно) иллюстрирует один из вариантов использования временных регистров (temporary registers) $t0 и $t1. Сначала в регистр $t1 загружается единица (непосредственная адресация), потом к единице прибавляется число два и результат записывается в регистр $t0:

```
main:
    li $t1, 1 # непосредственная адресация
    add $t0, $t1, 2 # трехадресный формат команды
    li $v0, 10
    syscall
```

По инструкции syscall выполняется тот или иной системный вызов. Какой именно — зависит от содержимого регистров. Страница MIPS System Calls подробно рассказывает об этом. В частности, наличие в регистре $v0 числа 4 соответствует выводу на печать строки. Если же в $v0 записать число 10, будет выполнен системный вызов, ответственный за выход из программы.

spim является эмулятором процессора MIPS32. Сохранив приведенный выше фрагмент кода в файле с именем mips32addr.asm, его можно загрузить в эмулятор и проверить содержимое нужных регистров:

```
# spim
load «mips32addr.asm»
run
print $t0
```

Для наглядности приведем **пример** программы на языке ассемблера MIPS, печатающей фразу «Доброе утро!».

```
.text
main:
    la $a0, thetext # прямая адресация
li $v0, 4  # непосредственная адресация
syscall
li $v0, 10
syscall
.data
thetext:.asciiz «Доброе утро!\n»
```

Как видно, исходный код программы, представленной выше, содержит две секции:.text и.data. В .text размещаются инструкции, которые нельзя изменить в ходе выполнения программы. Секция .data содержит записи, которые линкер (компоновщик) может занести в память, перед тем как программа будет исполнена.

MIPS поддерживает три формата команд: регистровый (Register type, R-type), непосредственный (Immediate, I-type) и формат перехода (Jump type, J-type). В рамках одного и того же формата доступно использование различных режимов адресации. Напомним, что под режимом адресации понимается метод определения адреса хранения нужной информации.

В документации о режимах адресации MIPS часто используются следующие обозначения:

- op (operation code) — код операции (6 бит);
- rs (source register) — регистр-источник (5 бит);
- rd (destination register) — регистр-приемник (5 бит);
- rt (target register or branch condition) — целевой регистр или условие перехода (5 бит);
- immediate — непосредственное смещение перехода или адреса (16 бит);
- target — цель перехода (26 бит);
- shamt (shift amount) — размер сдвига (5 бит);
- funct (function field) — поле функции (6 бит).

Ассемблер MIPS поддерживает несколько способов работы с операндами при адресации (подробнее см. [Ellard1994, 66]):

- (регистр) — обращение к содержимому регистра;
- константа — постоянный адрес в числовой форме;
- константа (регистр) — сумма постоянного адреса и содержимого регистра;
- символ — адрес размещения символа;
- символ + константа — адрес символа + постоянный адрес в числовой форме;

- символ + константа (регистр) — адрес символа + постоянный адрес в числовой форме + содержимое регистра.

Контрольные вопросы и задания

1. Опишите процедуру связывания программ на разных языках программирования.
2. Напишите программу на языке ассемблера: в регистр AX поместить число 1234, в регистр BX поместить число F0F0, в регистр CX поместить команду логического сложения (код команды 09D8), записать команду в ОЗУ по адресу 10D.
3. Напишите программу для процессора семейства Intel, которая увеличивает число, находящееся в регистре AX, в 10 раз.
4. Опишите последовательность действия процессора на примере временного сохранения трех целочисленных 2-байтовых переменных N1, N2, N3 в стековую память и их восстановление. Указатель стека содержит адрес 4012.
5. Опишите язык ассемблера, его достоинства и недостатки.
6. Перечислите сновные команды языка ассемблера.
7. Какие вы знаете регистры общего назначения?
8. Понятие счетчика команд.
9. Сегментные регистры процессора. Какую информацию содержат сегментные регистры в реальном и защищенном режимах работы процессора?
10. Приведите примеры логических и побитовых операций.
11. Перечислите команды работы со стеком.
12. Приведите примеры команд вызова прерываний команд ввода-вывода в порты.
13. Какие команды работы со стеком можно использовать для установки флагов?

Глава 7
ОРГАНИЗАЦИЯ ВЫЧИСЛИТЕЛЬНЫХ СИСТЕМ

В связи с кризисом классической структуры ЭВМ дальнейшее развитие вычислительной техники привело к идеям построения вычислительных систем.

Вычислительная система отличается от ЭВМ наличием нескольких вычислителей, реализующих параллельную обработку.

Как технические, так и программные средства вычислительной системы имеют модульную структуру построения, позволяющую наращивать ее в зависимости от назначения и условий эксплуатации системы. Как мы уже выясняли, программная автоматизация управления вычислительным процессом осуществляется с помощью операционной системы.

7.1. Классификация вычислительных систем

В гл. 1 рассмотрен наиболее распространенный способ классификации вычислительных систем — классификация М. Дж. Флинна, которая базируется на понятиях двух потоков: команд и данных.

В этой главе рассматриваются ВС, которые по классификации М. Дж. Флинна относятся к группе MIMD (Multiple Instruction, Multiple Data) — системы с множественным потоком команд и множественным потоком данных. К подобному классу относится большинство параллельных многопроцессорных вычислительных систем.

По архитектурным особенностям, заложенным в архитектуру ВС, можно выделить следующие направления:

- векторно-конвейерные суперкомпьютеры;
- симметричные мультипроцессорные системы (SMP);
- системы с массовым параллелизмом (MPP);
- кластерные системы.

Векторно-конвейерные компьютеры. Название этому семейству компьютеров дали два принципа, заложенные в архитектуре процессоров: конвейерная организация обработки потока команд и введение в систему команд набора векторных операций, которые позволяют оперировать с целыми массивами данных. В отличие от традиционного подхода векторные команды оперируют целыми массивами независимых данных, что позволяет эффективно загружать доступные

конвейеры, т.е. команда вида A = B + C может означать сложение двух массивов, а не двух чисел.

Длина одновременно обрабатываемых векторов в современных векторных компьютерах составляет, как правило, 128 или 256 элементов. Очевидно, что векторные процессоры должны иметь гораздо более сложную структуру и, по сути, содержать множество арифметических устройств. Основное назначение векторных операций состоит в распараллеливании выполнения операторов цикла, в которых в основном и сосредоточена большая часть вычислительной работы.

Как правило, несколько специальных векторно-конвейерных процессоров работают одновременно над общей памятью (аналогично SMP) в рамках многопроцессорных конфигураций. Несколько таких узлов могут быть объединены с помощью коммутатора (аналогично MPP).

Представителями этого класса являются компьютеры Cray J90/T90, Cray SV1, NEC SX-4/SX-5.

Симметричные мультипроцессорные системы (Symmetric Multi-Processing, SMP). Характерной чертой многопроцессорных систем SMP архитектуры является то, что все процессоры имеют прямой и равноправный доступ к любой точке общей памяти. Первые SMP системы состояли из нескольких однородных процессоров и массива общей памяти, к которой процессоры подключались через общую системную шину. Однако очень скоро обнаружилось, что такая архитектура непригодна для создания масштабных систем. Первая возникшая проблема — большое число конфликтов при обращении к общей шине. Остроту этой проблемы удалось частично снять разделением памяти на блоки, подключение к которым с помощью коммутаторов позволило распараллелить обращения от различных процессоров. Однако и в таком подходе неприемлемо большими казались накладные расходы для систем более чем с 32 процессорами. Современные системы SMP архитектуры состоят, как правило, из нескольких однородных серийно выпускаемых микропроцессоров и массива общей памяти, подключение к которой производится либо с помощью общей шины, либо с помощью коммутатора. Наличие общей памяти значительно упрощает организацию взаимодействия процессоров между собой и упрощает программирование, поскольку параллельная программа работает в едином адресном пространстве. Этот класс ВС представлен компьютерами SUN StarFire 15K, SGI Origin 3000, HP Superdome.

Системы с массовым параллелизмом (Massively Parallel Processing, MPP). Проблемы, присущие многопроцессорным системам с общей

памятью, простым и естественным образом устраняются в системах с массовым параллелизмом. Компьютеры этого типа представляют собой многопроцессорные системы с распределенной памятью, в которых с помощью некоторой коммуникационной среды объединяются однородные вычислительные узлы. Каждый из узлов состоит из одного или нескольких процессоров, собственной оперативной памяти, коммуникационного оборудования, подсистемы ввода-вывода, т.е. обладает всем необходимым для независимого функционирования. При этом на каждом узле может функционировать либо полноценная операционная система, либо урезанный вариант, поддерживающий только базовые функции ядра, а полноценная ОС работает на специальном управляющем компьютере. Процессоры в таких системах имеют прямой доступ только к своей локальной памяти. Доступ к памяти других узлов реализуется обычно с помощью механизма передачи сообщений. Этот класс ВС представлен следующими компьютерами: Cray T3D/T3E, nCUBE2, IntelParagon.

Кластерные системы. Кластерные технологии стали логическим продолжением развития идей, заложенных в архитектуре MPP-систем. Если процессорный модуль в MPP-системе представляет собой законченную вычислительную систему, то следующий шаг напрашивается сам собой: почему бы в качестве таких вычислительных узлов не использовать обычные серийно выпускаемые компьютеры. Развитие коммуникационных технологий, а именно появление высокоскоростного сетевого оборудования и специального программного обеспечения, реализующего механизм передачи сообщений над стандартными сетевыми протоколами, сделало кластерные технологии общедоступными. Сегодня не составляет большого труда создать небольшую кластерную систему, объединив вычислительные мощности компьютеров отдельной лаборатории или учебного класса. Привлекательной чертой кластерных технологий является то, что они позволяют для достижения необходимой производительности объединять в единые вычислительные системы компьютеры самого разного типа, начиная от персональных компьютеров и заканчивая мощными суперкомпьютерами.

7.2. Векторные и векторно-конвейерные вычислительные системы

В средствах векторной обработки под **вектором** понимается одномерный массив однотипных данных (обычно в форме с плавающей

запятой), регулярным образом размещенных в памяти ВС. Если обработке подвергаются многомерные массивы, их также рассматривают как векторы. Такой подход допустим, если учесть, каким образом многомерные массивы хранятся в памяти ВМ. Пусть имеется массив данных A, представляющий собой прямоугольную матрицу размерности 4×5 (рис. 7.1)

a_{11}	a_{12}	a_{13}	a_{14}	a_{15}
a_{21}	a_{22}	a_{23}	a_{24}	a_{25}
a_{31}	a_{32}	a_{33}	a_{34}	a_{35}
a_{41}	a_{42}	a_{43}	a_{44}	a_{45}

Рис. 7.1. Прямоугольная матрица данных

При размещении матрицы в памяти все ее элементы заносятся в ячейки с последовательными адресами, причем данные могут быть записаны строка за строкой или столбец за столбцом (рис. 7.2). С учетом такого размещения многомерных массивов в памяти вполне допустимо рассматривать их как векторы и ориентировать соответствующие вычислительные средства на обработку одномерных массивов данных (векторов).

Рис. 7.2. Способы размещения в памяти матрицы 4×5

Действия над многомерными массивами имеют свою специфику. В двумерном массиве обработка может вестись как по строкам, так и по столбцам. Это выражается в том, с каким шагом должен меняться адрес очередного выбираемого из памяти элемента. Если рассмотренная в примере матрица расположена в памяти *построчно*, то адреса последовательных элементов строки различаются на единицу, а для элементов столбца шаг равен пяти. При размещении матрицы *по столбцам* единице будет равен шаг по столбцу, а шаг по строке — четырем. В векторной концепции для обозначения шага, с которым элементы вектора извлекаются из памяти, применяется термин «шаг по индексу» (stride).

Еще одной характеристикой вектора является число составляющих его элементов — длина вектора.

Векторно-конвейерные вычислительные системы относятся к классу SIMD-систем. Основные принципы, заложенные в архитектуру векторно-конвейерных систем:

- конвейерная организация обработки потока команд;
- введение в систему команд набора векторных операций, которые позволяют оперировать с целыми массивами данных.

Длина обрабатываемых векторов в современных векторно-конвейерных системах составляет, как правило, 128 или 256 элементов. Основное назначение векторных операций состоит в распараллеливании выполнения операторов цикла, в которых обычно сосредоточена большая часть вычислительной работы.

Первый векторно-конвейерный компьютер Cray-1 появился в 1976 г. Архитектура этого компьютера оказалась настолько удачной, что он положил начало целому семейству компьютеров.

Обобщенная структура векторного процессора приведена на рис. 7.3. На схеме показаны основные узлы процессора без детализации некоторых связей между ними.

Обработка всех n компонентов векторов-операндов задается одной векторной командой. Элементы векторов представляются числами в форме с плавающей запятой (ПЗ). АЛУ векторного процессора может быть реализовано в виде единого конвейерного устройства, способного выполнять все предусмотренные операции над числами с ПЗ. Однако более распространена иная структура, в которой АЛУ состоит из отдельных блоков сложения и умножения, а иногда и блока для вычисления обратной величины, когда операция деления $\frac{X}{Y}$ реализуется в виде $X\left(\frac{1}{Y}\right)$. Каждый из таких блоков также конвейеризирован. Кроме того, в состав векторной вычислительной системы обычно включается и скалярный процессор, что позволяет параллельно выполнять векторные и скалярные команды.

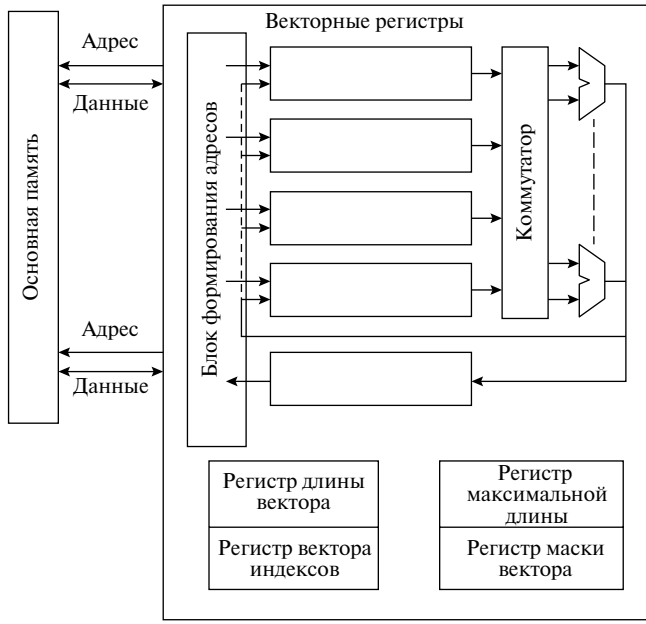


Рис. 7.3. Упрощенная структура векторного процессора

Для хранения векторов-операндов вместо множества скалярных регистров используются векторные регистры, представляющие собой совокупность скалярных регистров, объединенных в очередь типа FIFO, способную хранить 50—100 чисел с плавающей запятой. Набор векторных регистров (V_a, V_b, V_c,...) имеется в любом векторном процессоре. Система команд векторного процессора поддерживает работу с векторными регистрами и обязательно включает в себя команды:

- загрузки векторного регистра содержимым последовательных ячеек памяти, указанных адресом первой ячейки этой последовательности;
- выполнения операций над всеми элементами векторов, находящихся в векторных регистрах;
- сохранения содержимого векторного регистра в последовательности ячеек памяти, указанных адресом первой ячейки этой последовательности.

Примером одной из наиболее распространенных операций, возлагаемых на векторный процессор, может служить операция перемножения матриц. Рассмотрим перемножение двух матриц **A** и **B** размерности 3×3:

$$\begin{bmatrix} c_{11} & c_{12} & c_{13} \\ c_{21} & c_{22} & c_{23} \\ c_{31} & c_{32} & c_{33} \end{bmatrix} = \begin{bmatrix} a_{11} & a_{12} & a_{13} \\ a_{21} & a_{22} & a_{23} \\ a_{31} & a_{32} & a_{33} \end{bmatrix} \times \begin{bmatrix} b_{11} & b_{12} & b_{13} \\ b_{21} & b_{22} & b_{23} \\ b_{31} & b_{32} & b_{33} \end{bmatrix}.$$

Элементы матрицы результата C связаны с соответствующими элементами исходных матриц **A** и **B** операцией скалярного произведения:

$$c_{ij} = \sum_{k=1}^{3} a_{ik} \times b_{kj}.$$

Так, элемент c_{11} вычисляется как

$$c_{11} = a_{11} \times b_{11} + a_{12} \times b_{21} + a_{13} \times b_{31}.$$

Это требует трех операций умножения и после инициализации c_{11} нулем — трех операций сложения. Общее число умножений и сложений для рассматриваемого примера составляет $9 \times 3 = 27$. Если рассматривать связанные операции умножения и сложения как одну кумулятивную операцию $c + a \times b$, то для умножения двух матриц $n \times n$ необходимо n^3 операций типа «умножение—сложение». Вся процедура сводится к получению n^2 скалярных произведений, каждое из которых является итогом n операций «умножение—сложение», учитывая, что перед вычислением каждого элемента c_{ij} его необходимо обнулить. Таким образом, скалярное произведение состоит из k членов:

$$\mathbf{C} = \mathbf{A}_1\mathbf{B}_1 + \mathbf{A}_2\mathbf{B}_2 + \mathbf{A}_3\mathbf{B}_3 + \mathbf{A}_4\mathbf{B}_4 + ... + \mathbf{A}_k\mathbf{B}_k.$$

Векторный процессор с конвейеризированными блоками обработки для вычисления скалярного произведения показан на рис. 7.4.

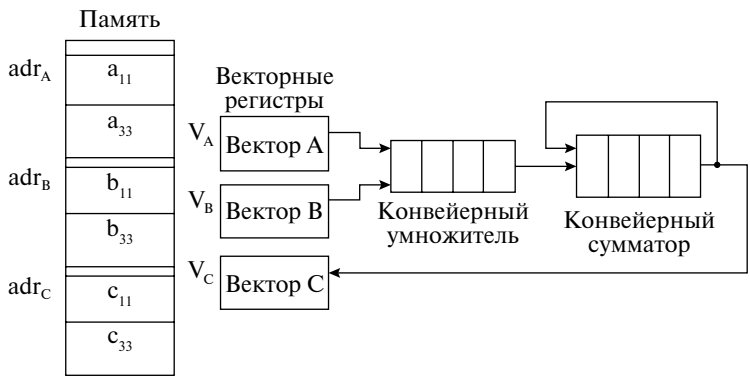


Рис. 7.4. Векторный процессор для вычисления скалярного произведения

Векторы **A** и **B**, хранящиеся в памяти начиная с адресов adr_A и adr_B, загружаются в векторные регистры V_A и V_B соответственно. Предполагается, что конвейерные умножитель и сумматор состоят из четырех сегментов, которые вначале инициализируются нулем, поэтому в течение первых восьми циклов, пока оба конвейера не заполнятся, на выходе сумматора будет ноль. Пары ($\mathbf{A}_i$, $\mathbf{B}_i$) подаются на вход умножителя и перемножаются в темпе одна пара за цикл. После первых четырех циклов произведения начинают суммироваться с данными, поступающими с выхода сумматора. В течение следующих четырех циклов на вход сумматора поступают суммы произведений из умножителя с нулем. К концу восьмого цикла в сегментах сумматора находятся четыре первых произведения $\mathbf{A}_1\mathbf{B}_1$,..., $\mathbf{A}_4\mathbf{B}_4$, а в сегментах умножителя — следующие четыре произведения: $\mathbf{A}_5\mathbf{B}_5$,..., $\mathbf{A}_8\mathbf{B}_8$. К началу девятого цикла на выходе сумматора будет $\mathbf{A}_1\mathbf{B}_1$, а на выходе умножителя — $\mathbf{A}_5\mathbf{B}_5$. Таким образом, девятый цикл начнется со сложения в сумматоре $\mathbf{A}_1\mathbf{B}_1$и $\mathbf{A}_5\mathbf{B}_5$. Десятый цикл начнется со сложения $\mathbf{A}_2\mathbf{B}_2 + \mathbf{A}_6\mathbf{B}_6$ и т.д. Процесс суммирования в четырех секциях выглядит так:

$$\mathbf{C} = \mathbf{A}_1\mathbf{B}_1 + \mathbf{A}_5\mathbf{B}_5 + \mathbf{A}_9\mathbf{B}_9 + \mathbf{A}_{13}\mathbf{B}_{13} +$$
$$+ \mathbf{A}_2\mathbf{B}_2 + \mathbf{A}_6\mathbf{B}_6 + \mathbf{A}_{10}\mathbf{B}_{10} + \mathbf{A}_{14}\mathbf{B}_{14} +$$
$$+ \mathbf{A}_3\mathbf{B}_3 + \mathbf{A}_7\mathbf{B}_7 + \mathbf{A}_{11}\mathbf{B}_{11} + \mathbf{A}_{15}\mathbf{B}_{15} +$$
$$+ \mathbf{A}_4\mathbf{B}_4 + \mathbf{A}_8\mathbf{B}_8 + \mathbf{A}_{12}\mathbf{B}_{12} + \mathbf{A}_{16}\mathbf{B}_{16} +$$

Когда больше не остается членов для сложения, система заносит в умножитель четыре нуля. При этом в четырех сегментах конвейера сумматора содержатся четыре скалярных произведения, соответствующие четырем суммам, приведенным в четырех строках показанного выше уравнения. Далее четыре частичные суммы складываются для получения окончательного результата.

Программа для вычисления скалярного произведения векторов **A** и **B**, хранящихся в областях памяти с начальными адресами adr_A и adr_B, имеет вид:

V_load V_A, adr_A;
V_load V_B, adr_B;
V_multiplyV_C, V_A, V_B.

Первые две векторные командыV_load загружают векторы из памяти в векторные регистры V_A и V_B. Векторная команда умножения V_multiply вычисляет произведение для всех пар одноименных элементов векторов и записывает полученный вектор в векторный регистр V_C.

Важным элементом векторного процессора (ВП) является *регистр длины вектора*. Этот регистр определяет, сколько элементов фактически содержит обрабатываемый в данный момент вектор, т.е. сколько индивидуальных операций с элементами нужно сделать. В некоторых ВП присутствует также *регистр максимальной длины вектора*, определяющий максимальное число элементов вектора, которое может быть одновременно обработано аппаратурой процессора. Этот регистр используется при разделении очень длинных векторов на сегменты, длина которых соответствует максимальному числу элементов, обрабатываемых аппаратурой за один прием.

Часто приходится выполнять такие операции, в которых участвуют не все элементы векторов. Векторный процессор обеспечивает данный режим с помощью *регистра маски вектора*. В этом регистре каждому элементу вектора соответствует один бит. Установка бита в единицу разрешает запись соответствующего элемента вектора результата в выходной векторный регистр, а сброс в ноль — запрещает.

Элементы векторов в памяти расположены регулярно, и при выполнении векторных операций достаточно указать значение шага по индексу. Существуют случаи, когда необходимо обрабатывать только ненулевые элементы векторов. Для поддержки подобных операций в системе команд ВП предусмотрены операции *упаковки/распаковки* (gather/scatter). Операция упаковки формирует вектор, содержащий только ненулевые элементы исходного вектора, а операция распаковки производит обратное преобразование. Обе эти задачи векторный процессор решает с помощью вектора индексов, для хранения которого используется *регистр вектора индексов*, по структуре аналогичный регистру маски. В векторе индексов каждому элементу исходного вектора соответствует один бит. Нулевое значение бита свидетельствует, что соответствующий элемент исходного вектора равен нулю.

Применение векторных команд окупается по двум причинам. Во-первых, вместо многократной выборки одних и тех же команд достаточно произвести выборку только одной векторной команды, что позволяет сократить издержки за счет устройства управления и уменьшить требования к пропускной способности памяти. Во-вторых, векторная команда обеспечивает процессор упорядоченными данными. Когда инициируется векторная команда, ВС знает, что ей нужно извлечь n пар операндов, расположенных в памяти регулярным образом. Таким образом, процессор может указать памяти на необходимость начать извлечение таких пар. Если используется память с чередованием адресов, эти пары могут быть получены со скоростью одной пары за цикл процессора и направлены для обработки

в конвейеризированный функциональный блок. При отсутствии чередования адресов или других средств извлечения операндов с высокой скоростью преимущества обработки векторов существенно снижаются.

Структуры типа «память—память» и «регистр—регистр»

Принципиальное различие архитектур векторных процессоров проявляется в том, каким образом осуществляется доступ к операндам. При организации «*память—память*» элементы векторов поочередно извлекаются из памяти и сразу же направляются в функциональный блок. По мере обработки получающиеся элементы вектора результата сразу же заносятся в память. В архитектуре типа «*регистр—регистр*» операнды сначала загружаются в *векторные регистры*, каждый из которых может хранить сегмент вектора (например, 64 элемента). Векторная операция реализуется путем извлечения операндов из векторных регистрови занесения результата в векторный регистр.

Преимущество ВП с режимом «*память—память*» состоит в возможности обработки длинных векторов, в то время как в процессорах типа «*регистр—регистр*» приходится разбивать длинные векторы на сегменты фиксированной длины. К сожалению, за гибкость режима «*память—память*» приходится расплачиваться относительно большим *временем запуска*, представляющим собой временной интервал между инициализацией команды и моментом, когда первый результат появится на выходе конвейера. Большое время запуска в процессорах типа «*память—память*» обусловлено скоростью доступа к памяти, которая намного меньше скорости доступа к внутреннему регистру. Однако, когда конвейер заполнен, результат формируется в каждом цикле. Модель времени работы векторного процессора имеет вид:

$$T = s + \alpha \times N,$$

где s — время запуска, α — константа, зависящая от команды (обычно 1/2, 1 или 2) и N — длина вектора.

Архитектура типа «*память—память*» реализована в векторно-конвейерных ВС Advanced Scientific Computer фирмы «Texas Instruments Inc.», семействе вычислительных систем фирмы «Control Data Corporation», прежде всего Star 100, серии Cyber 200 и ВС типа ETA-10. Все эти вычислительные системы появились в середине 70-х гг. прошлого века после длительного цикла разработки, но к середине 80-х гг. от них отказались. Причиной послужило слишком большое время запуска — порядка 100 циклов процессора. Это означает, что операции с короткими векторами выполняются очень неэффек-

тивно, и даже при длине векторов в 100 элементов процессор достигал только половины потенциальной производительности.

В вычислительных системах типа «*регистр—регистр*» векторы имеют сравнительно небольшую длину (в ВС семейства Cray — 64), но время запуска значительно меньше, чем в случае «*память—память*». Этот тип векторных систем гораздо более эффективен при обработке коротких векторов, но при операциях над длинными векторами векторные регистры должны загружаться сегментами несколько раз. В настоящее время ВП типа «*регистр—регистр*» доминируют на компьютерном рынке.

Обработка длинных векторов и матриц

Аппаратура векторных процессоров типа «*регистр—регистр*» ориентирована на обработку векторов, длина которых совпадает с длиной векторных регистров (ВР), поэтому обработка коротких векторов не вызывает проблем — достаточно записать фактическую длину вектора в регистр длины вектора.

Если размер векторов превышает емкость ВР, используется техника разбиения исходного вектора на сегменты одинаковой длины, совпадающей с емкостью векторных регистров (последний сегмент может быть короче), и последовательной обработки полученных сегментов. В английском языке этот прием называется *strip-mining*. Процесс разбиения обычно происходит на стадии компиляции, но в ряде ВП данная процедура производится по ходу вычислений с помощью аппаратных средств на основе информации, хранящейся в регистре максимальной длины вектора.

Ускорение вычислений

Для повышения скорости обработки векторов все функциональные блоки векторных процессоров строятся по конвейерной схеме, причем так, чтобы каждая ступень любого из конвейеров справлялась со своей операцией за один такт (число ступеней в разных функциональных блоках может быть различным). С этой целью в некоторых векторных ВС (например, Cray C90) конвейеры во всех функциональных блоках продублированы (рис. 7.5).

На конвейер 0 всегда подаются элементы векторов с четными номерами, а на конвейер 1 — с нечетными. В начальный момент на первую ступень конвейера 0 из ВР V_1 и V_2 поступают нулевые элементы векторов. Одновременно первые элементы векторов из этих регистров подаются на первую ступень конвейера 1. На следующем такте на конвейер 0 подаются вторые элементы из V_1 и V_2, а на конвейер 1 — третьи

элементы и т.д. Аналогично происходит распределение результатов в выходном векторном регистре V_3. В итоге функциональный блок при максимальной загрузке в каждом такте выдает не один результат, а два. В скалярных операциях работает только конвейер 0.

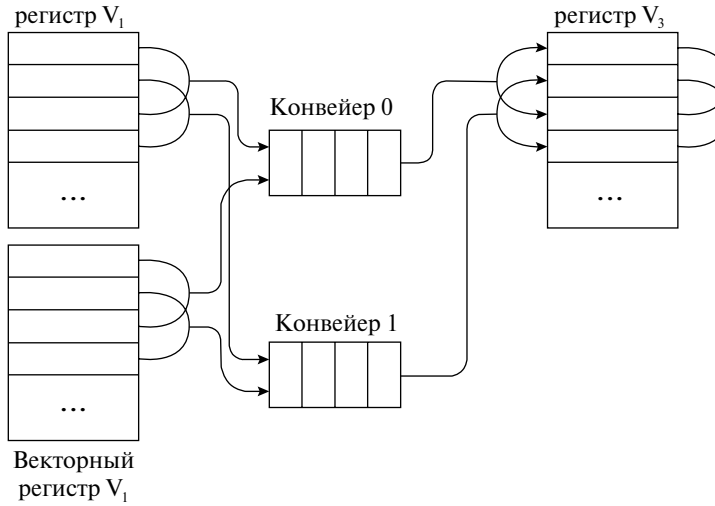


Рис. 7.5. Выполнение векторных операций при двух конвейерах

Интересной особенностью некоторых ВП типа «*регистр—регистр*» (например, ВС фирмы «CrayResearchInc.») является *сцепление векторов* (vector chaining или vector linking), когда ВР результата одной векторной операции используется в качестве входного регистра для последующей векторной операции. Для примера рассмотрим последовательность из двух векторных команд, предполагая, что длина векторов равна 64:

$$\mathbf{V}_2 = \mathbf{V}_0 \times \mathbf{V}_1, \mathbf{V}_4 = \mathbf{V}_2 + \mathbf{V}_3.$$

Результат первой команды служит операндом для второй. Первая векторная команда должна послать в конвейерный умножитель до 64 пар чисел. Примерно в середине выполнения команды складывается ситуация, когда несколько начальных элементов вектора $\mathbf{V}_2$ будут уже содержать недавно вычисленные произведения; часть элементов $\mathbf{V}_2$ все еще будет находиться в конвейере, а оставшиеся элементы операндов $\mathbf{V}_0$ и $\mathbf{V}_1$ еще остаются во входных векторных регистрах, ожидая загрузки в конвейер. Такая ситуация показана на рис. 7.6, где элементы векторов $\mathbf{V}_0$ и $\mathbf{V}_1$, находящиеся в конвейерном

умножителе, имеют темную закраску. В этот момент система извлекает элементы $\mathbf{V}_0[k]$ и $\mathbf{V}_1[k]$ с тем, чтобы направить их на первую ступень конвейера, в то время как $\mathbf{V}_2[j]$ покидает конвейер.

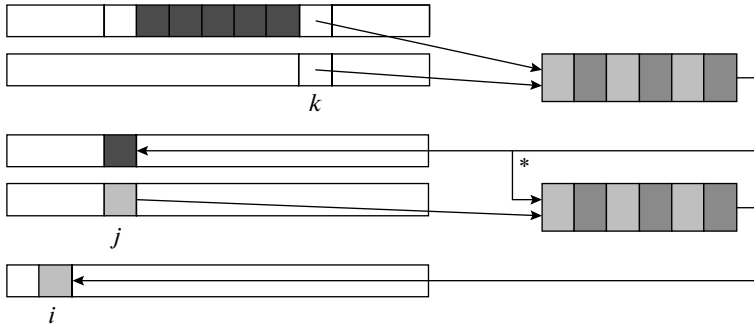


Рис. 7.6. Сцепление векторов

Сцепление векторов иллюстрирует линия, обозначенная звездочкой. Одновременно с занесением $\mathbf{V}_2[j]$ в ВР этот элемент направляется и в конвейерный сумматор, куда также подается и элемент $\mathbf{V}_3[j]$. Как видно из рисунка, выполнение второй команды может начаться до завершения первой, и, поскольку одновременно выполняются две команды, процессор формирует два результата за цикл ($\mathbf{V}_4[i]$ и $\mathbf{V}_2[j]$) вместо одного. Без сцепления векторов пиковая производительность Cray-1 была бы 80 MFLOPS (один полный конвейер производит результат каждые 12,5 нс). При сцеплении трех конвейеров теоретический пик производительности — 240 MFLOPS. В принципе сцепление векторов можно реализовать и в векторных процессорах типа «*память—память*», но для этого необходимо повысить пропускную способность памяти. Без сцепления необходимы три «канала»: два для входных потоков операндов и один — для потока результата. При использовании сцепления требуется обеспечить пять каналов: три входных и два выходных.

С середины 90-х гг. прошлого века векторно-конвейерные ВС стали уступать свои позиции другим более технологичным видам систем.

7.3. Симметричные мультипроцессорные системы

Симметричная мультипроцессорная система (SMP, Symmetric Multiprocessor) — это вычислительная система, обладающая следующими характеристиками:

- Имеется два или более процессора сопоставимой производительности.
- Процессоры совместно используют основную память и работают в едином виртуальном и физическом адресном пространстве.
- Все процессоры связаны между собой посредством шины или по иной схеме, так что время доступа к памяти для любого из них одинаково.
- Все процессоры разделяют доступ к устройствам ввода-вывода либо через одни и те же каналы, либо через разные каналы, обеспечивающие доступ к одному и тому же внешнему устройству.
- Все процессоры способны выполнять одинаковые функции.
- Любой из процессоров может обслуживать внешние прерывания.
- Вычислительная система управляется интегрированной операционной системой, которая организует и координирует взаимодействие между процессорами и программами на уровне заданий, задач, файлов и элементов данных.

В отличие от слабо связанных мультипроцессорных систем (кластеров), где в качестве физической единицы обмена информацией выступает сообщение или полный файл, в SMP допустимо взаимодействие на уровне отдельного элемента данных. Благодаря этому достигается высокий уровень связности между процессами.

Хотя технически SMP-системы симметричны, в их работе присутствует небольшой фактор перекоса, который вносит программное обеспечение. На время загрузки системы один из процессоров получает статус ведущего (master). Это не означает, что позже, во время работы какие-то процессоры будут ведомыми — все они в SMP-системе равноправны. Термин «ведущий» относится к тому процессору, который по умолчанию будет руководить первоначальной загрузкой ВС.

Операционная система планирует процессы или нити процессов (threads) сразу по всем процессорам, скрывая при этом от пользователя многопроцессорный характер SMP-архитектуры.

По сравнению с однопроцессорными схемами SMP-системы имеют преимущество по следующим показателям.

Производительность. Если подлежащая решению задача поддается разбиению на несколько частей так, что отдельные части могут выполняться параллельно, то множество процессоров дает выигрыш в производительности относительно одиночного процессора того же типа (рис. 7.7).

Готовность. В симметричном мультипроцессоре отказ одного из компонентов не ведет к отказу системы, поскольку любой из процессоров в состоянии выполнять те же функции, что и другие.

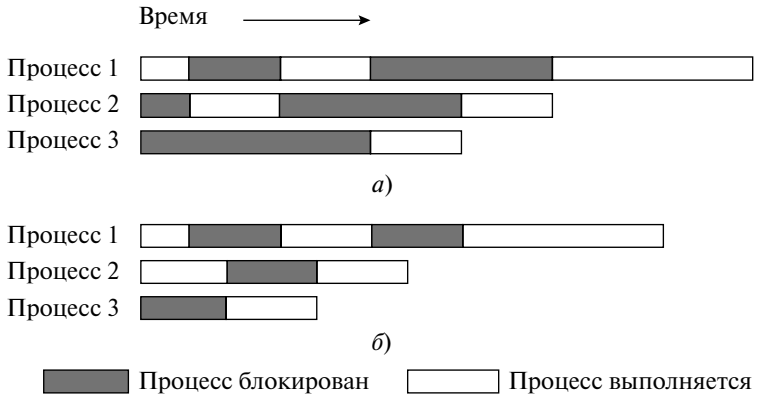


Рис. 7.7. Мультипрограммирование и мультипроцессорная обработка: *а* — мультипрограммирование; *б* — мультипроцессорная обработка

Расширяемость. Производительность системы может быть увеличена добавлением дополнительных процессоров.

Масштабируемость. Варьируя число процессоров в системе, можно создать системы различной производительности и стоимости.

На рис. 7.8 в самом общем виде показана архитектура симметричной мультипроцессорной ВС.

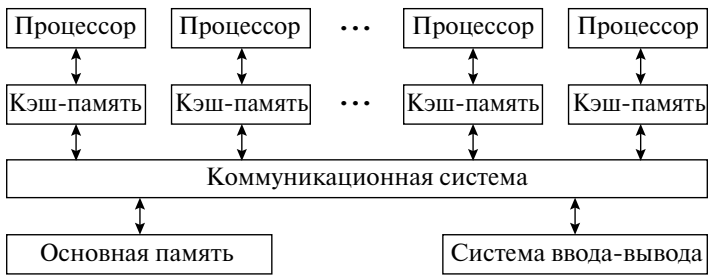
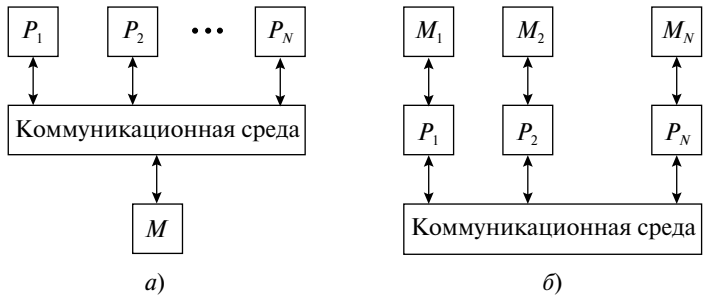


Рис. 7.8. Организация симметричной мультипроцессорной системы

Как уже отмечалось, MIMD-система содержит много процессоров, которые (как правило, асинхронно) выполняют разные команды над разными данными. Подавляющее большинство современных высокопроизводительных ЭВМ на верхнем уровне иерархии имеют архитектуру MIMD.

Для MIMD-систем в настоящее время общепризнана классификация, основанная на используемых способах организации оператив-

ной памяти в этих системах. По этой классификации прежде всего различают мультипроцессорные вычислительные системы (или мультипроцессоры) или вычислительные системы с разделяемой памятью (multiprocessors, common memory systems, shared-memory systems) и мультикомпьютерные вычислительные системы (мультикомпьютеры) или вычислительные системы с распределенной памятью (multicomputers, distributed memory systems). Структура мультипроцессорной и мультикомпьютерной систем приведена на рис. 7.9, где P_i — процессор, M_i — модуль памяти.



а) *б*)

Рис. 7.9:

а — структура мультипроцессора; *б* — структура мультикомпьютера

Мультипроцессоры. В мультипроцессорах адресное пространство всех процессоров является единым. Это значит, что если в программах нескольких процессоров мультипроцессора встречается одна и та же переменная, то для получения или изменения значения этой переменной эти процессоры будут обращаться в одну физическую ячейку общей памяти. Это обстоятельство имеет как положительные, так и отрицательные последствия.

С одной стороны, не нужно физически перемещать данные между коммутирующими программами, что исключает затраты времени на межпроцессорный обмен.

С другой стороны, так как одновременное обращение нескольких процессоров к общим данным может привести к получению неверных результатов, необходимы системы синхронизации параллельных процессов и обеспечения когерентности памяти. Поскольку процессорам необходимо очень часто обращаться к общей памяти, требования к пропускной способности коммуникационной среды чрезвычайно высоки.

Последнее обстоятельство ограничивает число процессоров в мультипроцессорах несколькими десятками. Остроту проблемы

доступа к общей памяти частично удается снять разделением памяти на блоки, которые позволяют распараллелить обращения к памяти от различных процессоров.

Отметим еще одно преимущество мультипроцессоров — мультипроцессорная система функционирует под управлением единственной копии операционной системы (обычно UNIX-подобной) и не требует индивидуальной настройки каждого процессорного узла.

Однородные мультипроцессоры с равноправным (симметричным) доступом к общей оперативной памяти принято называть SMP-системами (системами с симметричной мультипроцессорной архитектурой). SMP-системы появились как альтернатива дорогим мультипроцессорным системам на базе векторно-конвейерных процессоров и векторно-параллельных процессоров.

Мультикомпьютеры. Вследствие простоты своей архитектуры наибольшее распространение в настоящее время получили мультикомпьютеры. Мультикомпьютеры не имеют общей памяти. Поэтому межпроцессорный обмен в таких системах осуществляется обычно через коммуникационную сеть с помощью передачи сообщений.

Каждый процессор в мультикомпьютере имеет независимое адресное пространство. Поэтому наличие переменной с одним и тем же именем в программах разных процессоров приводит к обращению к физически разным ячейкам собственной памяти этих процессоров. Это обстоятельство требует физического перемещения данных между коммутирующими программами в разных процессорах. Чаще всего основная часть обращений производится каждым процессором к собственной памяти. Поэтому требования к коммутационной среде ослабляются. В результате число процессоров в мультикомпьютерных системах может достигать нескольких тысяч, десятков тысяч и даже сотен тысяч.

Пиковая производительность крупнейших систем с общей памятью ниже пиковой производительности крупнейших систем с распределенной памятью; стоимость систем с общей памятью выше стоимости аналогичных по производительности систем с распределенной памятью.

Однородные мультикомпьютеры с распределенной памятью называются вычислительными системами с массивно-параллельной архитектурой (MPP-системами).

Нечто среднее между SMP-системами и MPP-системами представляют собой NUMA-системы.

7.4. Кластерные вычислительные системы

Одно из самых современных направлений в области создания вычислительных систем — это кластеризация. По производительности и коэффициенту готовности кластеризация представляет собой альтернативу симметричным мультипроцессорным системам.

Как мы уже отмечали, вычислительный кластер — это MIMD-система (мультикомпьютер), состоящая из множества отдельных компьютеров (узлов вычислительного кластера), объединенных единой коммуникационной средой.

В качестве узла кластера может выступать как однопроцессорная ВМ, так и ВС типа SMP или MPP. Каждый узел в состоянии функционировать самостоятельно и отдельно от кластера.

Каждый узел имеет свою локальную оперативную память. При этом общей физической оперативной памяти для узлов, как правило, не существует. Коммуникационная среда вычислительных кластеров обычно позволяет узлам взаимодействовать между собой только посредством передачи сообщений. В целом, вычислительный кластер следует рассматривать как единую аппаратно-программную систему, имеющую единую коммуникационную систему, единый центр управления и планирования загрузки.

Архитектура кластерных вычислений сводится к объединению нескольких узлов высокоскоростной сетью. Наряду с термином «кластерные вычисления» часто применяются такие названия, как кластер рабочих станций (workstation cluster), гипервычисления (hypercomputing), параллельные вычисления на базе сети (network-based concurrent computing).

Перед кластерами ставятся две задачи:

- достичь большой вычислительной мощности;
- обеспечить повышенную надежность ВС.

Первый коммерческий кластер создан корпорацией DEC в начале 80-х гг. прошлого века.

В качестве узлов кластеров могут использоваться как одинаковые ВС (гомогенные кластеры), так и разные (гетерогенные кластеры). По своей архитектуре кластерная ВС является слабосвязанной системой.

Преимущества, достигаемые с помощью кластеризации:

- Абсолютная масштабируемость. Возможно создание больших кластеров, превосходящих по вычислительной мощности даже самые производительные одиночные ВМ. Кластер в состоянии содержать десятки узлов, каждый из которых представляет собой мультиплексор.

- Наращиваемая масштабируемость. Кластер строится так, что его можно наращивать, добавляя новые узлы небольшими порциями.
- Высокий коэффициент готовности. Поскольку каждый узел кластера — самостоятельная ВМ или ВС, отказ одного из узлов не приводит к потере работоспособности кластера. Во многих системах отказоустойчивость автоматически поддерживается программным обеспечением.
- Превосходное соотношение цена/производительность. Кластер любой производительности можно создать, соединяя стандартные ВМ, при этом его стоимость будет ниже, чем у одиночной ВМ с эквивалентной вычислительной мощностью.

На уровне аппаратного обеспечения кластер — это просто совокупность независимых вычислительных систем, объединенных сетью. При соединении машин в кластер почти всегда поддерживаются прямые межмашинные связи. Решения могут быть простыми, основывающимися на аппаратуре Ethernet, или сложными с высокоскоростными сетями с пропускной способностью в сотни мегабайтов в секунду (система RS/6000 SP компании IBM, системы фирмы «Digital» на основе Memory Channel, ServerNet корпорации «Compaq»).

Узлы кластера контролируют работоспособность друг друга и обмениваются специфической информацией. Контроль работоспособности осуществляется с помощью специального сигнала, называемого heartbeat («сердцебиение»). Этот сигнал передается узлами кластера друг другу, чтобы подтвердить их нормальное функционирование.

Неотъемлемой частью кластера является специализированное программное обеспечение (ПО), на которое возлагается задача обеспечения бесперебойной работы при отказе одного или нескольких узлов. Такое ПО производит перераспределение вычислительной нагрузки при отказе одного или нескольких узлов кластера, а также восстановление вычислений при сбое в узле. Кроме того, при наличии в кластере совместно используемых дисков кластерное программное обеспечение поддерживает единую файловую систему.

Узлы вычислительного кластера могут функционировать под управлением разных операционных систем. Однако чаще всего используются стандартные UNIX-подобные системы. Заметим, что с точки зрения разработки прикладных параллельных программ нет каких-либо принципиальных различий между однородными вычислительными кластерами и MPP-системами.

Классификация вычислительных кластеров по типу узловых процессоров

Вычислительные кластеры классифицируются прежде всего по характеру узловых процессоров (см. рис. 7.10).

В качестве узлов вычислительного кластера обычно используют персональные компьютеры, рабочие станции и SMP-серверы. Если в качестве узла кластера используется SMP-система, то такой вычислительный кластер называется SMP-кластером.

Если в качестве узлов вычислительного кластера используются персональные ЭВМ или рабочие станции, то обычной является ситуация, когда во время решения задачи на кластере на узлах этого кластера продолжают выполняться последовательные задания пользователей. В результате относительная производительность узлов кластера меняется случайным образом и в широких пределах. Решением проблемы было бы написание самоадаптирующейся пользовательской программы. Однако эффективное решение этой задачи представляется весьма проблематичным. Ситуация усугубляется, если среди узловых компьютеров вычислительного кластера имеются файловые серверы. При этом во время решения задачи на кластере в широких пределах может меняться загрузка коммуникационной среды, что делает непредсказуемыми коммуникационные расходы задачи.

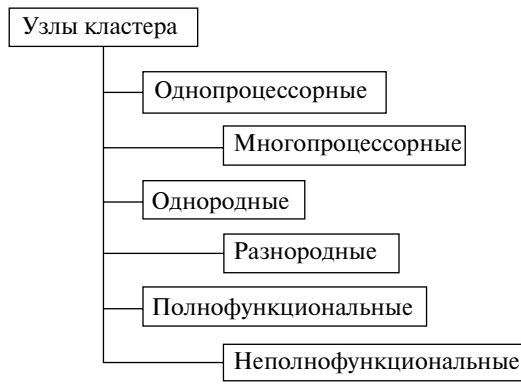


Рис. 7.10. Классификация узлов вычислительных кластеров

Классификация вычислительных кластеров по однородности узлов

Как и всякие MIMD-системы, вычислительные кластеры разделяются на однородные кластерные системы (однородные вычисли-

тельные кластеры) и гетерогенные кластерные системы (гетерогенные вычислительные кластеры).

Обычно, когда говорят о вычислительных кластерах, подразумевают однородные вычислительные кластеры. Однако часто при наращивании кластера приходится использовать процессоры, отличающиеся не только по производительности, но и по архитектуре от узловых процессоров кластера. Поэтому постепенно однородный вычислительный кластер может стать неоднородным. Эта неоднородность создает следующие проблемы. Различие в производительности процессоров усложняет задачу распределения работ между процессорами. Различие в архитектуре процессоров требует подготовки разных выполняемых файлов для разных узлов, а в случае различий в представлении данных может потребовать и преобразования их форматов при передаче сообщений между узлами.

Классификация вычислительных кластеров по функциональности узлов

Узлы вычислительного кластера могут представлять собой полнофункциональные компьютеры, которые могут работать и как самостоятельные единицы. Производительность такого кластера обычно невысока.

Для создания высокопроизводительных вычислительных кластеров системные блоки узловых компьютеров делают значительно более простыми, чем в первом случае (неполнофункциональными). Здесь нет необходимости снабжать компьютеры узлов графическими картами, мониторами, дисковыми накопителями и другим периферийным оборудованием. Периферийное оборудование устанавливается только на одном или немногих управляющих компьютерах (HOST-компьютерах). Такой поход позволяет значительно уменьшить стоимость системы.

При классификации кластеров используется и ряд других классификационных признаков (рис. 7.11).

Рис. 7.11. Классификация вычислительных кластеров

Рассмотрим два из них:

- классификация по стандартности комплектующих;
- классификация по функциональной направленности.

Классификация вычислительных кластеров по стандартности комплектующих

С точки зрения стандартности комплектующих можно выделить два класса кластерных систем:

- вычислительный кластер строится целиком из стандартных комплектующих;
- при построении кластера используются эксклюзивные или нешироко распространенные комплектующие.

Вычислительные кластеры первого класса имеют низкие цены и простое обслуживание. Широкое распространение кластерные технологии получили как средство создания именно относительно дешевых систем суперкомпьютерного класса из составных частей массового производства.

Кластеры второго класса позволяют получить очень высокую производительность, но являются, естественно, более дорогими.

Классификация вычислительных кластеров по их функциональной направленности

С функциональной точки зрения кластерные системы можно разделить на:

- высокоскоростные кластерные системы (High Performance) — HP-кластеры;
- кластерные системы высокой готовности (High Availability) — HA-кластеры.

Высокоскоростные кластеры используются в областях, которые требуют значительной вычислительной мощности. Кластеры высокой готовности используются везде, где стоимость возможного простоя превышает стоимость затрат, необходимых для построения отказоустойчивой системы.

Высокоскоростные кластеры. Производительность вычислительного кластера, очевидно, зависти от производительности его узлов. С другой стороны, производительность кластера, как и всякой системы с распределенной памятью, сильно зависит от производительности коммуникационной среды. Обычно при построении вычислительных кластеров используют достаточно дешевые коммуникационные среды. Такие среды обеспечивают производительность на один-два порядка более низкую, чем производительность коммуни-

кационных сред суперкомпьютеров. Поэтому находится не так много задач, которые могут достаточно эффективно решаться на больших кластерных системах.

Влияние производительности коммуникационной среды на общую производительность кластерной системы зависит от характера выполняемой задачи. Если задача требует частого обмена данными между подзадачами, которые решаются на разных узлах вычислительного кластера, то быстродействию коммуникационной среды следует уделить максимум внимания. Соответственно, чем меньше взаимодействуют части задачи между собою, тем меньше внимания можно уделить быстродействию коммуникационной среды.

Разработано множество технологий соединения компьютеров в кластер.

Для того чтобы вычислительная система обладала высокими показателями готовности, необходимо, чтобы ее компоненты были максимально надежными, чтобы система была отказоустойчивой, а также, чтобы была возможной «горячая» замена компонентов (без останова системы). Благодаря кластеризации при отказе одного из компьютеров системы задачи могут быть автоматически (операционной системой) перераспределены между другими (исправными) узлами вычислительного кластера. Таким образом, отказоустойчивость кластера обеспечивается дублированием всех жизненно важных компонент вычислительной системы. Самыми популярными коммерческими отказоустойчивыми системами в настоящее время являются двухузловые кластеры.

Выделяется еще один класс вычислительных кластеров — *вычислительные сети* (GRID), объединяющие ресурсы множества кластеров, многопроцессорных и однопроцессорных ЭВМ, которые могут принадлежать разным организациям и быть расположенными в разных странах.

Разработка параллельных программ для вычислительных сетей усложняется из-за следующих проблем. Ресурсы (количество узлов, их архитектура, производительность), которые выделяются задаче, определяется только в момент обработки сетью заказа на выполнение это задачи. Поэтому программист не имеет возможности разработать программу для конкретной конфигурации вычислительной сети. Программу приходится разрабатывать так, чтобы она могла динамически (без перекомпиляции) самонастраиваться на выделенную конфигурацию сети. Кроме того, к неоднородности коммуникационной среды добавляется изменчивость ее характеристик, вызываемая изменениями загрузки сети. В лучшем случае программа

должна разрабатываться с учетом этой неоднородности коммуникационной среды, что представляет собой весьма непростую задачу. Как мы отмечали выше, подобная проблема имеет место и для вычислительных кластеров, построенных на основе персональных компьютеров или рабочих станций.

Эффективная производительность кластерных вычислительных систем (real applications performance — RAP) оценивается как 5–15% от их пиковой производительности (peak advertised performance, PAP). Для сравнения: у лучших малопроцессорных систем из векторных процессоров это соотношение оценивается как 30–50 %.

7.5. Тенденции развития средств вычислительной техники

Перспективы развития суперкомпьютерных технологий

В последнее время суперкомпьютерные технологии в России объявлены как один из приоритетов государственной политики.

Сегодня критические (прорывные) технологии в государствах, строящих экономику, основанную на знаниях, исследуются и разрабатываются на базе широкого использования суперкомпьютерных технологий. Без серьезной суперкомпьютерной инфраструктуры невозможно развивать перспективные технологии (биотехнологии, нанотехнологии, решения для энергетики будущего и т.п.), создать современные изделия высокой сложности (аэрокосмическая техника, суда, энергетические блоки электростанций различных типов).

Суперкомпьютерные технологии по праву считаются важнейшим фактором обеспечения конкурентоспособности экономики страны, а единственным способом победить конкурентов объявляют возможность обогнать их в расчетах.

В развитии суперкомпьютерной отрасли можно выделить два слоя:

- Технологии уровня «N» — это суперкомпьютерные технологии будущего, которые еще не вполне освоены, а только-только разрабатываются.
- Инновационные технологии — это совершенно новые технические решения, недоступные на рынке.

На базе инновационных технологий создают суперкомпьютеры, которые сильно вырываются вперед. Как правило, это машины, соответствующие первым 10 местам списка Top500. Эти вычислительные системы сверхвысокой производительности обладают мощностью, которая радикально отличает их от всех других машин. И на платформе таких систем можно выполнить расчеты, которые невоз-

можно повторить (ни за какое разумное время) на суперкомпьютерах более низкого класса. На базе таких расчетов можно создать в разных отраслях принципиально новые материалы, новые технологические решения, новые изделия, которые позволят обладающей ими стороне быть вне конкуренции и существенно оторваться от других игроков в соответствующей отрасли. Технологии уровня «N-1» — это технологии более низкого уровня, отработанные решения, широкодоступные на рынке. Суперкомпьютеры на их базе доступны (и даже могут быть воспроизведены) во многих странах. Соответственно, расчеты, выполняемые на таких машинах, могут быть воспроизведены многими. На базе таких расчетов можно создать в разных отраслях конкурентоспособные материалы, технологические решения, изделия — достичь нормального качества, заурядной конкурентоспособности. С такими изделиями можно выходить на мировой рынок, но на нем придется вести изнурительную конкурентную борьбу с десятком подобных товаров, созданных на базе подобных расчетов.

Таким образом:

1. Для того чтобы страна победила в конкуренции, ей необходимо победить в вычислениях.

2. Для того чтобы победить в вычислениях, необходимо создавать свои собственные вычислительные системы сверхвысокой производительности.

3. Необходимые для этого ключевые технические решения относятся к суперкомпьютерным технологиям уровня N. Они не доступны на рынке по двум причинам: многие из них еще не доведены до «продуктовой готовности» и никто не заинтересован отдавать в чужие руки «оружие победы» в конкуренции.

4. Таким образом, для того чтобы страна победила в конкуренции, ей предстоит создать свои собственные суперкомпьютерные технологии уровня N. По крайней мере в части ключевых технических решений, необходимых для разработки вычислительных систем сверхвысокой производительности.

На сегодняшний день и ближайшую перспективу в составе систем сверхвысокой производительности можно выделить:

1. **Аппаратные средства**:

- вычислительные узлы — процессоры, оперативная память, возможно ускорители вычислений и локальные диски;
- системная сеть — связывает вычислительные узлы и используется для организации вычислительного процесса (синхронизация и обмен данными между вычислительными фрагментами);
- системы хранения данных (СХД);
- подсистемы ввода-вывода данных;

- подсистемы визуализации;
- инфраструктура системы:
  — вспомогательная сеть (управление задачами, передача файлов);
  — подсистема мониторинга и управления аппаратурой;
  — подсистемы охлаждения и электропитания.

2. **Базовое и системное программное обеспечение**, в том числе:

- операционная система;
- системы низкоуровневой поддержки эффективного и отказоустойчивого использования аппаратных средств и реализации перспективных подходов к программированию;
- средства поддержки программирования — языки, инструменты и системы программирования.

Системы сверхвысокой производительности — это всегда системы на пределе существующих технических возможностей: процессоров и спецпроцессоров (ускорителей вычислений), системной сети, подсистем охлаждения и электропитания и т.д. В таких системах новые оригинальные разработки используются гораздо чаще, чем коммерчески доступные решения. И для этого есть несколько причин, в том числе:

- часто используются самые свежие разработки, которые только что созданы и еще не успели выйти на рынок как коммерчески доступные решения. Для вывода их на рынок потребуется несколько лет. Причем задержка может быть и умышленной — сохранение отрыва от конкурентов;
- системы строятся как уникальные объекты в единичных экземплярах — для многих коммерческих компаний единичные проекты не интересны.

Процессоры и сопроцессоры. Сегодня в системах сверхвысокой производительности используются как традиционные i86-совместимые процессоры Intel и AMD (пять систем из десяти), так и эксклюзивные решения — процессоры IBM Power BQC (четыре системы) и SPARC64 VIIIfx (одна система). Графические процессоры как ускорители вычислений используются в двух случаях из десяти.

Системная сеть. Технические характеристики системной сети — темп выдачи сообщений, задержка, пропускная способность — самым серьезным образом влияют на реальную производительность суперкомпьютера. Это влияние тем больше, чем больше размер системы. Для систем сверхвысокой производительности требуется системная сеть с рекордными техническими характеристиками. Вот почему в этих системах чаще всего используются для системной сети либо решения собственной разработки, либо дорогие и самые высококачественные коммерчески доступные решения.

Инфраструктура. Приходится разрабатывать оригинальные решения и для инфраструктуры: подсистем охлаждения, энергоснабжения, управления. Например, именно в настоящее время происходит перелом в используемых подходах в подсистеме охлаждения.

Инженерная логика здесь весьма простая:

- системы сверхвысокой производительности имеют высокие показатели электропотребления — как правило, от десятка мегаватт и выше. А значит, они имеют очень высокий уровень выделения тепла в вычислителе. И общая тенденция связана с ростом электропотребления (и соответственно — тепловыделения);
- стремление к максимальной производительности влечет стремление к увеличению плотности расположения электроники вычислителя: меньше длина проводников — значит меньше задержки на передачу сигналов, выше производительность;
- как результат мы имеем все большее и большее тепловыделение, приходящееся на единицу объема;
- значит, технологии охлаждения вычислителя должны непрерывно совершенствоваться.

В результате в последние годы мы наблюдаем смену технологий охлаждения электроники вычислителей.

Рассмотрим основные направления развития процессоров.

1. Повышение тактовой частоты.

Для повышения тактовой частоты при выбранных материалах используются: более совершенный технологический процесс с меньшими проектными нормами; увеличение числа слоев металлизации; более совершенная схемотехника меньшей каскадности и с более совершенными транзисторами, а также более плотная компоновка функциональных блоков кристалла.

Так, все производители микропроцессоров перешли на технологию КМОП, хотя Intel, например, использовала БиКМОП для первых представителей семейства Pentium. Известно, что биполярные схемы и КМОП на высоких частотах имеют примерно одинаковые показатели тепловыделения, но КМОП-схемы более технологичны, что и определило их преобладание в микропроцессорах.

Уменьшение размеров транзисторов, сопровождаемое снижением напряжения питания с 5 В до 2,5—3 В и ниже, увеличивает быстродействие и уменьшает выделяемую тепловую энергию. Все производители микропроцессоров перешли с проектных норм 0,35—0,25 мкм на 0,18 мкм и 0,12 мкм и стремятся использовать уникальную 0,07 мкм технологию (табл. 7.1).

Таблица 7.1

Основные параметры вычислительной техники с 2005 г.

Год производства	2005	2006	2007	2010	2013	2016
DRAM, нм	80	70	65	45	32	32
МП, нм	80	70	65	45	32	32
Uпит, В	0,9	0,9	0,7	0,6	0,5	0,4
P, Вт	170	180	190	218	251	288

При минимальном размере деталей внутренней структуры интегральных схем 0,1–0,2 мкм достигается оптимум, ниже которого все характеристики транзистора быстро ухудшаются. Практически все свойства твердого тела, включая его электропроводность, резко изменяются и «сопротивляются» дальнейшей миниатюризации, возрастание сопротивления связей происходит экспоненциально. Потери даже на кратчайших линиях внутренних соединений такого размера «съедают» до 90% сигнала по уровню и мощности.

При этом начинают проявляться эффекты квантовой связи, в результате чего твердотельное устройство становится системой, действие которой основано на коллективных электронных процессах. Проектная норма 0,05–0,1 мкм (50–100 нм) — это нижний предел твердотельной микроэлектроники, основанной на классических принципах синтеза схем.

Уменьшение длины межсоединений актуально для повышения тактовой частоты работы, так как существенную долю длительности такта занимает время прохождения сигналов по проводникам внутри кристалла. Например, в Alpha 21264 предприняты специальные меры по кластеризации обработки, призванные локализовать взаимодействующие элементы микропроцессора.

Проблема уменьшения длины межсоединений на кристалле при использовании традиционных технологий решается путем увеличения числа слоев металлизации. Так, Cyrix при сохранении 0,6 мкм КМОП технологии за счет увеличения с 3 до 5 слоев металлизации сократила размер кристалла на 40% и уменьшила выделяемую мощность, исключив существовавший ранее перегрев кристаллов.

Одним из шагов в направлении уменьшения числа слоев металлизации и уменьшения длины межсоединений стала технология, использующая медные проводники для межсоединений внутри кристалла, разработанная фирмой IBM и используемая в настоящее время и другими фирмами — изготовителями СБИС.

2. Увеличение объема и пропускной способности подсистемы памяти.

Возможные решения по увеличению пропускной способности подсистемы памяти включают создание кэш-памяти одного или нескольких уровней, а также увеличение пропускной способности интерфейсов между процессором и кэш-памятью и конфликтующей с этим увеличением пропускной способности между процессором и основной памятью. Совершенствование интерфейсов реализуется как увеличением пропускной способности шин (путем увеличения частоты работы шины и/или ее ширины), так и введением дополнительных шин, расшивающих конфликты между процессором, кэш-памятью и основной памятью. В последнем случае одна шина работает на частоте процессора с кэш-памятью, а вторая — на частоте работы основной памяти. При этом частоты работы второй шины, например, равны 66, 66, 166 МГц для микропроцессоров Pentium Pro-200, Power PC 604E-225, Alpha 21164-500, работающих на тактовых частотах 300, 225, 500 МГц соответственно. При ширине шин 64, 64, 128 разрядов это обеспечивает пропускную способность интерфейса с основной памятью 512, 512, 2560 Мбайт/с соответственно.

Общая тенденция увеличения размеров кэш-памяти реализуется по-разному:

- внешние кэш-памяти данных и команд с двухтактовым временем доступа объемом от 256 Кбайт до 2 Мбайт со временем доступа 2 такта в HP PA-8000;
- отдельный кристалл кэш-памяти второго уровня, размещенный в одном корпусе в PentiumPro;
- размещение отдельных кэш-памяти команд и кэш-памяти данных первого уровня объемом по 8 Кбайт и общей для команд и данных кэш-памяти второго уровня объемом 96 Кбайт в Alpha 21164.

Наиболее используемое решение состоит в размещении на кристалле отдельных кэш-памятей первого уровня для данных и команд с возможным созданием внекристальной кэш-памяти второго уровня. Например, в Pentium II использованы внутрикристальные кэш-памяти первого уровня для команд и данных по 16 Кбайт каждая, работающие на тактовой частоте процессора, и внекристальный кэш второго уровня, работающий на половинной тактовой частоте.

3. Увеличение количества параллельно работающих исполнительных устройств.

Каждое семейство микропроцессоров демонстрирует в следующем поколении увеличение числа функциональных исполнительных устройств и улучшение их характеристик, как временных (сокращение числа ступеней конвейера и уменьшение длительности

каждой ступени), так и функциональных (введение MMX-расширений системы команд и т.д.).

В настоящее время процессоры могут выполнять до 6 операций за такт. Однако число операций с плавающей точкой в такте ограничено двумя для R10000 и Alpha 21164, а 4 операции за такт делает HP PA-8500.

Для того чтобы загрузить функциональные исполнительные устройства, используются переименование регистров и предсказание переходов, устраняющие зависимости между командами по данным и управлению, буферы динамической переадресации.

Широко используются архитектуры с длинным командным словом — VLIW. Так, архитектура IA-64, развиваемая Intel и HP, использует объединение нескольких инструкций в одной команде (EPIC). Это позволяет упростить процессор и ускорить выполнение команд. Процессоры с архитектурой IA-64 могут адресоваться к 4 Гбайтам памяти и работать с 64-разрядными данными. Архитектура IA-64 используется в микропроцессоре Merced, обеспечивая производительность до 6 Гфлоп при операциях с одинарной точностью и до 3 Гфлоп — с повышенной точностью на частоте 1 ГГц.

4. Системы на одном кристалле и новые технологии.

В настоящее время получили широкое развитие системы, выполненные на одном кристалле, — SOC (System On Chip). Сфера применения SOC — от игровых приставок до телекоммуникаций. Такие кристаллы требуют применения новейших технологий.

Основной технологический прорыв в области SOC удалось сделать корпорации IBM, которая в 1999 г. смогла реализовать сравнительно недорогой процесс объединения на одном кристалле логической части микропроцессора и оперативной памяти. В новой технологии, в частности, используется так называемая конструкция памяти с врезанными ячейками (trench cell). В этом случае конденсатор, хранящий заряд, помещается в некое углубление в кремниевом кристалле. Это позволяет разместить на нем свыше 24 тыс. элементов, что почти в 8 раз больше, чем на обычном микропроцессоре, и в 2—4 раза больше, чем в микросхемах памяти для ПК. Следует отметить, что, хотя кристаллы, объединяющие логические схемы и память на одном кристалле, выпускались и ранее, например, такими фирмами, как «Toshiba», «Siemens AG» и «Mitsubishi», подход, предложенный IBM, выгодно отличается по стоимости. Причем ее снижение никоим образом не сказывается на производительности.

Использование новой технологии открывает широкую перспективу для создания более мощных и миниатюрных микропроцессоров и по-

могает создавать компактные, быстродействующие и недорогие электронные устройства: маршрутизаторы, компьютеры, контроллеры жестких дисков, сотовые телефоны, игровые и Интернет-приставки.

Нанотехнологии

Нанотехнологии — это технологии, оперирующие величинами порядка нанометра. Это технологии манипуляции отдельными атомами и молекулами, в результате которых создаются структуры сложных спецификаций. Слово «нано» (в древнегреческом языке «nano» — «карлик») означает миллиардную часть единицы измерения и является синонимом бесконечно малой величины, в сотни раз меньшей длины волны видимого света и сопоставимой с размерами атомов. Поэтому переход от «микро» к «нано» — это уже не количественный, а качественный переход: скачок от манипуляции веществом к манипуляции отдельными атомами. Мир таких бесконечно малых величин намного меньше, чем мир сегодняшних микрокристаллов и микротранзисторов.

Сейчас работы в области нанотехнологий ведутся в четырех основных направлениях:

- молекулярная электроника;
- биохимические и органические решения;
- квазимеханические решения на основе нанотрубок;
- квантовые компьютеры.

Наночастицы

Современная тенденция к миниатюризации показала, что вещество может иметь совершенно новые свойства, если взять очень маленькую частицу этого вещества. Частицы размерами от 1 до 100 нанометров обычно называют наночастицами. Так, например, оказалось, что наночастицы некоторых материалов имеют очень хорошие каталитические и адсорбционные свойства. Другие материалы показывают удивительные оптические свойства, например, сверхтонкие пленки органических материалов применяют для производства солнечных батарей. Такие батареи хоть и обладают сравнительно низкой квантовой эффективностью, зато более дешевы и могут быть механически гибкими. Удается добиться взаимодействия искусственных наночастиц с природными объектами наноразмеров — белками, нуклеиновыми кислотами и др. Тщательно очищенные наночастицы могут самовыстраиваться в определенные структуры. Такая структура содержит строго упорядоченные наночастицы и также зачастую проявляет необычные свойства.

Нанообъекты делятся на 3 основных класса: трехмерные частицы, получаемые взрывом проводников, плазменным синтезом, восстановлением тонких пленок и т.д.; двумерные объекты — пленки, получаемые методами молекулярного наслаивания, CVD, ALD, методом ионного наслаивания и т.д.; одномерные объекты — вискеры, эти объекты получаются методом молекулярного наслаивания, введением веществ в цилиндрические микропоры и т.д. Также существуют нанокомпозиты — материалы, полученные введением наночастиц в какие-либо матрицы. На данный момент обширное применение получил только метод микролитографии, позволяющий получать на поверхности матриц плоские островковые объекты размером от 50 нм, применяется он в электронике; метод CVD и ALD в основном применяется для создания микронных пленок. Прочие методы в основном используются в научных целях. В особенности следует отметить методы ионного и молекулярного наслаивания, поскольку с их помощью возможно создание реальных монослоев.

Особый класс составляют органические наночастицы как естественного, так и искусственного происхождения.

Поскольку многие физические и химические свойства наночастиц, в отличие от объемных материалов, сильно зависят от их размера, в последние годы проявляется значительный интерес к методам измерения размеров наночастиц в растворах: анализ траекторий наночастиц, динамическое светорассеяние, седиментационный анализ, ультразвуковые методы.

Самоорганизация наночастиц и самоорганизующиеся процессы

Один из важнейших вопросов, стоящих перед нанотехнологией, — как заставить молекулы группироваться определенным способом, самоорганизовываться, чтобы в итоге получить новые материалы или устройства. Этой проблемой занимается раздел химии — супрамолекулярная химия. Она изучает не отдельные молекулы, а взаимодействия между молекулами, которые способны упорядочить молекулы определенным способом, создавая новые вещества и материалы. Обнадеживает то, что в природе действительно существуют подобные системы и осуществляются подобные процессы. Так, известны биополимеры, способные организовываться в особые структуры. Один из примеров — белки, которые не только могут сворачиваться в глобулярную форму, но и образовывать комплексы — структуры, включающие несколько молекул белков. Уже сейчас существует метод синтеза, использующий специфические свойства молекулы

ДНК. Берется комплементарная ДНК (кДНК), к одному из концов подсоединяется молекула А или Б. Имеем 2 вещества: ----А и ----Б, где ---- — условное изображение одинарной молекулы ДНК. Теперь, если смешать эти 2 вещества, между двумя одинарными цепочками ДНК образуются водородные связи, которые притянут молекулы А и Б друг к другу. Условно изобразим полученное соединение: ====АБ. Молекула ДНК может быть легко удалена после окончания процесса.

Однако явления самоорганизации не замыкаются только на спонтанном упорядочении молекул и/или иных частиц в результате их взаимодействия. Существуют и другие процессы, которым присуща способность к самоорганизации, не являющиеся предметом супрамолекулярной химии. Одним из таких процессов является электрохимическое анодное оксидирование (анодирование) алюминия, а именно та его разновидность, что приводит к формированию пористых анодных оксидных пленок (ПАОП). ПАОП представляют собой квазиупорядоченные мезопористые структуры с порами, расположенными нормально к поверхности образца и имеющими диаметр от единиц до сотен нанометров и длину от долей до сотен микрометров. Существуют процессы, позволяющие в существенной степени увеличить степень упорядоченности расположения пор и создавать на основе ПАОА наноструктурированные одно-, двух- и трехмерные массивы.

В последнее время резко увеличилось количество публикаций о новых достижениях в области нанотехнологий. Самые свежие новости можно найти, например, на сайте http://www.nanonewsnet.com/.

Компьютеры и микроэлектроника

Центральные процессоры — 15 октября 2007 г. компания «Intel» заявила о разработке нового прототипа процессора, содержащего наименьший структурный элемент размерами примерно 45 нм. В дальнейшем компания намерена достичь размеров структурных элементов до 5 нм. Основной конкурент «Intel» — компания AMD также давно использует для производства своих процессоров нанотехнологические процессы, разработанные совместно с компанией IBM. Характерным отличием от разработок «Intel» является применение дополнительного изолирующего слоя SOI, препятствующего утечке тока за счет дополнительной изоляции структур, формирующих транзистор. Уже существуют рабочие образцы процессоров с транзисторами размером 14 нм и опытные образцы на 10 нм.

Жесткие диски — в 2007 г. Питер Грюнберг и Альберт Ферт получили Нобелевскую премию по физике за открытие GMR-эффекта, позволяющего производить запись данных на жестких дисках с атомарной плотностью информации.

Сканирующий зондовый микроскоп — микроскоп высокого разрешения, основанный на взаимодействии иглы кантилевера (зонда) с поверхностью исследуемого образца. Обычно под взаимодействием понимается притяжение или отталкивание кантилевера от поверхности из-за сил Ван-дер-Ваальса. Но при использовании специальных кантилеверов можно изучать электрические и магнитные свойства поверхности. СЗМ может исследовать как проводящие, так и непроводящие поверхности даже через слой жидкости, что позволяет работать с органическими молекулами (ДНК). Пространственное разрешение сканирующих зондовых микроскопов зависит от характеристик используемых зондов. Разрешение достигает атомарного по горизонтали и существенно превышает его по вертикали.

Антенна-осциллятор — 9 февраля 2005 г. в лаборатории Бостонского университета была получена антенна-осциллятор размерами порядка 1 мкм. Это устройство насчитывает 5000 млн атомов и способно осциллировать с частотой 1,49 Гц, что позволяет передавать с ее помощью огромные объемы информации.

Плазмоны — коллективные колебания свободных электронов в металле. Характерной особенностью возбуждения плазмонов можно считать так называемый плазмонный резонанс. Длина волны плазмонного резонанса, например, для сферической частицы серебра диаметром 50 нм составляет примерно 400 нм, что указывает на возможность регистрации наночастиц далеко за границами дифракционного предела (длина волны излучения много больше размеров частицы). В начале 2000 г., благодаря быстрому прогрессу в технологии изготовления частиц наноразмеров, был дан толчок к развитию новой области нанотехнологии — наноплазмонике. Стало возможным передавать электромагнитное излучение вдоль цепочки металлических наночастиц с помощью возбуждения плазмонных колебаний.

Фотоника

Фотоника — это технология излучения, передачи, регистрации света при помощи волоконной оптики и оптоэлектроники.

Довольно давно уже известна оптимальная среда для передачи огромных массивов данных — это свет, бегущий по волоконно-оп-

тическим кабелям. А все компьютерные транзисторы работают с электрическим током, текущим по медным проводам. Исследователям лабораторий Intel удалось органически совместить кремний со светом — так родилась кремниевая фотоника.

16 февраля 2004 г. впервые было продемонстрировано устройство, передающее информацию по волоконно-оптическому кабелю со скоростью 1 Гбит в секунду!

Луч света, идущий по оптическому волокну, расщепляется на два луча, затем один из лучей проходит через специальное устройство, в котором световые колебания могут сдвигаться по фазе. После сложения лучей наблюдается интерференция. Наличие света считают «1», а его отсутствие — «0».

До сих пор существовали быстрые модуляторы (устройства, преобразующие свет в последовательность битов информации), но они были очень дорогими, сложными в производстве и делались с использованием экзотических материалов (таких как арсенид галлия или фосфид индия). Самые быстрые кремниевые модуляторы работали на скоростях около 20 МГц. Кремниевый модулятор Intel работает со скоростью более 1 ГГц, исследователи надеются повысить эту скорость еще раз в 10!

У кремниевой фотоники есть масса преимуществ. Прежде всего это то, что по оптическому волокну можно передавать тысячи потоков сигналов на разных длинах световых волн, тогда как по медному проводу может идти лишь один ток. Теоретический предел для такой передачи близок к 100 трлн бит в секунду — этого достаточно, чтобы передать по одному волокну (в 30 раз тоньше человеческого волоса) телефонные переговоры всех жителей Земли одновременно.

Микропроцессорная технология потенциально имеет много назначений: создание персональных электронных партнеров, интеллектуализация (в известном смысле «оживление») всей техносферы, усиление и защита функций организма с помощью персональных медико-кибернетических устройств, в том числе вживляемых в организм.

В результате эволюции электронной технологии от «микро» к «нано» и ее слияния с «генной», вероятно, будет достигнуто состояние, при котором станет возможным синтез в массовых количествах любых технических устройств. Однако основная цель будущей нанотехнологии, по всей вероятности,— создание структур, способных к эволюции и саморазвитию.

Контрольные вопросы и задания

1. Примеры классификации вычислительных систем.
2. Приведите примеры векторно-конвейерной системы.
3. Приведите примеры симметричных мультипроцессорных систем (SMP).
4. Какие уровни параллелизма реализуют симметричные мультипроцессорные системы?
5. Что такое кластерные системы?
6. Есть ли разница между производительностью (performance) вычислительной системы и ее быстродействием?
7. Какие две проблемы призвана решить кластерная организация вычислительной системы?
8. Существуют ли ограничения на число узлов в кластерной ВС? И если существуют, то чем они обусловлены?
9. Какие задачи в кластерной вычислительной системе возлагаются на специализированное (кластерное) программное обеспечение?
10. Каким образом может быть организовано взаимодействие между узлами кластерной ВС?
11. Сформулируйте основные тенденции развития микропроцессоров.
12. За счет каких факторов достигают повышения тактовой частоты МП?
13. Какие архитектурные особенности приводят к улучшению характеристик МП?
14. В чем заключается сущность гипотезы Минского в развитии MIMD-вычислительных систем?
15. Каким образом решается проблема уменьшения длины межсоединений на кристалле при использовании традиционных технологий?
16. Что такое нанотехнологии? В каких направлениях они развиваются?
17. Приведите примеры использования нанотехнологий.
18. Что такое фотоника? Расскажите о ее достижениях.

СПИСОК ЛИТЕРАТУРЫ

1. *Таненбаум Э.*, *Остин Т.* Архитектура компьютера. 6-е изд. СПб.: Питер, 2014.
2. *Гибсон Г.*, *Лю Ю-Чжен.* Микропроцессоры семейства 8086/8088. М.: Радио и связь, 1987.
3. *Микушин А.В.* Занимательно о микроконтроллерах. СПб.: БХВ-Петербург, 2006.
4. *Микушин А.В., Сажнев А.М., Сединин В.И.* Цифровые устройства и микропроцессоры. СПб.: БХВ-Петербург, 2010.
5. *Олифер В.Г.*, *Олифер Н.А.* Сетевые операционные системы. СПб.: Питер, 2009.
6. *Майоров С.А.*, *Кириллов В.В.*, *Приблуда А.А.* Введение в микроЭВМ. Л.: Машиностроение, 1988.
7. *Гук М.* Аппаратные средства IBM PC. СПб.: Питер, 2006.
8. *Угрюмов Е.П.* Цифровая схемотехника. СПб.: БХВ-Петербург, 2004.
9. *Аванесян Г.Р.*, *Левшин В.П.* Интегральные микросхемы ТТЛ, ТТЛШ: Справочник. М.: Машиностроение, 1993.
10. *Атовмян И.О.* Архитектура вычислительных систем. М.: МИФИ, 2002.
11. *Бродин В.Б.*, *Шагурин И.И.* Микропроцессор i486. Архитектура, программирование, интерфейс. М.: ДИАЛОГ-МИФИ, 1993.
12. *Гордеев А.В.* Операционные системы: Учебник для вузов. 2-е изд. СПб.: Питер, 2007.
13. *Олифер В.Г.*, *Олифер Н.А.* Сетевые операционные системы. СПб.: Питер, 2002.
14. *Гуров В.В.* Синтез комбинационных схем в примерах. М.: МИФИ, 2001.
15. *Гуров В.В.*, *Ленский О.Д.*, *Соловьев Г.Н.*, *Чуканов В.О.* Архитектура, структура и организация вычислительного процесса в ЭВМ типа IBM PC / Под ред. Г.Н. Соловьева. М.: МИФИ, 2002.
16. *Каган Б.М.* Электронные вычислительные машины и системы. М.: Энергоатомиздат, 1991.
17. *Казаринов Ю.М.*, *Номоконов В.Н.*, *Подклетнов Г.С.* и др. Микропроцессорный комплект К1810: Структура, программирование, применение / Под ред. Ю.М. Казаринова. М.: Высшая школа, 1990.
18. *Киселев А.В.*, *Корнеев В.В.* Современные микропроцессоры. М.: Нолидж, 1998

Электронные ресурсы

1. *Барский А.Б.* Электронная книга. Архитектура параллельных вычислительных систем Интернет-университет информационных технологий. ИНТУИТ. ISBN: 978-5-9556-0071-0

2. *Гуров В.В.*, *Чуканов В.О.* Электронная книга. Архитектура и организация ЭВМ. ИНТУИТ Национальный открытый университет, 2005.

Интернет-ресурсы

1. http://all-ht.ru Информационный сайт о высоких технологиях
2. http://perscom.ru ПерсКом
3. http://www.inf1.info Планета информатики

ОГЛАВЛЕНИЕ

Учебное издание

Вера Владимировна Степина

ОСНОВЫ АРХИТЕКТУРЫ, УСТРОЙСТВО И ФУНКЦИОНИРОВАНИЕ ВЫЧИСЛИТЕЛЬНЫХ СИСТЕМ

Учебник

Оригинал-макет подготовлен в Издательстве «КУРС»

Подписано в печать 05.12.2017.
Формат 60×90/16. Бумага офсетная. Гарнитура Newton.
Печать цифровая. Усл. печ. л. 18.0.
Доп. тираж 100 экз. Заказ № 00000

ТК 653023-809914-051217

ООО Издательство «КУРС»
127273, Москва, ул. Олонецкая, д. 17А, офис 104.
Тел.: (495) 203-57-83.
E-mail: kursizdat@gmail.com http://www.kursizdat.ru

ООО «Научно-издательский центр ИНФРА-М»
127282, Москва, ул. Полярная, д. 31В, стр. 1
Тел.: (495) 280-15-96, 280-33-86. Факс: (495) 280-36-29
E-mail: books@infra-m.ru http://www.infra-m.ru