# Основные понятия:

*Вычислительная машина* (ВМ) — физическая система (устройство или комплекс устройств), предназначенная для механизации или автоматизации процесса алгоритмической обработки информации или вычислений.

# Типизация по назначению:

Калькулятор, Консольный компьютер ,Миникомпьютер, Мейнфрейм, Рабочая станция, Сервер, Суперкомпьютер.

Персональный компьютер: Настольный компьютер, Ноутбук (Лэптоп), Нетбук , Игровая приставка (Игровая консоль), Карманный компьютер (КПК), Смартфон, Надеваемый компьютер (носимый компьютер), Планшетный персональный компьютер («Чистый планшетник», планшетный ноутбук, UMPC)

# Интегральные схемы:

Кремниевая пластина: тонкая круглая пластина особо чистого монокристалла кремния, обычно диаметром **200 или 300 мм**, из которой изготавливаются микросхемы.На пластине в ходе технологического процесса обработки формируется **матрица ячеек**, содержащих одинаковые электронные схемы.Пластина разрезается по границам ячеек на **кристаллы** (die), или **чипы** (chip).Под микроскопом к ним припаиваются **выводы** и каждый кристалл помещается в **защитный корпус.**

# Классификация:

Данные, с которыми оперируют вычислительные машины, могут быть представлены в непрерывном, дискретном или комбинированном виде.

Аналоговая вычислительная машина (АВМ): представление информации реализуется путём замены математических величин некоторыми физическими величинами (угол поворота вала, сила тока и тому подобное)

Цифровая вычислительная машина (ЦВМ):устройство переработки информации, представленной в цифровых кодах.

Гибридная вычислительная машина (ГВМ):сочетается ряд особенностей цифровых и аналоговых вычислительных машин.

# Цифровое представление:

Дискретный сигнал:информационный сигнал, который представляется в виде отдельных отсчетов взятых по времени.

Цифровой сигнал:*дискретный сигнал*, *квантованный по амплитуде* .

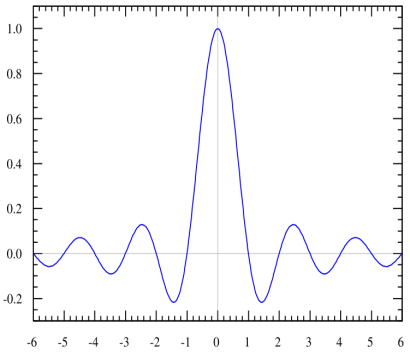
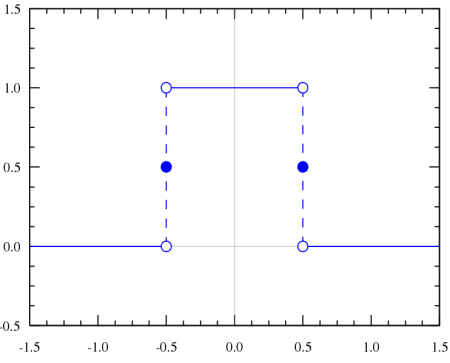
Теорема Котельникова (теорема Найквиста — Шеннона):если аналоговый сигнал *x*(*t*) имеет ограниченный спектр, то он может быть восстановлен однозначно и без потерь по своим дискретным отсчётам, взятым с частотой более удвоенной максимальной частоты спектра *f*max .

*f дискр> 2f*max

# Преобразование Фурье:

Описывает коэффициенты («амплитуды») при разложении исходной функции на элементарные составляющие — **гармонические колебания с разными частотами**.



# Многоуровневая компьютерная организация:

Существует принципиальная разница между тем, что удобно для людей, и тем, что «удобно» для машин. Человек хочет сделать одно, но машина может сделать только другое.

Машинные команды: обычно их стараются сделать как можно **проще**, чтобы избежать сложностей при конструировании компьютера и снизить затраты на необходимую электронику . так как большинство машинных языков очень **примитивны**, использовать их **трудно** и **утомительно .**

Проблему можно решить двумя способами: разработка новых команд, которые более удобны для человека, чем встроенные машинные команды.

# Языки, уровни и виртуальные машины:

Встроенные машинные команды формируют язык Я0.

Новые команды в совокупности формируют язык Я1.

Первый способ – **трансляция:** замена каждой команды Я1 на эквивалентный набор команд в языке Я0. компьютер выполняет новую программу, написанную на языке Я0, вместо старой программы, написанной на Я1.

# Виды трансляторов:

**Адресный**: Функциональное устройство, преобразующее виртуальный адрес в реальный (физический) адрес.

**Диалоговый**: Обеспечивает использование языка программирования в режиме разделения времени.

**Многопроходной**: Формирует объектный модуль за несколько просмотров исходной программы.

**Обратный**: Детранслятор (декомпилятор, дизассемблер).

**Оптимизирующий**: Выполняет оптимизацию кода в создаваемом объектном модуле.

**Синтаксически-ориентированный** (синтаксически-управляемый): Получает на вход описание синтаксиса и семантики языка и текст на описанном языке, который и транслируется в соответствии с заданным описанием.

**Тестовый**: Набор макрокоманд языка ассемблера, позволяющих задавать различные отладочные процедуры в программах, составленных на языке ассемблера.

# Трансляция:

**Компилятор**: транслятор, который преобразует программы в машинный язык, принимаемый и исполняемый непосредственно процессором

Примеры: С, С++

# Языки, уровни и виртуальные машины:

Второй способ – **интерпретация**: написание программы на языке Я0, которая берет программы, написанные на языке Я 1, в качестве входных данных, рассматривает каждую команду по очереди и сразу выполняет эквивалентный набор команд языка Я0. не требует составления новой программы на Я0.

Примеры : APL, LISP (чистая интерпретация), Python, Java, Perl

# Трансляция vs интерпретация:

В обоих методах компьютер в конечном итоге выполняет набор команд на языке Я0, эквивалентных командам Я1.

Различия:

Трансляция: вся программа Я1 переделывается в программу Я0.

Интерпретация**:** происходит последовательная перекодировка и выполнение команд.

Взаимопроникновение: интерпретаторы могут быть компилирующими. в трансляторах может требоваться интерпретация для конструкций метапрограммирования .

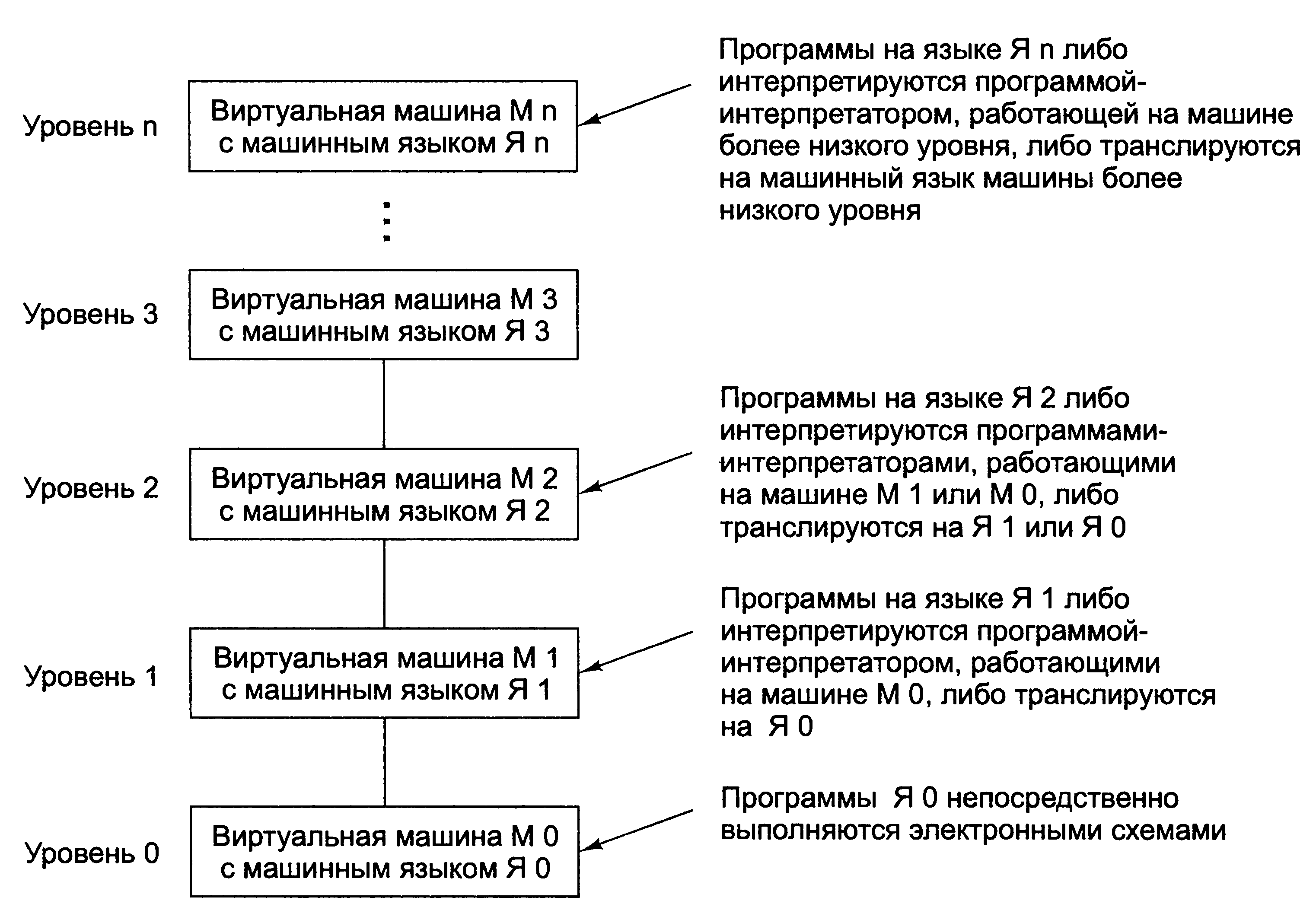
**смешанная реализация**: интерпретатор перед исполнением программы транслирует её на промежуточный язык (например, в байт-код), более удобный для интерпретации

Примеры: Java (перед интерпретацией, компиляция в байт-код). в процессорах Pentium с архитектурой NetBurst машинный код перед транслируется в последовательность **микроопераций** (кэш-трасс).

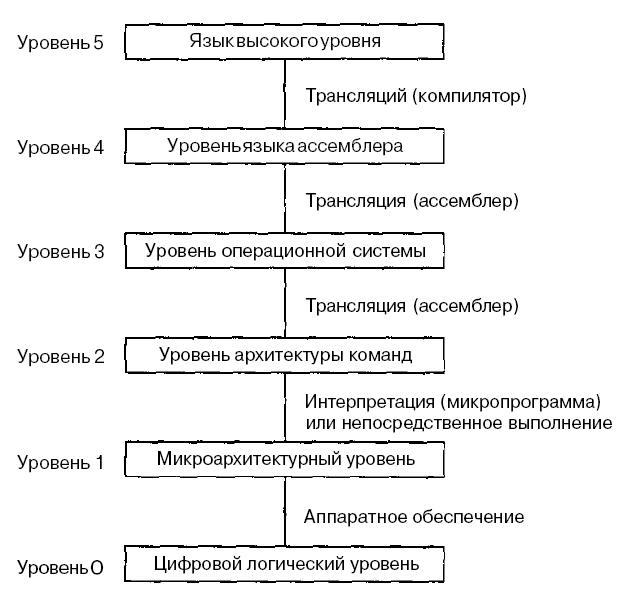
# Виртуальные машины:

Обычно проще представить себе существование **виртуальной машины,** для которой машинным языком является язык Я1 (назовем М1). Чтобы трансляцию или интерпретацию можно было осуществить **без больших трудностей**, языки **Я0 и Я1 не должны сильно различаться.** Язык Я1 может быть все еще неудобным для человека – решение создавать **языки более высокого уровня.**

# Многоуровневая система:



# Современные многоуровневые машины:



# Уровень 0:

Аппаратное обеспечение машины.

Объекты наз. **вентилями**, которые состоят из транзисторов.

Несколько вентилей формируют **1 бит 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:

**Микроархитектурный уровень** : Регистры, которые формируют локальную память. АЛУ (арифметико-логическое устройство): выполняет простые арифметические операции. Регистры вместе с АЛУ формируют **тракт данных,** по которому поступают данные.

Основная операция тракта данных состоит в следующем :

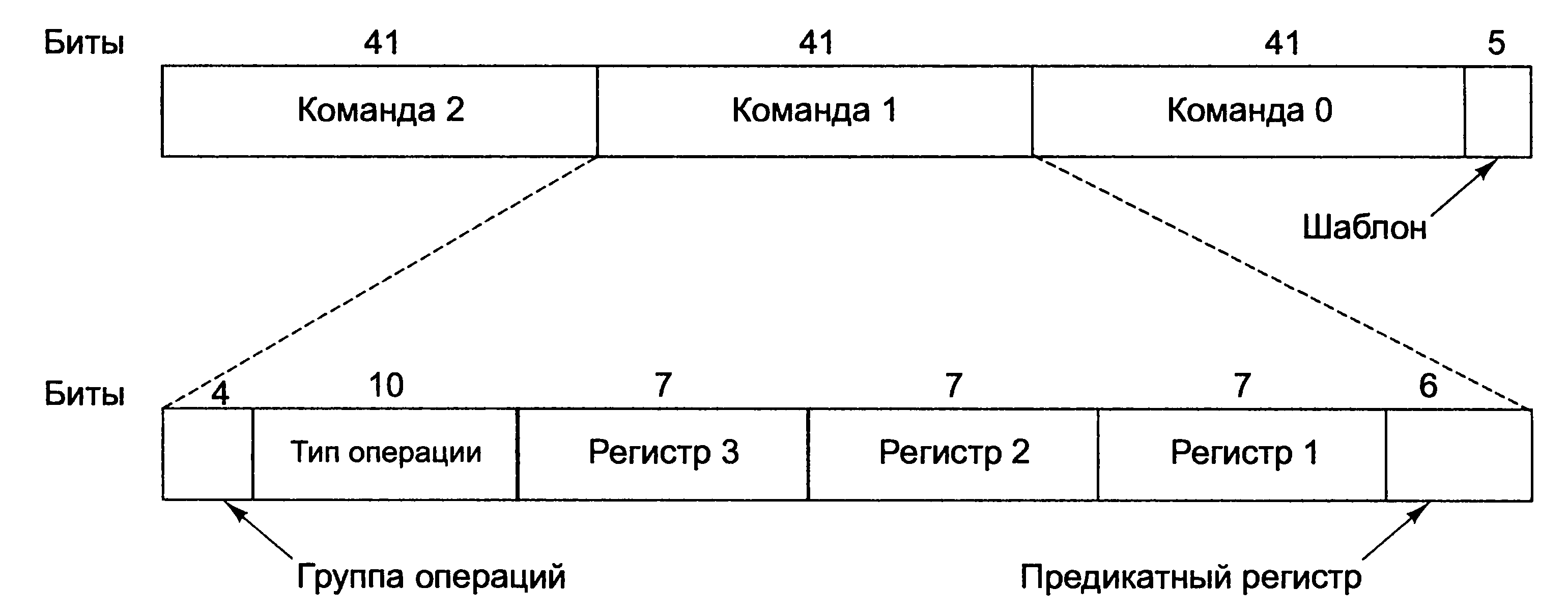
* + - Выбирается один или два регистра
    - АЛУ производит над ними какую-либо операцию
    - результат помещается в один из этих регистров

На некоторых машинах работа тракта данных контролируется **микропрограммой.** На других машинах тракт данных контролируется аппаратными средствами.

# Уровень 2:

**Уровнь архитектуры системы команд**: Включает набор машинных команд, т.е. команды, которые выполняются микропрограммой-интерпретатором или аппаратным обеспечением.





# Уровень 3:

**Уровень операционных систем:** ОС выполняет две мало связанные функции: 1)обеспечение пользователю-программисту удобств посредством предоставления для него **расширенной машины**. 2)повышение **эффективности** использования компьютера путем **рационального** **управления** его **ресурсами**.

Особенности: набор **новых** **команд**, другая **организация** **памяти**, способность выполнять **две и более программ одновременно** и некоторые другие .

# Уровень 4 и 5:

Уровни 1-3: обычно **интерпретируются ,** цифровые языки: программы состоят из длинных рядов цифр, которые удобны для компьютеров, но совершенно неудобны для людей.

Уровни 4 и 5: обычно, хотя и не всегда, поддерживаются транслятором. Языки содержат слова и сокращения, понятные человеку.

Уровень 4: представляет собой символическую форму одного из языков более низкого уровня.

Программы сначала **транслируются** на язык уровня 1, 2 или 3, а затем **интерпретируются** соответствующей виртуальной или фактически существующей машиной.

Программа, которая выполняет трансляцию, называется **асемблером**.

Уровень 5: обычно состоит из языков, разработанных для прикладных программистов.

Такие языки называются **языками высокого уровня.** Наиболее известные среди них: BASIC, С, C++, Java, LISP, Prolog, ...

Программы, написанные на этих языках, обычно транслируются на уровень 3 или 4.

Трансляторы, которые обрабатывают эти программы, называются **компиляторами.**

Иногда также используется метод интерпретации: Например, программы на языке Java обычно интерпретируются.

**Лекция 1 - Методы и средства построения распределенных программных систем**

«Распределенная система это набор независимых узлов (компьютеров), которые представляются пользователю как единая система.»

«Распределенная система это множество независимых компьютеров соединенных сетью с программным обеспечением, обеспечивающим их совместное функционирование.»

«Система, состоящая из набора двух или более независимых узлов, которые координируют свою работу посредством синхронного или асинхронного обмена сообщениями.»

«Система, чьи компоненты размещены на различных узлах, взаимодействующие и управляемые только посредством передачи сообщений.»

«...система нескольких автономных вычислительных узлов, взаимодействующих для выполнения общей цели.»

«…я не могу объяснить, что такое распределенная система, но узнаю ее как только мне ее покажут»

**Последствия «распределенности»**

Параллельность

--Необходимость в разделяемых ресурсах

коллизии при доступе

--Параллельное выполнение

«гонки потоков»

--Необходимость в синхронизации

**Нет «глобального» времени**

--Асинхронная передача сообщений

--Ограниченная точность синхронизации часов

**Нет «глобального» состояния системы**

--В распределенной системе нет ни одного процесса, которому было бы известно текущее глобальное состояние системы

* Процессы выполняются автономно, изолированно
* Сбои отдельных процессов могут остаться необнаруженными
* Отдельные процессы могут не подозревать об общесистемном сбое
* Сбои происходят чаще, чем в централизованной системе
* Новые причины сбоев (которых не было в монолитных системах)
* Сетевые сбои изолируют процессы и фрагментируют систему на изолированные части

**Принципы разделения**

* **Функциональное разделение**: узлы выполняют различные задачи

Клиент / сервер

Хост / Терминал

Сборка данных/ обработка данных

* **Естественное разделение** (определяемое задачей)

Система обслуживания сети супермаркетов

Сеть для поддержки коллективной работы

* Мотивы разделения
* **Необходимость обмена данными между системами**
* **Распределение нагрузки/балансировка**

назначение задач на процессоры так, чтобы оптимизировать общую загрузку системы

* **Усиление мощности**

различные узлы работают над одной задачей

распределенные системы, содержащие набор микропроцессоров, по мощности могут приближаться к суперкомпьютеру

* **Физическое разделение**

система строится в предположении, что узлы физически разделены (требования к надежности, устойчивости к сбоям)

* **Экономические**

набор дешевых чипов может обеспечить лучшие показатели отношения цена/производительность, чем мэйнфрэйм

* **Специализация компонентов**

упрощение и удешевление

**Service Oriented Architecture**

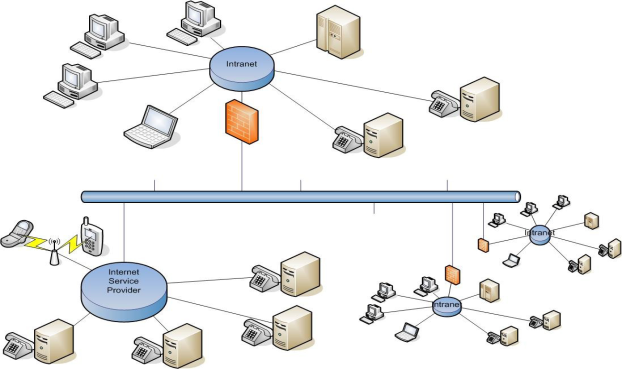
* SOA: SW architecture where all **components** are designed to be **services**
* Apps composed of interoperable services
  + Easy to tailor new version for subset of users
  + Also easier to recover from mistake in design
* Contrast to “SW silo” without internal APIs

CEO: Amazon shall use SOA!

1. “All teams will henceforth expose their data and functionality through   
   service interfaces
2. Teams must communicate with each other through these interfaces
3. There will be no other form of interprocess communication allowed: no direct linking, no direct reads of another team's data store, no shared-memory model, no back-doors whatsoever. The only communication allowed is via service interface calls over the network.
4. It doesn't matter what [API protocol] technology you use.
5. Service interfaces, without exception, must be designed from the ground up to be externalizable. That is to say, the team must plan and design to be able to expose the interface to developers in the outside world. No exceptions.
6. Anyone who doesn't do this will be fired.
7. Thank you; have a nice day!”

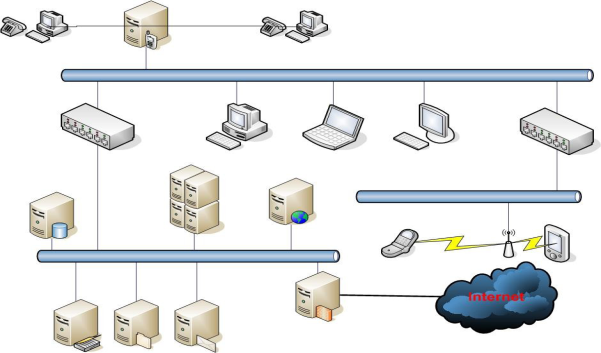
**Пример: Intranet**

* Администрируется локально
* Взаимодействие с Internet
* Предоставление сервисов (внутренним и внешним пользователям)



**Пример: Internet**

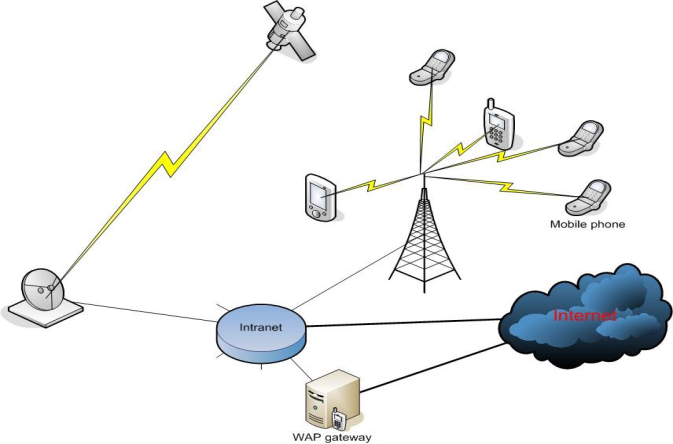
* Гетерогенная сеть компьютеров и приложений
* Реализация взаимодействия – стек протоколов Internet



**Пример: Wireless Information Devices**

* Система сотовой связи (GSM)

--Ресурсы разделяемы (радио частота, время передачи на частоте,...)



**Требования к распределенным системам**

* Открытость
* Безопасность
* Масштабируемость
* Механизмы обработки ошибок и восстановления после сбоев
* Методы решения проблем параллелизма
* Прозрачность
* Управляемость
* **Открытость**
* Гарантирует расширяемость
* Возможность повторного использования
* Важные факторы:
  + --Наличие четких спецификаций
  + --Наличие полной документации (!)
  + --Опубликованные интерфейсы
  + --Тестирование и проверка на многих платформах
* Использование открытых протоколов и стандартов (пример – стандарты OGC)

**Безопасность. Актуальные проблемы**

* Атаки типа DoS/DDoS (отказы в обслуживании)
* Безопасность мобильного кода

--Непредсказуемые эффекты

--Может вести себя подобно троянскому коню

**Масштабируемость**

* *Распределенная система масштабируема, если она остается эффективной при увеличении числа обслуживаемых пользователей или ресурсов*
* Проблемы:

--Контроль стоимости ресурсов

--Контроль потерь производительности

* Некоторые рекомендации

--Децентрализация алгоритмов

--Тиражирование (репликация) и кэширование данных

**Обработка ошибок**

Сбои более частые, чем в централизованных системах

Обработка сбоев включает:

--Определение факта сбоя (может оказаться невозможным)

--Маскирование

--Восстановление

Диагностика

--Может быть возможна (ошибки передачи – контрольная сумма)

--Может быть невозможна (удаленный сервер не работает или просто очень загружен)

**Параллелизм**

Контроль и планирование параллелизма

Правильное планирование доступа в параллельных потоках (устранение взаимоисключений, транзакции)

Синхронизация (семафоры, критические секции)

--Безопасно, но уменьшают производительность

Разделяемые объекты (ресурсы) должны работать корректно в многопоточной среде

**Прозрачность**

*Сокрытие гетерогенной и распределенной структуры системы так, чтобы пользователю система представлялась монолитной*

* **Прозрачность доступа:**

--доступ к локальным и удаленным ресурсам посредством одинаковых вызовов

* **Прозрачность расположения:**

--доступ к ресурсам вне зависимости от их физического расположения

* **Прозрачность параллелизма:**
  + возможность нескольких процессов параллельно работать с ресурсами, не оказывая влияния друг на друга
* **Прозрачность репликации:**
  + возможность нескольким экземплярам одного ресурса использоваться без знания физических особенностей репликации
* **Прозрачность обработки ошибок:**
  + защита программных компонентов от сбоев, произошедших в других программных компонентах. Восстановление после сбоев
* **Прозрачность мобильности:**
  + Возможность переноса приложения между платформами, без его переделки
* **Прозрачность производительности:** 
  + Возможность конфигурации системы с целью увеличения производительности при изменении состава платформы выполнения
* **Прозрачность масштабируемости:**
  + Возможность увеличения производительности без изменения структуры программной системы и используемых алгоритмов

**Управляемость**

Распределенные ресурсы не имеют центральной точки управления

Локальная оптимизация системы не всегда означает глобальную оптимизацию

--Нужен глобальный взгляд на проблему

--Не всегда возможен (есть системы, не принадлежащие одному владельцу)

**Сложности при реализации**

* Выбор архитектуры
* Гетерогенная среда
* Сложность развертывания
* Сложность отладки

**Архитектура**

Сильнейшая зависимость важнейших характеристик от выбранной архитектуры

* + Интерфейс модулей системы определяет количество «нелокальных» вызовов
  + Практическая невозможность отказа от принятых в начале проектирования неверных решений

Решение: использование широко известных шаблонов архитектур

* + Например, **шаблоны проектирования** (GoF) – «*Приёмы объектно-ориентированного проектирования. Паттерны проектирования*»

**Гетерогенная среда**

Различные компоненты системы выполняются на различных платформах

Решения:

* + Использование распространенных открытых стандартов и протоколов
  + Использование промежуточного программного обеспечения (middleware)

**Сложность развертывания**

Различные части приложения должны быть развернуты на различных узлах (с различными платформами)

Решения:

* + Разделение на модули развертывания
  + Использование специализированных инструментальных средств (deploy tools)

**Сложность отладки**

--Нет глобального состояния

--Параллельность → неповторяемость (невоспроизводимость) результатов выполнения

--Компоненты распределены по разным узлам

**Некоторые итоги**

*Распределенная система:*

--Автономные (но соединенные средой передачи данных) узлы

--Взаимодействие посредством передачи сообщений

* Много доводов в пользу того, что **распределенные системы** **нужны** и их нужно **уметь проектировать и реализовывать**
* Распределенные системы **существуют** и их нужно уметь **развивать** и **поддерживать**
* При разработке распределенных систем возникает много **специфических** **проблем**

**Лекция 2 - Модели OSI, RMI, … Middleware**

**Middleware. Получаемые преимущества**

\*\* Независимость от размещения:

Для вызовов с использованием RMI и RPC не нужно знание физического размещения вызываемых объектов (но нужно знание размещения сервисов именования)

\*\* Независимость от транспортного протокола:

Использование того или иного транспортного протокола прозрачно для разработчика

\*\* Независимость от платформы

\*\* Используются независимые способы представления данных (external data representation, XDR)

\*\* Независимость от используемых языков/систем программирования

\*\* Использование независящих от языка средств определения структур данных, модулей и интерфейсов (например, CORBA IDL ― Interface Definition Language)

**Объектная модель**

\*\*Компоненты объектной модели - объектные ссылки

Могут быть присвоены переменным соответствующих типов, использоваться в качестве параметров методов как передаваемые или возвращаемые аргументы

* + Интерфейсы
    - Определяют наборы методов и их сигнатуры
  + Методы
    - Могут изменять состояние объекта
    - Могут привести к вызовам других методов
  + Исключения
  + Сборка мусора
    - Освобождение памяти, более не используемой приложением

**Модель распределенных объектов**

Представляет собой вариацию модели клиент-сервер

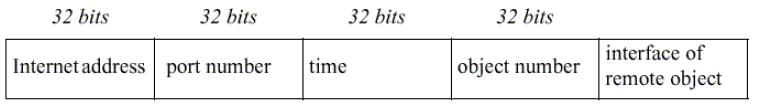
* «Серверы» содержат объекты, методы которых могут вызываться удаленно объектами – «клиентами». Интерфейсы удаленных вызовов определены с помощью специальных средств, таких как, например, CORBA IDL
* Клиенты вызывают методы удаленных объектов, используя какую-либо из реализаций RMI
* Вызов удаленных методов реализуется с использованием протокола типа «запрос-ответ»
  + Запрос содержит ссылку на вызываемый метод и параметры вызова; осуществляется посредством объектной ссылки на серверный объект
  + Ответ содержит параметры, возвращаемые клиенту

**Термины**

**Удаленный объект** - объект, могущий принимать удаленные вызовы (B на предыдущем слайде)

**Удаленная объектная ссылка** необходима вызывающей стороне для обращения к удаленному объекту

Может иметь следующий вид:



Может быть использована как аргумент вызываемого удаленно метода – как в качестве принимаемого, так и в качестве возвращаемого параметра

**Распределенные объекты vs. локальные объекты**

Жизненный цикл

* + Создание, использование и удаление

Обращение

* + Более сложное

Латентность (задержки при обращении)

Доступность (не всегда доступный!)

Ошибки

* + Больше типов ошибок

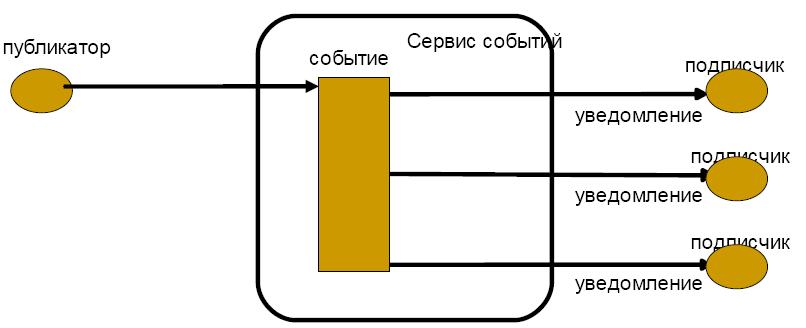
Безопасность

* + Больше угроз

**Модель, основанная на событиях**

* Объекты реагируют на события
  + например, вызов методов, щелчок мыши, …
* События могут изменять состояние объектов
* Множество объектов, расположенных на различных узлах могут извещаться о наступлении **событий определенного типа**
* Используется модель «публикатор - подписчик»
  + Объект, генерирующий события, называется **публикатором** - он публикует список типов событий, на которые остальные объекты могут подписаться
  + Объекты, подписывающиеся на события определенного типа, называются **подписчиками**. Они получают уведомление о наступлении события

**Система, основанная на событиях**



Сервис событий обеспечивает механизмы:

* + Публикации событий
  + Подписки на события
  + Уведомления о наступлении события
  + Получения событий

**Реализации Middleware**

Различные реализации *существенно* отличаются по архитектуре, способам использования, области применимости, предоставляемым сервисам и т.д.

* + Sun RPC
  + Java RMI (Sun)
  + DCOM (Microsoft)
  + CORBA (OMG)
  + Web – сервисы (различные реализации)
  + …

# Поколения:

Нулевое поколение (1642-1945): механические компьютеры .

Первое поколение (1945-1955): электронные лампы.

Второе поколение (1955-1965): транзисторы.

Третье поколение (1965-1980): интегральные схемы.

Четвертое поколение (1980-?): сверхбольшие интегральные схемы.

# Нулевое поколение — механические компьютеры (1642-1945):

# Блез Паскаль (1623-1662):

Первым человеком, создавшим счетную машину, был французский ученый **Блез Паскаль**, в честь которого назван один из языков программирования . Паскаль сконструировал эту машину в 1642 году - она была **механическая**: с шестеренками и ручным приводом. Счетная машина Паскаля могла выполнять только операции **сложения** и **вычитания.**

# Чарльз Бэббидж (1792-1871):

Профессор математики Кембриджского университета, изобретатель спидометра.

Разработал и сконструировал **разностную машину.** Эта механическая машина могла только **складывать** и **вычитать**, подсчитывала таблицы чисел для морской навигации. В машину был заложен только один алгоритм — **метод конечных разностей с использованием полиномов**. Любую функцию можно приблизить полиномами! (т. Вейерштрасса). Приступил к созданию **аналитической машины.**

1. компонента: 1)запоминающее устройство (**память**), 2)**вычислительное устройство**, 3)устройство **ввода** (для считывания перфокарт), 4)устройство **вывода** (перфоратор и печатающее устройство).

**Память** состояла из **1000 слов по 50 десятичных разрядов**, каждое из которых содержало переменные и результаты. Вычислительное устройство принимало операнды из памяти, затем выполняло операции сложения, вычитания, умножения или деления и возвращало полученный результат обратно в память. Аналитическая машина программировалась на ассемблере – ей было необходимо программное обеспечение. Чтобы создать это программное обеспечение, Бэббидж нанял молодую женщину — **Аду Августу Ловлейс**, дочь знаменитого британского поэта Байрона.

# Ада Ловлейс:

Была первым в мире программистом. В ее честь назван современный язык программирования Ada.

# Влияние Бэббиджа:

Для аналитической машины нужны были **тысячи и тысячи шестеренок**, сделанных с такой точностью, которая была невозможна в XIX веке. Но идеи Бэббиджа опередили его эпоху, и даже сегодня большинство современных компьютеров по строению сходны с аналитической машиной.

# Джон Атанасофф:

1930ые: Построил машину для решения **линейных уравнений**. В ней использовалась **бинарная арифметика** и **информационные емкости** (прототип современной динамической памяти). Состояла из электромагнитных реле и ламп.

# Harvard Mark I:

Работа над первым компьютером Айкена «Mark I» была закончена в 1944 году. Компьютер содержал **72 слова по 23 десятичных разряда** каждое. На устройствах ввода-вывода использовалась перфолента. Вес - *5 тонн,* всего *750 000* компонентов.

Характеристики: 0.3 сек. – сложение. 6 сек. – умножение. 1 мин. – вычисление синуса .

# Первое поколение — электронные лампы (1945-1955):

# COLOSSUS:

Первый в мире **электронный цифровой** компьютер! Создавался британским правительством. Был **засекречен** 30 лет и не мог служить основой для дальнейшего развития

Назначение: **Расшифровка закодированных немецких сообщений.**

# Electronic Numerical Integrator and Computer (ENIAC):

**Моушли** со своим студентом, **Дж. Преспером Экертом**, начали конструировать электронный компьютер, который они назвали **ENIAC.** (**Electronic Numerical Integrator and Computer — электронный цифровой интегратор и калькулятор)**

Состав: **18 000 электровакуумных ламп** и **1500 реле**. ENIAC весил **30 тонн** и потреблял **140 киловатт** электроэнергии. У машины было **20 регистров**, каждый из которых мог содержать **10-разрядное десятичное число**.

Назначение: **Баллистические расчеты.**

# EDVAC (Electronic Discrete Variable Computer):

**ENIAC** программировалась извне

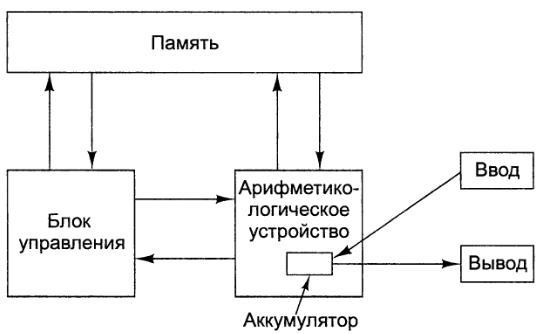
**EDVAC** = Электронная дискретная параметрическая машина

**Экерт**, **Моушли**, **Джон фон Нейман** и др. разработали EDVAC (1944) для решения этой проблемы.

**Десятичная арифметика**, используемая в машине ENIAC, где каждый разряд представлялся 10 электронными лампами (1 включена и 9 выключены): **бинарная арифметика**.

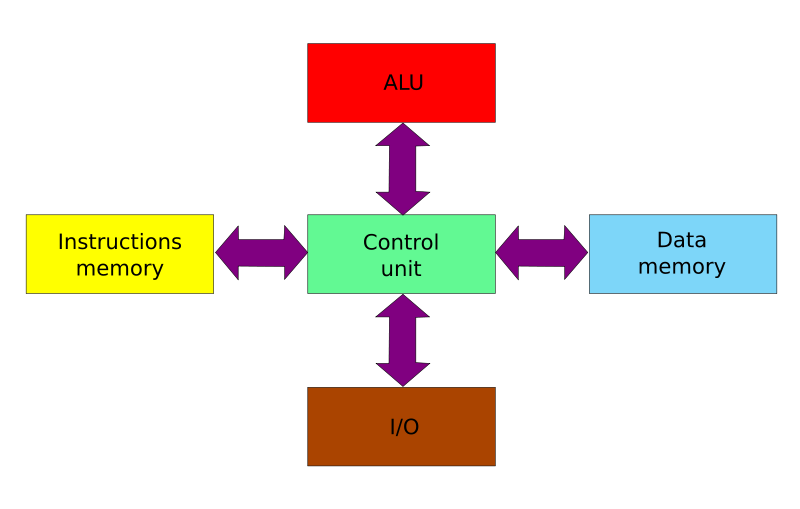
# Фон-неймановская ВМ:

Использовалась в **EDSАС**, первой машине с программой в памяти. Является **основой** большинства современных цифровых компьютеров.



Память включала **4096 слов**, каждое слово содержало 40 битов. Каждое слово содержало: **2 команды по 20 битов** (8 бит - тип команды, 12 бит – адрес слова из памяти (из 4096 слов)), или **целое число со знаком на 40 битов** . Внутренний регистр в 40 битов - **аккумулятор**.Типичная команда добавляла слово из памяти к аккумулятору или сохраняла содержимое аккумулятора в памяти.

# Гарвардська архітектура:



**Распределенная память – данные и команды**

# IBM – коммерциализация:

**SSEC = Selective Sequence Electronic Calculator**  (хранила 150 слов. Команды и данные читались с лент)

**IBM-701** (2048 слов по 36 битов. каждое слово содержало две команды. стал первым компьютером, лидирующим на рынке в течение десяти лет.)

**IBM-704** (4 Кбайт памяти на магнитных сердечниках, команды по 36 битов и процессор с плавающей точкой.)

В 1958 году - компания IBM начала работу над последним компьютером на электронных лампах,  
**IBM-709** (представлял собой усложненную версию IBM-704.)

# Второе поколение — транзисторы (1955-1965):

# Транзисторы:

**Транзистор** был изобретен сотрудниками лаборатории Bell Laboratories: **Джоном Бардином, Уолтером Браттейном и Уильямом Шокли**. в 1956 году они получили **Нобелевскую** премию в области физики. **Транзистор** может работать как очень **быстрый бинарный переключатель.**

# Первые транзисторные компьютеры:

Первый компьютер на транзисторах был построен в лаборатории МИТ. Он содержал **слова из 16 битов.** Компьютер назывался **ТХ-0** (Transistorized experimental computer 0 — экспериментальная транзисторная вычислительная машина 0). Один из инженеров из этой лаборатории, Кеннет Ольсен, в 1957 году основал компанию **DEC:** **Digital Equipment Corporation** — корпорация по производству цифровой аппаратуры. **PDP-1** – серийная машина, сходная с ТХ-0.

# DEC: машина PDP-1:

**Programmed Data Processor = PDP**. Появился только в 1961 году. Память **4 Кбайт слов по 18 битов**. Время цикла **5 микросекунд**. Этот параметр был в два раза меньше, чем у IBM-7090, транзисторного аналога IBM-709. PDP-1 был самым быстрым компьютером в мире в то время. **PDP-1** стоил **$120000**, a IBM-7090 стоил миллионы!!! Компания DEC продала десятки компьютеров PDP-1, и так появилась компьютерная промышленность.

# DEC: машина PDP-8:

Через несколько лет DEC разработал модель **PDP-8 :** **12-битный** компьютер, PDP-8 стоил гораздо дешевле, чем PDP-1 (**$16000**). Главное нововведение — **одна шина** (Omnibus). **Шина** (набор параллельно соединенных проводов для связи компонентов компьютера). Было продано 50000 компьютеров!!!

# IBM:

**IBM-7090**: транзисторная версия IBM-709

**IBM-7094**: У нее время цикла составляло 2 микросекунды. память состояла из 32 К слов по 16 битов. В основном использовалась для научных расчетов.

**IBM-1401:** использовалась для коммерческих расчетов. могла считывать и записывать магнитные ленты и перфокарты и распечатывать результат. не было регистров и фиксированной длины слова. память содержала 4 Кбайт по 8 битов (4 Кбайт).

# Третье поколение — интегральные схемы (1965-1980):

# Влияние:

Изобретение **кремниевой интегральной схемы** в 1958 году (изобретатель — Роберт Нойс): дало возможность помещать **десятки транзисторов** на одну небольшую микросхему.

Компьютеры на интегральных схемах: были **меньшего размера**, работали **быстрее** и стоили **дешевле**, чем их предшественники на транзисторах.

# IBM: проблема совместимости:

Компьютеры IBM-7094 и IBM-1401 были **несовместимы**!: разные наборы команд. разные системы ввода-вывода. разные компиляторы, библиотеки, ...

Серия компьютеров на транзисторах - **System/360**

# IBM 360:

Нововведения: **один и тот же язык** (ассемблер). **унифицированные** способы I/O . каждая новая модель была **больше по размеру** и **по мощности**, чем предыдущая. **Мультипрограммирование** (в памяти компьютера могло находиться одновременно **несколько** **программ).**

# IBM 360: A General-Purpose Register (GPR) Machine:

Процессор : **16 32-разрядных регистра общего назначения*.* 4 64-разрядных регистра для чисел с плавающей точкой.**

1. **разрядная** машина с адресной шиной **24 бита**

Формат данных: 8 бит = байт, 16 бит= пол-слова, 32 бит слово, 64 бит = двойное слово.

# IBM 360: способы адресации:

opcode

R1

8

4

4

R2

RR

R1¬ (R1) op (R2)

opcode

R

X

B

D

8

4

4

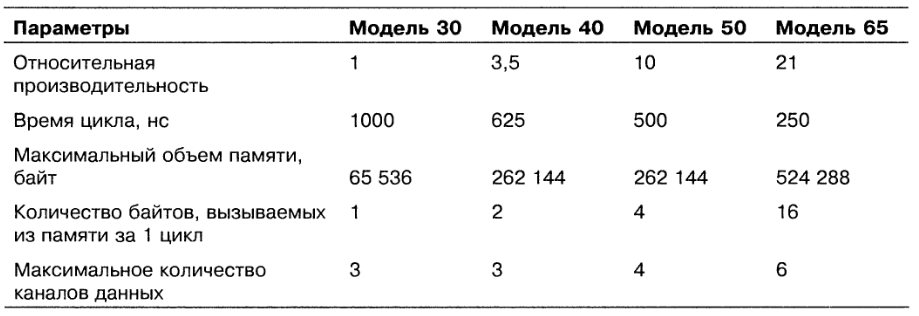
12

4

RD

# IBM-360:

Объем памяти у IBM-360 составлял 2^24 байтов (16 Мбайт).



# Идексные регистры:

Tom Kilburn, Manchester University, mid 50’s. Релизует след. Принцип: инф-я, находящаяся в процессоре в **конце** **выполнения** некоторой **инструкции**, используется для **выполнения следующей инструкции**: One or more **specialized registers** to **simplify address calculation .**

Index registers : special kind of **indirect addressing** an **immediate** constant (i.e. which is part of the instruction itself) is **added** to the contents of a register to **form** the address to the actual operand or data

In early computers : array operations => **linearly** **repeating** program code for **each** array element or by using rather "dirty" **self-modifying code** techniques

Позволяют **уменьшить** к-во операций: вызов инструкций из памяти, операндов, ...

Такие регистры **подобны аккумуляторам**

# Эволюция способов адресации:

1. Single accumulator, absolute address: LOAD x

2. Single accumulator, index registers: LOAD x, IX

3. Indirection: LOAD (x) AC ¬ M[M[x]] (для процедур)

4. Multiple accumulators, index registers, indirection: LOAD R, IX, x

5. Indirect through registers: LOAD RI, (RJ)

6. The works: LOAD RI, RJ, (RK) RJ = index, RK = base addr

# Форматы инструкций (команд):

***Two address formats****:* the destination is same as one of the operand sources

(Reg × Reg) to Reg RI  ¬ (RI) + (RJ)

(Reg × Mem) to Reg RI ¬ (RI) + M[x]

x can be specified directly or via a register. effective address calculation for x could include indexing, indirection, ...

***Three address formats****:* One destination and up to two operand sources per instruction

(Reg x Reg) to Reg RI ¬ (RJ) + (RK)

(Reg x Mem) to Reg RI ¬ (RJ) + M[x]

# Другие форматы команд:

***Zero address formats****:* operands on a stack

add M[sp-1] ¬ M[sp] + M[sp-1]

load M[sp] ¬ M[M[sp]]

Stack can be in registers or in memory (usually top of stack cached in registers)

***One address formats****:* Accumulator machines. Accumulator is always other **implicit** operand.

# Четвертое поколение — сверхбольшие интегральные схемы (1980-?):

# СБИС:

Появление **сверхбольших интегральных** схем **(СБИС)** в 80-х годах позволило помещать на одну плату сначала десятки тысяч, затем сотни тысяч и, наконец, миллионы транзисторов.

**Началась эра персональных компьютеров!!!**

# Персональные компьютеры:

Первые **персональные** **компьютеры** продавались в виде **комплектов**: печатную плату, набор интегральных схем, обычно включающий схему Intel 8080, несколько кабелей, источник питания и иногда 8-дюймовый дисковод. **Сложить из этих частей компьютер покупатель должен был сам!!!** Продавались фирмой **Micro Instrumentation and Telemetry Systems (MITS). Paul Allen** and **Bill Gates** also saw the magazine and began writing software for the Altair computer.

**Apple I** был разработан **Стивом Джобсом** и **Стивом Возняком**. Стоил – **$666.66.** Продавался **полностью собранным .**

Apple II: Усовершенствие Apple I. Продавался до 1993 г.

Macintosh : **Мышь** + **GUI.**

# IBM PC:

Проект по «сборке» компьютера ***Project Chess*** из существующих компонентов и технологий: ***IBM PC Technical Reference Manual*** . В качестве **центрального** **процессора** - **Intel 8088** и был создан персональный компьютер из серийных компонентов. Этот компьютер (IBM PC) появился в 1981 году и стал самым покупаемым компьютером в истории.

# Архитектура команд:

В середине 80-х годов на смену **CISC** пришел **RISC**. Complex instruction set computer (**CISC**)— компьютер на микропроцессоре с **полным набором команд** : instruction can execute several low-level operations, such as a load from memory, an arithmetic operation, and a memory store, **all in a single instruction**, **System/360**, **PDP-11**, **VAX**, **Motorola** **68000**, and **x86.**

Reduced instruction set computer (**RISC**) — компьютер с **сокращенным набором команд** :**simplified instructions** can provide higher performance if this simplicity enables much faster execution of each instruction. **DEC Alpha**, **AMD 29k**, **ARC**, **ARM**, **Atmel** AVR, **MIPS**, **PA-RISC**, **Power**, **SuperH**, **SPARC**…

# Семейство процессоров Intel (1/3):





**64-bit processors: IA-64- Itanium** / **Itanium 2** (Released May 29, 2001. 733 MHz to 1.73 GHz . 2MB cache)

**64-bit processors: Intel 64 – NetBurst**  (Mostly compatible with AMD's AMD64 architecture. Introduced Spring 2004.)

**Pentium 4F** (2.8–3.8 GHz или 3.0–3.6 GHz . 2MB cache)

**Pentium D**  (Dual-core microprocessor . No Hyper-Threading . **Smithfield** – **90 nm** process technology (2.66–3.2 GHz). **Presler** – **65 nm** process technology (2.8–3.6 GHz))

# СКИТ:

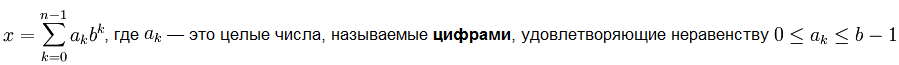
**СКИТ - 3**: 127-узловой кластер на многоядерных процессорах (75 узлов на двухъядерных процессорах Intel Xeon 5160. 52 узла на четырехъядерных процессорах Xeon 5345).Число **ядер-процессоров** в кластере **716 ядер**.

**СКИТ – 2**: 32-узловой кластер на микропроцессорах Intel Itanium2.

**СКИТ – 1**: 24-узловой кластер на микропроцессорах Intel Xeon .

# Позиционная система счисления:

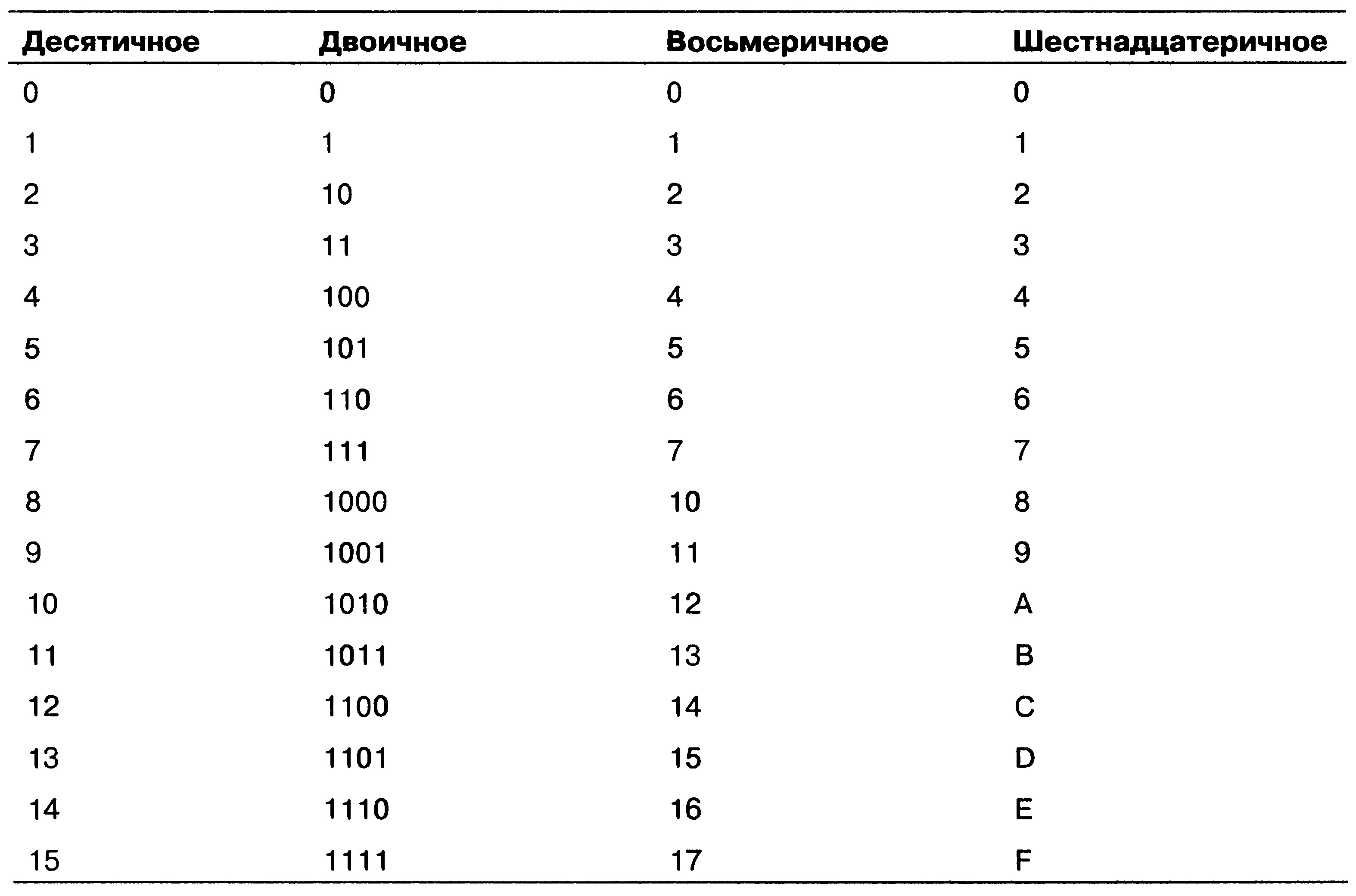
Целое число *x* в *b*-ричной системе счисления представляется в **виде конечной линейной комбинации степеней числа** *b.*



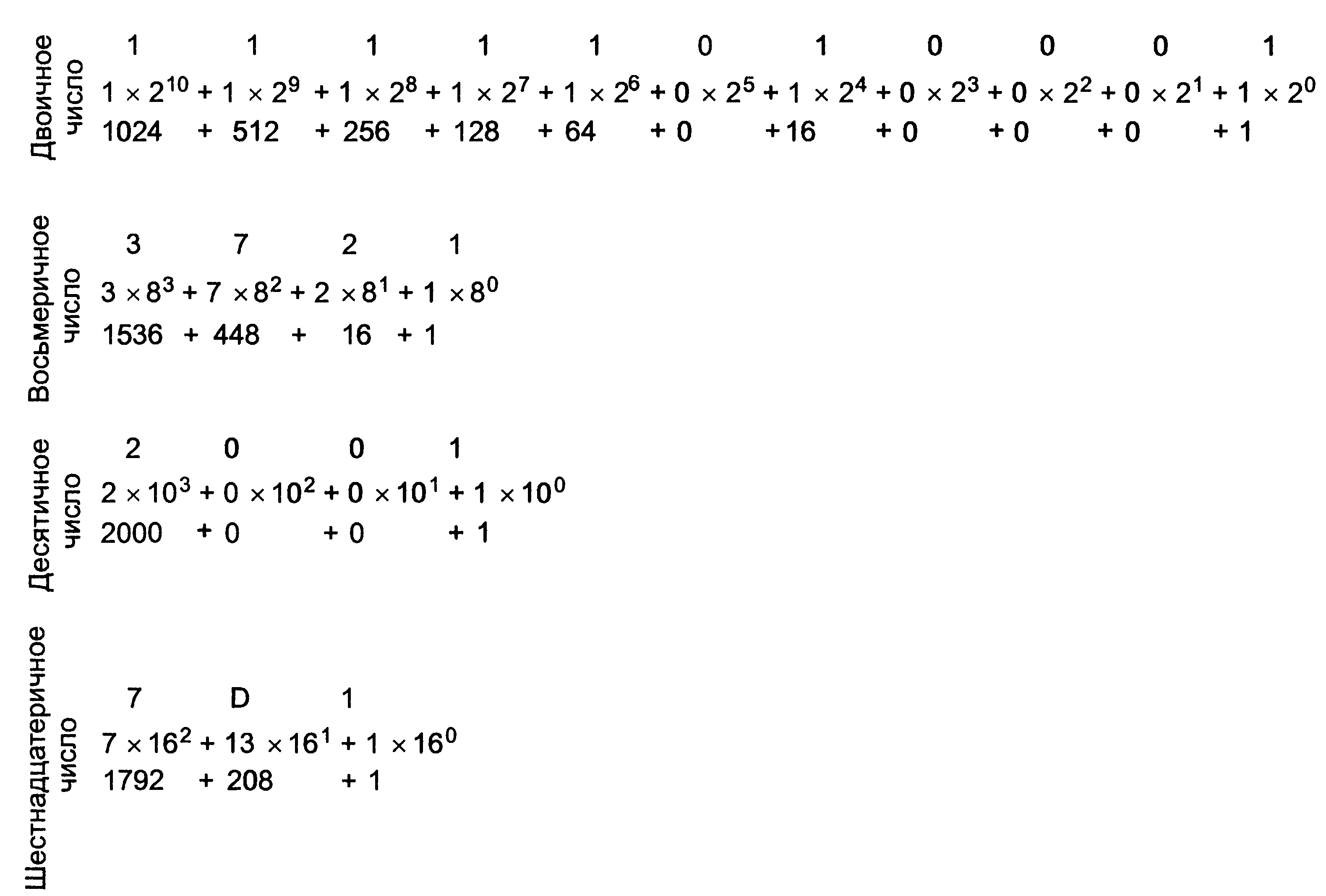
 

Важные системы счисления: двоичная, восьмеричная и шестнадцатеричная.

# 2, 8, 10, 16 представление:



# Число 2001:



# Правила:

2: 1 + 1 = 10

8: 7 +1 = 10

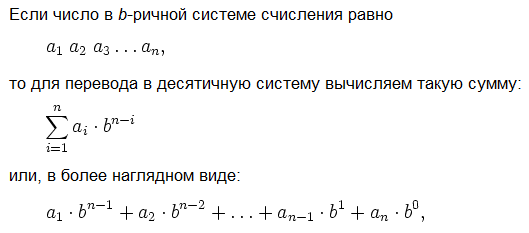
16: F + 1 = 10

(1) F + B = F + 1 + A = 10 + A = 1A

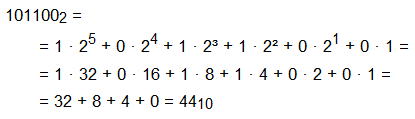
(2) 1 + 9 + A = A + A = 14

(3) 1 + A + 3 = E

# Преобразование чисел из одной системы счисления в другую:

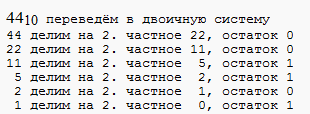
Перевод произвольной позиционной системы счисления в десятичную: 

Например:



# Перевод из десятичной в произвольную позиционную систему счисления:

Целая часть: 1) Последовательно **делить** **целую** часть десятичного числа на **основание**, пока десятичное число не станет равно нулю. 2) Полученные при делении **остатки** являются **цифрами** нужного числа. Число **в новой системе** записывают, начиная с **последнего остатка**.







# Перевод из двоичной в восьмеричную и шестнадцатеричную системы и наборот:

Для восьмеричной: разбиваем переводимое число на количество цифр, равное степени 2, т.е. 3.

Для шестнадцатеричной: на 4.





# Отрицательные двоичные числа:

Существует несколько систем. Рассмотрим дополнение до двух

**Х + (-Х) = 0**

Правило преобразования в отрицательное число: Сначала каждая единица меняется на ноль, а каждый ноль — на единицу. Затем к полученному результату прибавляется единица.

Число +6: 00000110 🡪 11111001

-6: 11111010

Проверка: 00000110 + 11111010 = 0

# Числа с плавающей точкой:

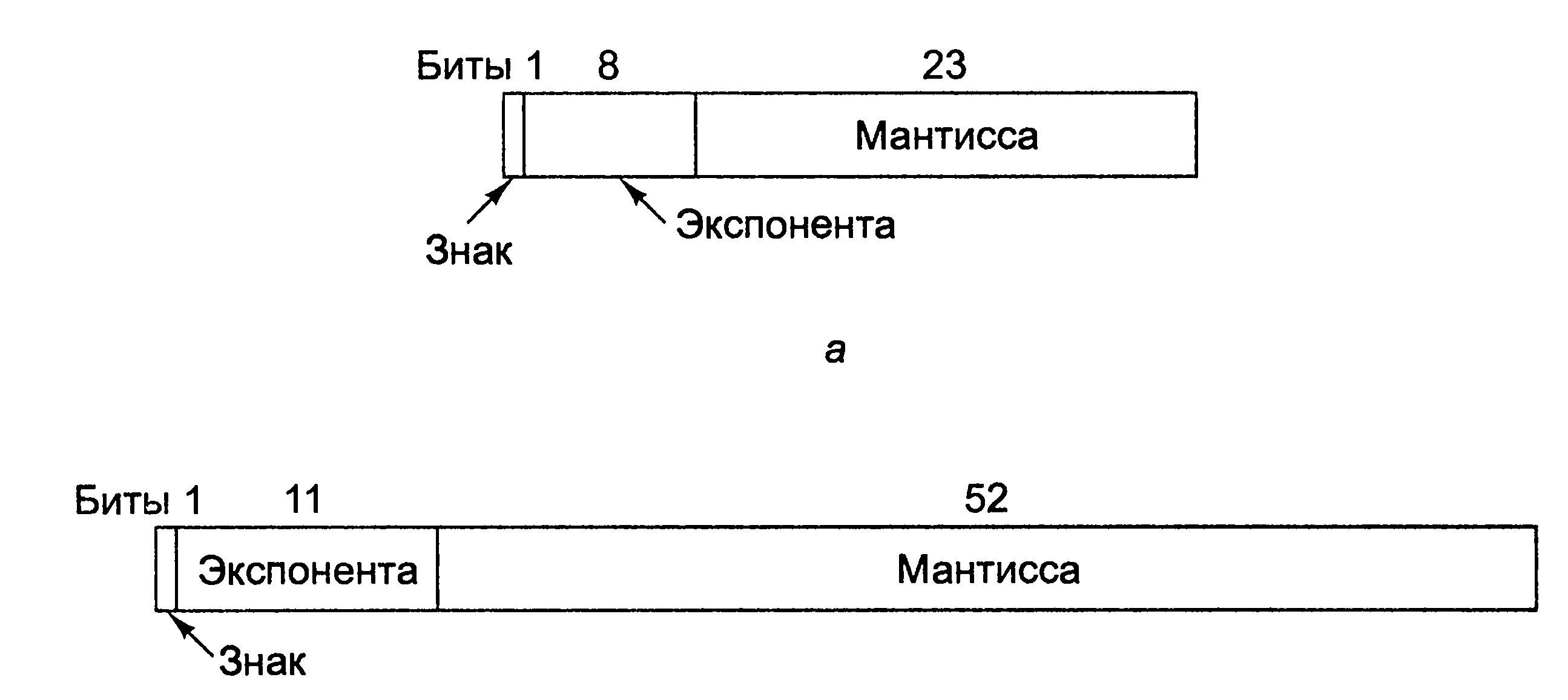
Дробное число хранится в форме **мантиссы** и **показателя степени**. Стандарт **IEEE 754**.

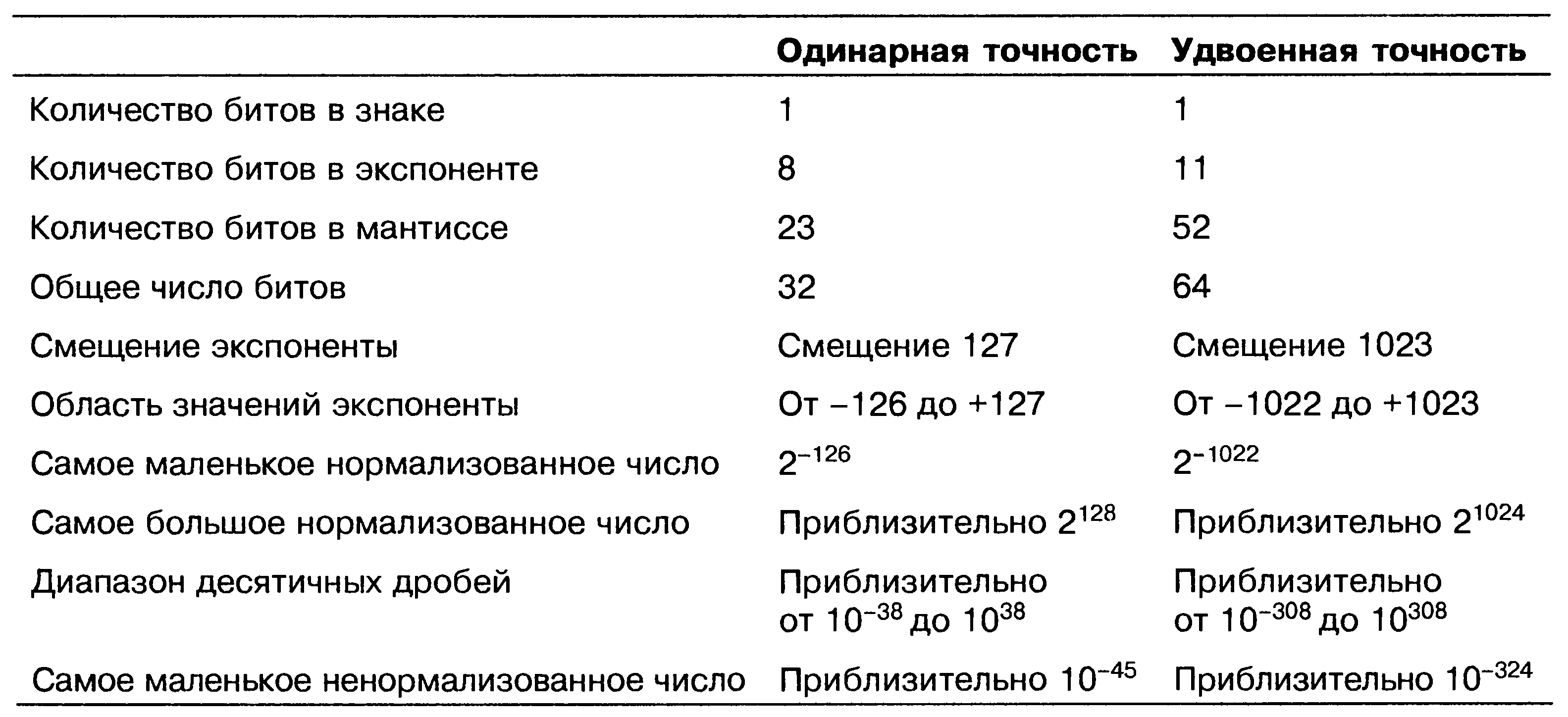
Реализация математических операций с числами с плавающей запятой в вычислительных системах может быть как **аппаратная**, так и **программная**.

# IEEE 754:

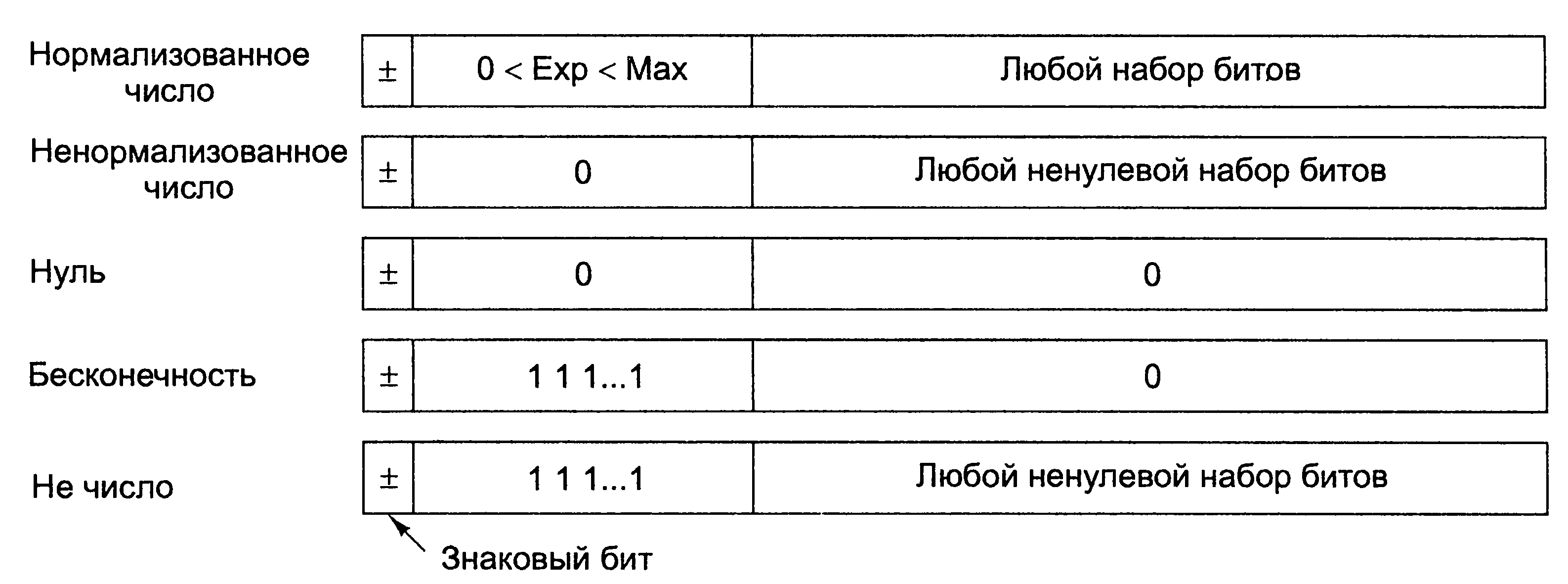
Определения форматов хранения мантиссы, экспоненты и знака, форматы положительного и отрицательного нуля, плюс и минус бесконечностей, а также определение "не числа" (NaN). Методы, которые будут использоваться для округления числа в процессе математических операций. Обработка исключительных ситуаций, таких как деление на ноль, переполнение и т.д.

Одинарная точность :

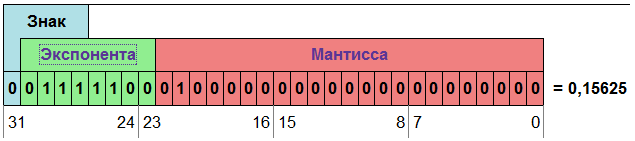




# IEEE 754 – типы:



# Преобразование из IEEE 754:



Экспонента: 011111002 = 12410

Мантисса: 01000000000000000000000

Для вычисления показателя степени из восьмиразрядного поля экспоненты вычитается смещение экспоненты равное 12710: 011111002 - 011111112 = 12410 - 12710 = -310.

Для вычисления мантиссы к единице добавляется дробная часть мантиссы из 23-х разрядного поля дробной части мантиссы 1,01000000000000000000002.

Число равно произведению мантиссы со знаком на двойку в степени экспоненты

1,012\*2-3 = 1012\*2-5 = 5\*2-5 = 0,15625

1,012\*2-3 = (1 + 0\*2-1 + 1\*2-2) \* 2-3 = 2-3 + 2-5 = 0,15625

1. = 1101 = 1.101 \* 2^3

Экспонента: 3 + 127 = 130 = 10000010

Мантисса: 10100000000000000000000

Число 13 в IEEE 754: 0.10000010.10100000000000000000000

**155.625**

* + 1\*27 +0\*26+0\*25+1\*24+1\*23+0\*22+1\* 21+  
    1\*20+1\*2-1+0\*2-2+1\*2-3
  + 128 + 0 + 0 + 16 + 8 + 0 + 2 + 1 + 0.5 + 0 + 0.125
  + **155.625 = 10011011.101** - число в двоичной системе с плавающей точкой
  + **10011011.101=1.0011011101\*2+7 =   
    = 1.0011011101\*2+111**  
    число в нормализованном экспоненциальном виде
  + смещенная экспонента  
    7 + 127 = 134 = 100001102
    - 710 = 1112

1 бит

0

8 бит

10000110

23 бит

00110111010000000000000

# Примеры – Python:

**>>> a = 2\*\*1000  
>>> a  
10715086071862673209484250490600018105614048117055336074437503883703510511249361224931983788156958581275946729175531468251871452856923140435984577574698574803934567774824230985421074605062371141877954182153046474983581941267398767559165543946077062914571196477686542167660429831652624386837205668069376L  
  
>>> x = 0.1  
>>> x  
0.10000000000000001  
  
>>> s=0.0  
>>> for i in range(10): s += 0.1  
>>> s  
0.99999999999999989  
>>> print(s)  
1.0**

**>>> import math  
>>> a = math.sqrt(2)  
>>> a  
1.4142135623730951  
>>> a\*a == 2  
False  
>>> a\*a  
2.0000000000000004**

# Примеры – C++:

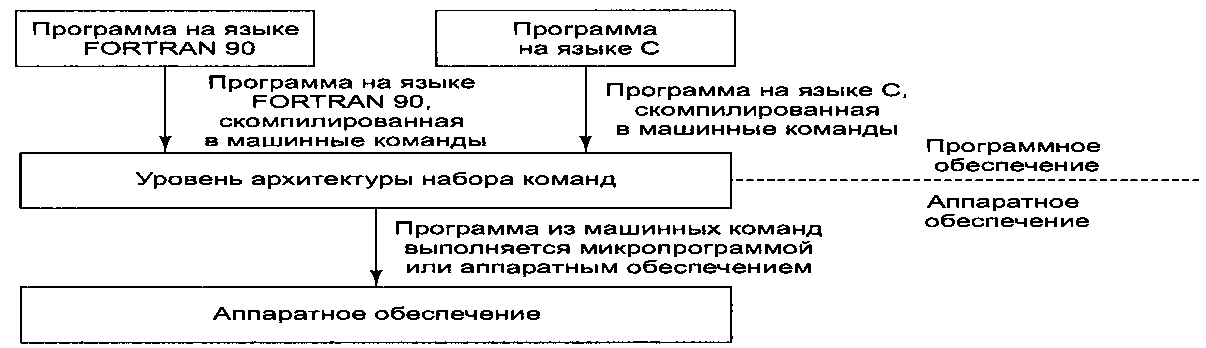
**int main(int argc, char\* argv[])  
{  
  
 double x = 0.1;  
   
 cout << x << endl;  
 printf("%1.17lf \n", x);  
   
 double sum = 0.0;  
   
 for (int i = 0; i < 10; i++)  
 sum += 0.1;  
   
 cout << sum << endl;  
 printf("%1.17lf \n", sum);  
 return 0;  
}**

# Уровень ISA:

Уровень архитектуры набора команд: **Instruction Set Architecture.**

Расположен между уровнями **микроархитектуры** и **операционной системы.**

Является связующим звеном между программным (**компиляторы**) и аппаратным обеспечением.



# Вопросы при создании ISA:

При появлении новой машины возникают вопросы: «Совместима ли машина с **предыдущими** версиями?», «Можно ли запустить на ней **прежнюю** **ОС**?», «Будут ли работать на этой машине **прежние** **приложения** и не потребуется ли заменять их **новыми** **версиями**?».

# Общий обзор ISA:

Программа уровня **архитектуры набора команд** : то, что получается в **результате** работы **компилятора**

Чтобы получить программу уровня ISA, создатель компилятора должен знать: какая **модель памяти** используется в машине, какие **регистры**, типы **данных** и **команды** имеются в наличии, и т. д.

Например, системы V9 SPARC (Version 9 SPARC) и JVM имеют официальные определения. Цель такого официального документа — дать возможность различным производителям выпускать машины данного типа, чтобы эти машины могли выполнять одни и те же программы и получать при этом одни и те же результаты

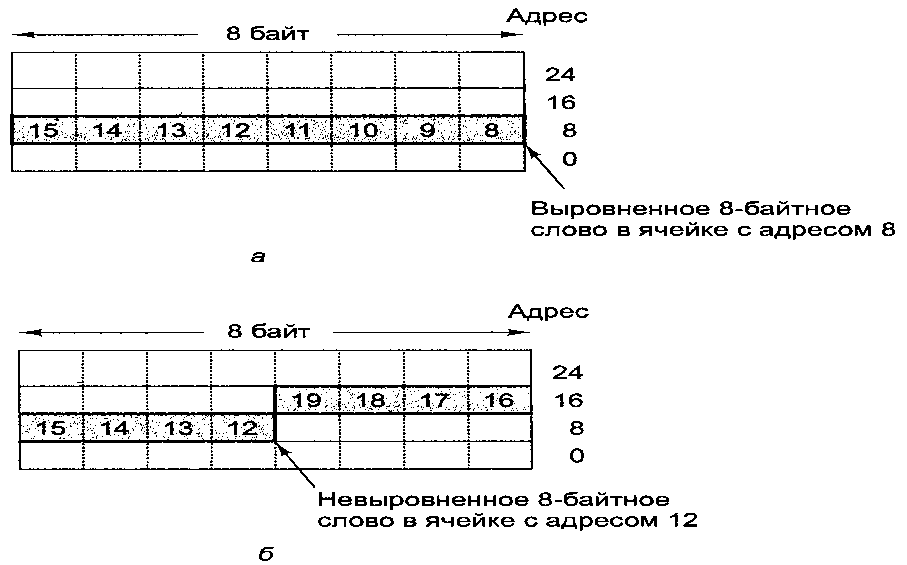
# Модели памяти:

Во всех компьютерах память разделена на ячейки, которые имеют последовательные адреса.

**Обычно размер ячейки — 8 бит = 1 байт.**

Байты обычно группируются в 4-байтные (32-разрядные) или 8-байтные (64-разрядные) слова **с командами манипулирования целыми словами**

Многие архитектуры требуют, чтобы слова **были выровнены в своих естественных границах**. Так, 4-байтное слово может начинаться с адреса 0, 4, 8 и т. д., но не с адреса 1 или 2.



**Выравнивание** адресов требуется довольно **часто**, поскольку при этом память работает наиболее **эффективно.**

Pentium 4: вызывает из памяти по 8 байт за обращение, использует **36-разрядные физические адреса**, но содержит только **33 адресных бита.** не сможет обратиться к невыровненной памяти напрямую, поскольку **младшие 3 бита** явным образом не определены. Эти биты всегда **равны 0**, и все адреса памяти кратны значению **8 байт**.

# Регистры:

RISC-машины: высокоскоростные процессоры и с относительно медленной памятью содержат как **минимум 32 регистра общего назначения.** в некоторых машинах регистры общего назначения полностью **симметричны** и **взаимозаменяемы**. В других машинах некоторые регистры общего назначения могут быть специализированными. В Pentium 4 имеется регистр EDX, который получает половину произведения при умножении и половину делимого при делении.

# Команды:

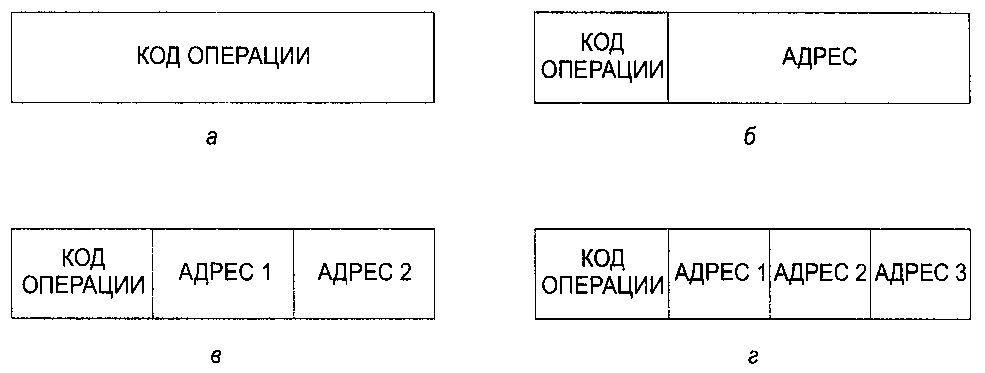
Типы: **LOAD** и **STORE**(предназначены для перемещения данных между памятью и регистрами), команда **MOVE** (служит для копирования данных из одного регистра в другой : **mov r1, r2 (r1 <- r2) ),** **арифметические** и **логические** команды (**add r1, r2, r3 (r1 <- r2 + r3)).**

Типы: команды **сравнения** элементов данных (**cmpl $1000, r1 )**, команды **переходов** в зависимости от результатов (**jne .L1**, **jump .L1).**

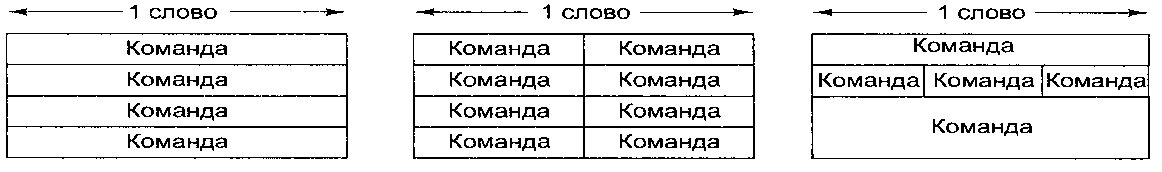
# Форматы команд:

Команда состоит из: **кода операции**, **операнды**, **результат.**

Процесс определения, где находятся операнды (то есть их адреса), называется **адресацией** .

Способы адресации: 

Отношения между размерами команды и слова:



# Критерии разработки форматов команд:

**минимизации длины** команд; **достаточный объем пространства** в формате команды для представления всех требуемых операндов; выбор числа битов в адресном поле. (современные системы требуют, чтобы адреса были у отдельных байтов, но при обращении к памяти **всегда считываются одно, два, а иногда даже четыре слова сразу.** UltraSPARC III (в результате считывания 1 байта из памяти единовременно вызываются **минимум 16 байт**, а иногда и **вся строка кэш-памяти размером 64 байта** ))

# Расширение кода операций:

Пусть n+k бит = код операции в k бит и одним адресом в n бит

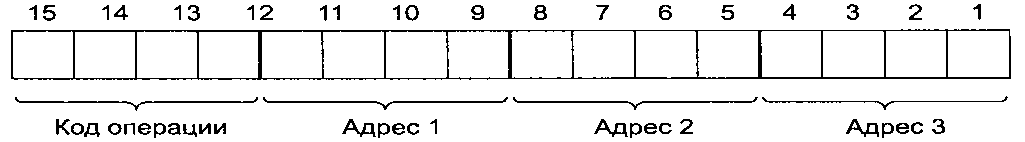
возможные интерпретации: k + 1 бит и адрес в n – 1, k - 1 бит и адрес в n + 1

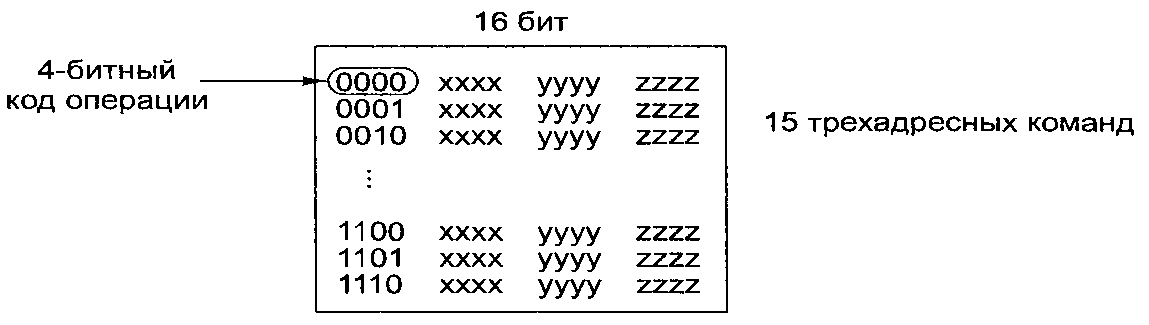
Такой механизм назыв. **расширением кода операций**

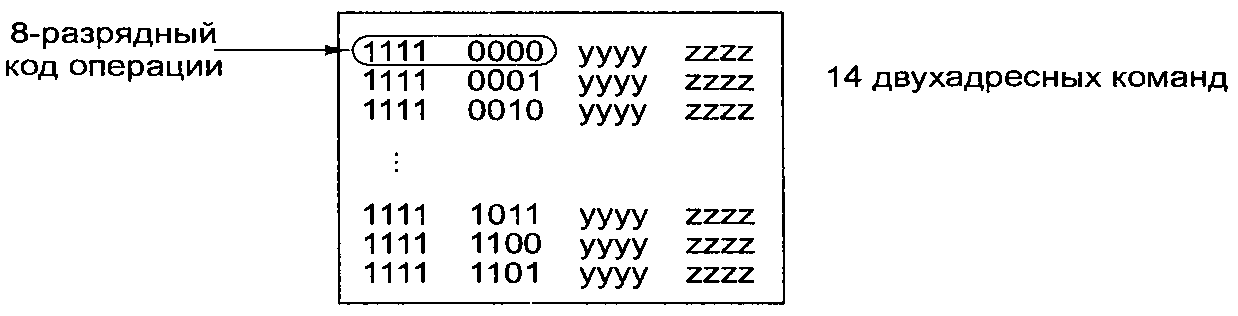
# Пример:

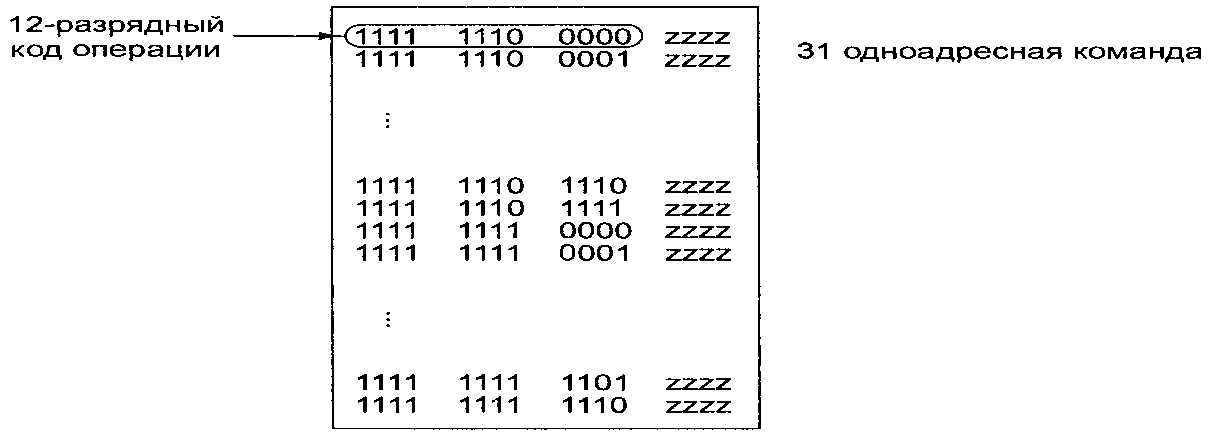
4-разрядный код операции и три 4-разрядных адресных поля (для 16 регистров)

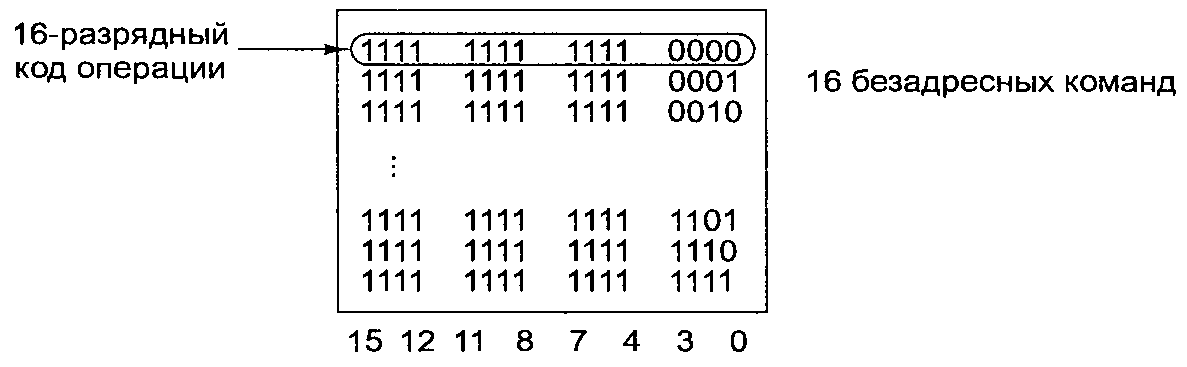
Возможный вариант: 16 трехадресных команд .











# Пример: вывод:

**Трехадресные** команды имеют **4-разрядный** код операции, **двухадресные** команды — **8-разрядный**, **одноадресные** команды — **12-разрядный**, а **безадресные** команды —**16-разрядный** . Идея расширения кода операций наглядно демонстрирует **компромисс** между пространством для **кодов операций** и пространством для **другой информации**.

# Проблемы Pentium 4 IA-32:

Типичная CISC-архитектура с **командами** **разной** длины и **огромным** количеством **различных** **форматов**: => переход к RISC-архитектуре.

IA-32 ориентирована на **двухадресные** команды : => отдельно команды **загрузки/сохранения** + **3-адресные регистровые команды.**

**Небольшой** и **нерегулярный** набор регистров. Сложности обработки **WAR-**зависимостей. **Многостадийный** **конвейер** (до 20 стадий).

# Pentium 4:

**80386**: первая **32-разрядная машина**, выпущенная компанией Intel

**IA-32**: 80486, Pentium, Pentium Pro, Pentium II, Pentium III, Pentium 4, Celeron, Xeon, Pentium M, Centrino и т. д.

Три операционных режима: 1) в **реальном режиме** все функции, которыми был наделен процессор со времен 8088, отключаются, и Pentium 4 работает как простой процессор 8088. 2)режим **виртуального процессора 8086**, который делает возможным исполнение старых программ, написанных для 8088, но с защитой. 3) **защищенный режим**, в котором Pentium 4 работает как Pentium 4, а не как 8088.

# Pentium 4 – защищенный режим:

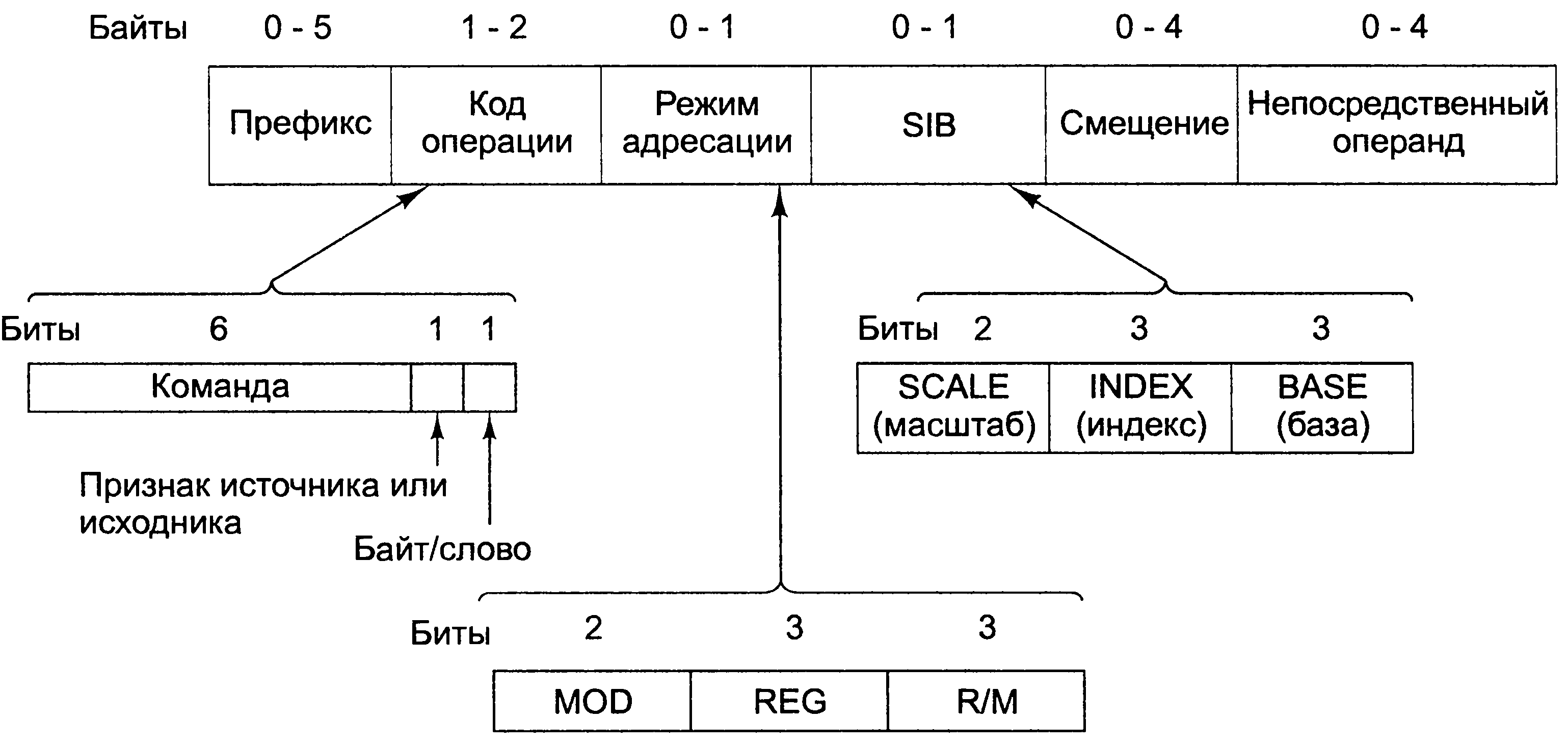
В этом режиме доступны 4 уровня привилегий, за- задаваемые битами во флаговом регистре (PSW).

Уровень 0 : соответствует привилегированному режиму на других компьютерах и обеспечивает полный доступ к машине. Используется операционной системой.

Уровень 3: предназначен для пользовательских программ. Блокируется доступ к определенным командам и регистрам управления, чтобы сбой какой-нибудь пользовательской программы не привел к краху всей системы.

Уровни 1 и 2: Не применяются.

# Формат команд: Pentium 4:



effective address (EA) is Base + (Index \* Scale) + Displacement

# Пример ADD:

Код операции

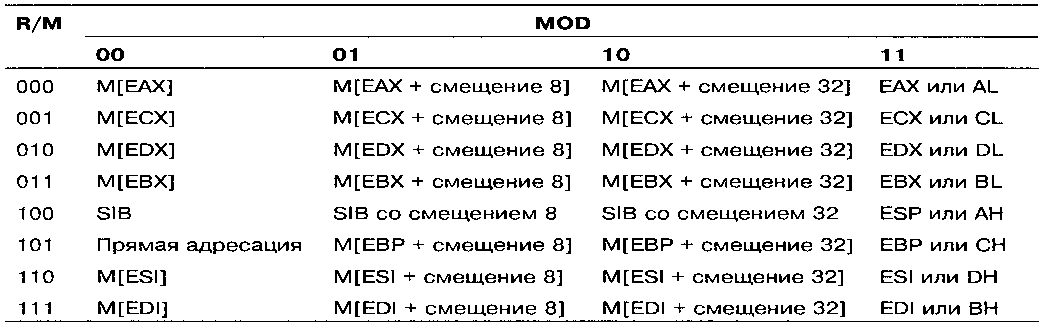
|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
| **0** | **0** | **0** | **0** | **0** | **0** | **d** | **s** |

If **d = 0** then the destination operand is a memory location: **add [ebx], al**

If **d = 1** then the destination operand is a register: **add al, [ebx]**

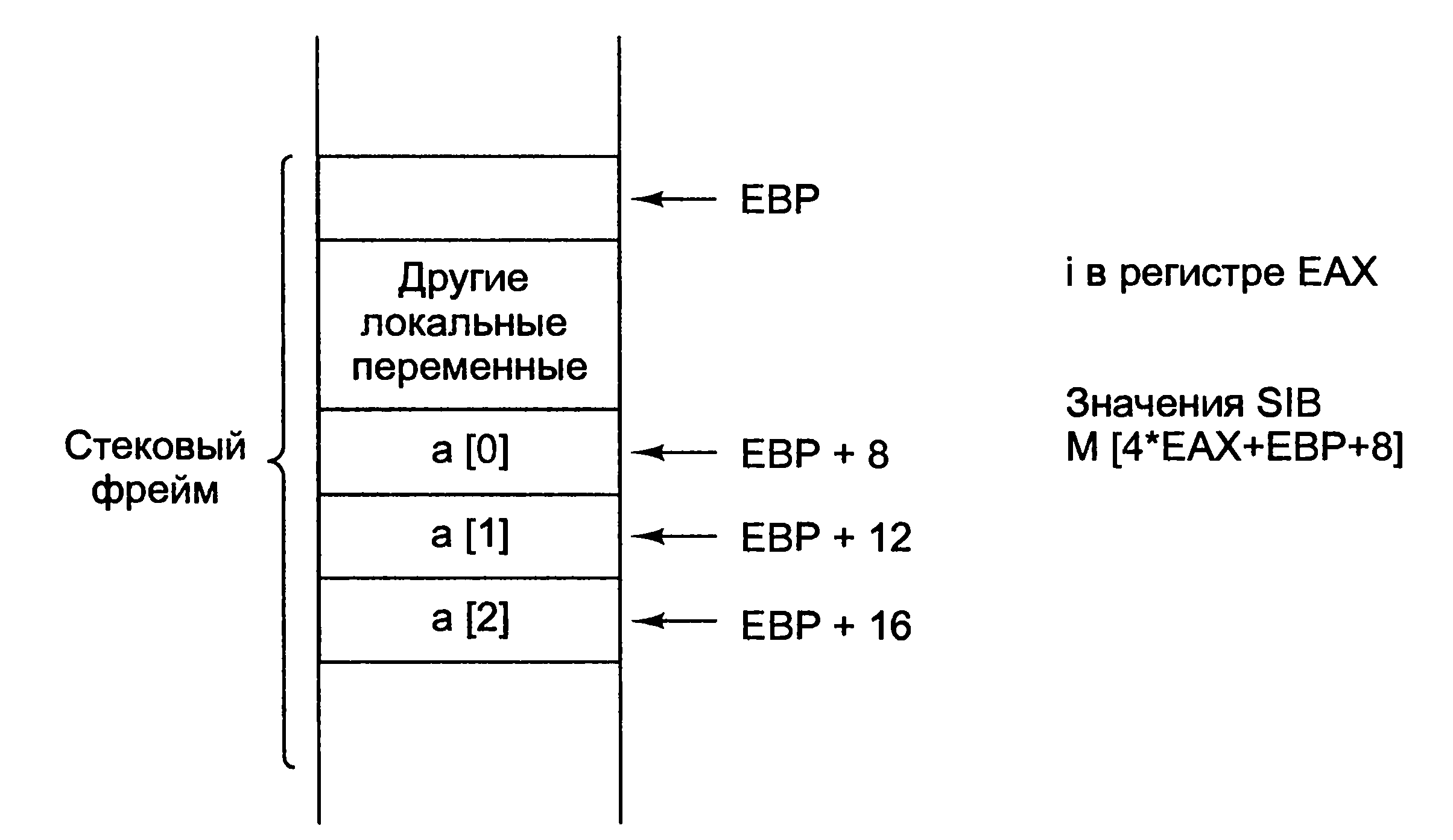
If **s = 0** -> 8-bit operand  
If **s = 1** -> 16- (with *prefix*) or 32-bit operands

# R/M, mod:



М[х] — это слово в памяти с адресом х

# SIB:



указатель на базу стекового фрейма, который содержит локальные переменные и массивы

# IA-64 (Itanium 2):

Новая **64-разрядная архитектура**. **Не совместима с x86-64**!!! Перенос нагрузки с периода выполнения **на компилятор**!!!

EPIC : **Explicitly Parallel Instruction Computing — вычисления с явным параллелизмом команд.**

# Itanium 2 – Память:

Всего предусмотрено 264 байт линейной памяти.

Имеющиеся команды позволяют обращаться к блокам памяти размером 1,2, 4, 8, 16 и 10 байт

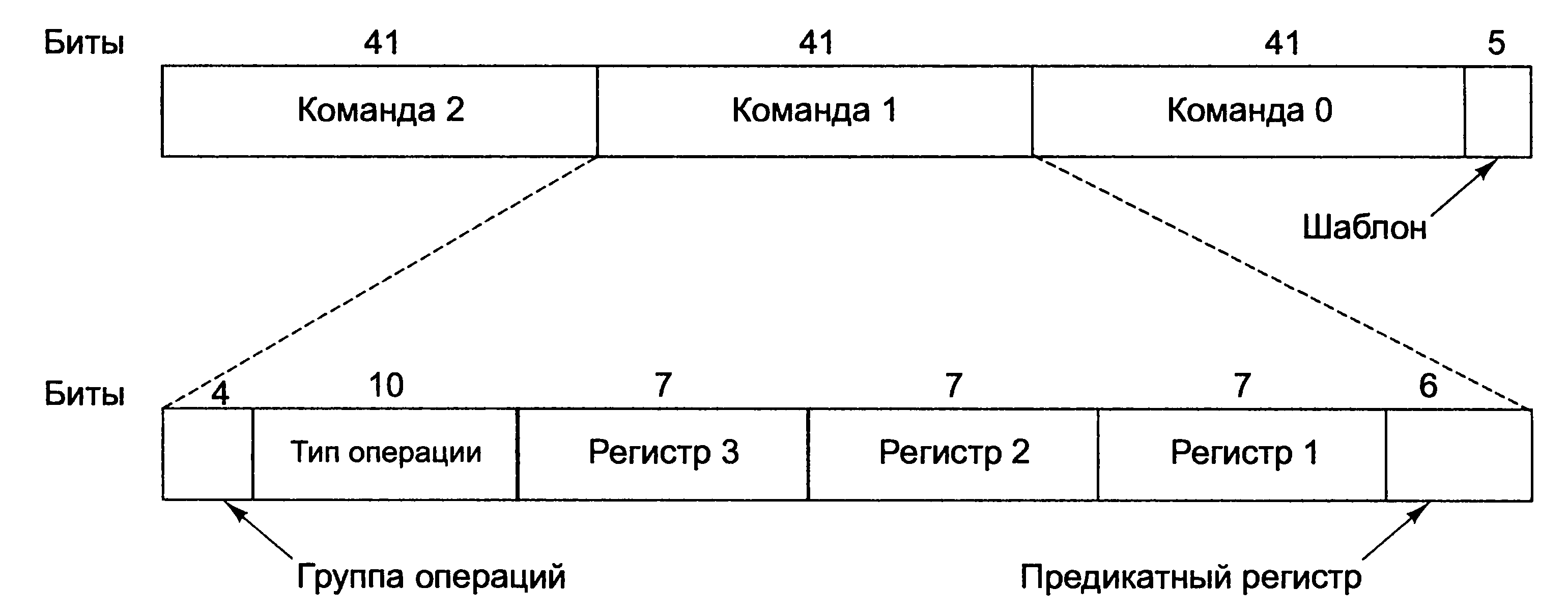
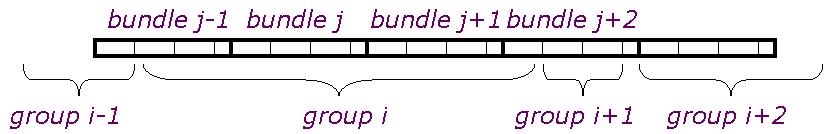
1. -разрядных регистров общего назначения. (Первые **32** из них - **статические**, оставшиеся **96** группируются в **стек регистров)**

Каждая процедура получает доступ к 32 статическим регистрам и некоторому (переменному) регистров, распределяемых динамически.

**128 64/80-разрядных регистра с плавающей точкой**, организованных по стандарту IEEE 745 и не группируемых в стек.

**64 1-разрядных предикативных** регистра.

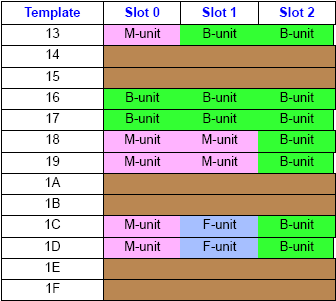
# Itanium 2 - Планирование команд:

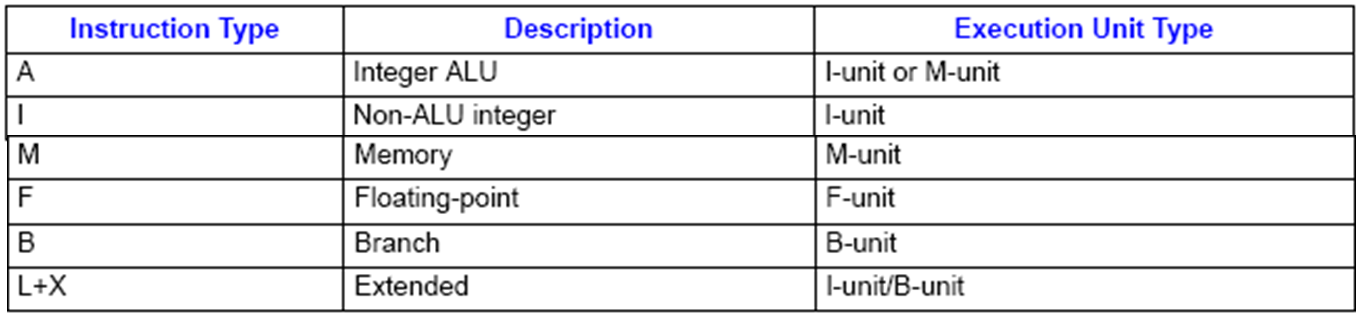


VLIW = Very Long Instruction Word

* Команды объединяются в 128-разрядные пучки (bundles)

# Itanium 2 – Шаблон:





# Itanium 2 – Предикация:

Используется для **сокращения** условных **переходов**. В предикатной архитектуре команды содержат **условия**, которые сообщают, в каком случае нужно выполнять команду, а в каком — нет. Предикатные команды могут помещаться в **конвейер** последовательно без каких-либо **проблем** и **простоев**.

Исходный код  
**if (R1 == 0)  
 R2 = R3;**

Ассемблер  
**CMP R1, 0  
BNE L1  
MOV R2, R3  
L1:**

Условная команда  
**CMOVZ R2,R3,R1**

**if(R1 == 0) {  
 R2 = R3;  
 R4 = R5;  
} else {  
 R6 = R7;   
 R8 = R9;  
}**

**CMP R1,0  
BNE L1  
MOV R2,R3  
MOV R4,R5  
BR L2  
L1: MOV R6,R7  
 MOV R8,R9  
L2:**

* **CMOVZ R2,R3,R1  
  CMOVZ R4,R5,R1  
  CMOVN R6,R7,R1  
  CMOVN R8,R9,R1**

Механизм: Каждая команда действительно выполняется, в самом конце конвейера, когда уже нужно сохранять результат в выходном регистре, производится проверка, истинно ли предсказание. Если да, то результаты просто записываются в выходной регистр . Если предсказание ложно, то записи в выходной регистр не происходит.