

# Архитектура ЭВМ

## План занятий:

1. Основы цифровой логики
2. Архитектура машинных команд
3. Архитектура оперативной памяти
  - a. Форматы хранения данных
  - b. Физическая организация данных
  - c. Протоколы обмена
4. Организация кэша
5. Внутренние шины
6. Организация вычислительных систем
7. MSR

## Книги:

1. Таненбаум
2. Цилькер, Орлов. Организация эвм и систем
3. Гук — «Аппаратные средства IBM PC»
4. Касперски — Техника оптимизации программ

## Домашнее задание

- **MDR** — регистр, хранящий данные, которые должны быть записаны в память или прочитаны из таковой по этому адресу.
- **MSR** (*Model Specific Register*)— специальные регистры процессоров архитектуры x86, наличие и назначение которых варьируется от модели к модели процессора. Программно доступны при помощи команд RDMSR и WRMSR. Адресуются 32-битным индексом, который помещается в регистр ECX.
- **ENIAC** (*Electronic Numerical Integrator and Computer* — Электронный числовой интегратор и вычислитель) — первый электронный цифровой вычислитель общего назначения, который можно было перепрограммировать для решения широкого спектра задач.
- **EDVAC** (*Electronic Discrete Variable Automatic Computer*) — одна из первых [электронных вычислительных машин](#). В отличие от своего предшественника ENIAC, это был компьютер на двоичной, а не десятичной основе. Как и ENIAC, EDVAC был разработан в Институте Мура Пенсильванского Университета для Лаборатории баллистических исследований. Армии США командой инженеров и учёных во главе с Джоном Преспером Экертом и Джоном Уильямом Мокли при активной помощи математика [фон Неймана](#) и [Германа Голдстайна](#).
- **MARK I** — первый американский программируемый компьютер. Разработан и построен в 1941 году по субподрядному договору с IBM группой из пяти инженеров-разработчиков под руководством [капитана 2-го ранга ВМФ США Говарда Эйкена](#) на основе более ранних наработок британского учёного

[Чарльза Бэббиджа](#). Программа исследований и создания машины финансировалась ВМФ США — заказчиком работ, генподрядчиком выступала компания IBM, после завершения работ по отладке вычислитель был передан в распоряжение флота и использовался им на завершающем этапе Второй мировой войны.

- **UNIVAC I** — первый условно коммерческий компьютер, созданный в США, и третий коммерческий компьютер в мире (после германского Z4 и британского Ferranti Mark 1). Спроектирован, в основном, [Джоном Экертом](#) и [Джоном Мокли](#), изобретателями компьютера ENIAC, на средства из федерального бюджета по заказу Армии и Военно-воздушных сил США.
- **EDSAC** — электронно-вычислительная машина, созданная Норрисом Уилксом. Первый в мире действующий и практически используемый компьютер с хранимой в памяти программой. Архитектура наследована от EDVAC.

## Базовые понятия

Разбор аббревиатура ЭВМ:

- **Э** — означает электронные элементы
- **В** — обработка информации

**ЭВМ** — совокупность программного обеспечения и программных средств, обеспечивающих автоматическую подготовку и решение пользовательских задач по обработке информации. Делятся на 2 типа: *цифровые, аналоговые*.

**Аналоговые** — непрерывные процессы.

**Цифровые** — для моделирования информации дискретной величины.

**Архитектура (в узком смысле)** — логическая организация или программно доступная модель. Узнаёт все ресурсы машины программными средствами.

Компоненты архитектуры ISA:

- Архитектура памяти
- Типы данных
- Набор регистров
- **IRQ** - система ввода-вывода
- **ISA** - архитектура

**Организация ВМ** — это совокупность сведений о её физическом устройстве, технологических характеристиках, технических особенностей.

**Архитектура (в широком смысле)** — совокупность логической модели и организационной модели виртуальной машины, рассматриваемая с точки зрения конкретного пользователя с конкретными целями на конкретном уровне детализации.

## Классическая архитектура ВМ

Включает в себя три уровня, строится на основе субъектно-ориентированного принципа.

1. **Внешний:** Субъект — пользователь. При этом сама модель выступает чёрным ящиком.
2. **Концептуальный:** Субъект — прикладной программист. Модель включает основные компоненты и их взаимодействие
3. **Внутренний:** Субъект — системный программист. Модель включает внутреннее устройство основных компонент ВМ.

## Принципы фон Неймана

- **Binary Coding:** вся информация кодируется набором 0 и 1. На самом деле, принадлежит Чарльзу Бэббиджу.
- **Линейность и однородность памяти:** команды и данные неразличимы между собой (RAM).
- **Программного управления:** вычислительная машина обрабатывает информацию автоматически под управлением программы, хранящейся в памяти (принцип хранения программы).
- **Неизменности цикла обработки команд:** исполнение конкретной программы реализуется последовательной обработкой каждой команды в режиме интерпретатора однотипным образом.

Следствия:

- Линейность и однородность
  - Можно написать самомодифицирующуюся программу
  - Идея компиляции

Этапы обработки:

1. FETCH: Извлечение из RAM в Регистр Команд
2. Вычисление адреса следующей команды
3. EXECUTE
  - Декодирование команды
  - Подготовка операндов
  - Выполнение команды
  - Запись результата

Аббревиатуры:

- IP — Instruction Pointer
- PC — Program Counter (счётчик команд)
- PK — лишняя
- PAK — Регистр Адреса Команд
- СК — Счётчик Команд

## Домашнее задание

**Транзисторно-транзисторная логика (ТТЛ, TTL)** — разновидность цифровых логических **микросхем**, построенных на основе **биполярных транзисторов** и резисторов. Название *транзисторно-транзисторный* возникло из-за того, что транзисторы используются как для выполнения логических функций (например, И, ИЛИ), так и для

усиления выходного сигнала (в отличие от [резисторно-транзисторной](#) и [диодно-транзисторной логики](#)).

**КМОП** (комплементарная структура металл-оксид-полупроводник; [англ. CMOS, complementary metal-oxide-semiconductor](#)) — технология построения электронных схем. В более общем случае — КМДП (со структурой металл-диэлектрик-полупроводник). В технологии КМОП используются [полевые транзисторы](#) с изолированным затвором с каналами разной проводимости. Отличительной особенностью схем КМОП по сравнению с биполярными технологиями ([ТТЛ](#), [ЭСЛ](#) и др.) является очень малое энергопотребление в статическом режиме.

Программа в фон-неймановской ЭВМ реализуется центральным процессором (ЦП) посредством последовательного исполнения образующих эту программу команд. Действия, требуемые для выборки (извлечения из основной памяти) и выполнения команды, называют циклом команды. В общем случае цикл команды включает в себя несколько составляющих (этапов):

- выборку команды;
- формирование адреса следующей команды;
- декодирование команды;
- вычисление адресов операндов;
- выборку операндов;

Цикл команды 139

- исполнение операции;
- запись результата.

Перечисленные этапы выполнения команды в дальнейшем будем называть стандартным циклом команды. Отметим, что не все из этапов присутствуют при выполнении любой команды (зависит от типа команды), тем не менее этапы выборки, декодирования, формирования адреса следующей команды и исполнения имеют место всегда.

В определенных ситуациях возможны еще два этапа:

- косвенная адресация;
- реакция на прерывание.

## Архитектурная модель Таненбаума

**Функционально-семантический принцип** — вычислительная машина трактуется как многоуровневая иерархия виртуальных машин. Имеет собственную программную среду и конкретную семантику.

Уровни:

1. **Прикладной** (Языки Высокого Уровня)
2. **Ассемблера**: Виртуальная Машина — нечто, что понимает ассемблерные программы
3. **Операционной системы**: Виртуальная Машина — программный комплекс, у которого есть управляющая система

4. **Архитектура команд:** основные программные инструменты — машинные команды (RISC, CISC, VLIW, ???)
5. **Микроархитектурный уровень:**
  - a. Конвеер
  - b. Организация многоядерности
  - c. Организация системы кэша
6. **Цифровой логический уровень:** логика работы аппаратного уровня
7. **Физический**

Примеры микроархитектур:

- NETBURST
- IVY BRIDGE

## Основы цифровой логики





Любое устройство аппаратной части можно рассматривать как большую сложную цифровую схему, при этом мы, разумеется, рассматриваем только логику работы устройства.

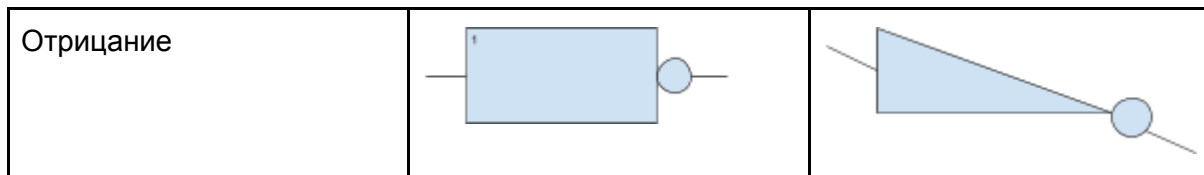
**Основным элементом** любой цифровой схемы является цифровой логический элемент (GATE или вентиль).

**Вентиль** — относится к элементам потенциального типа, характерными чертами которого являются гальваническая связь между входом и выходом и возможность построения схемы без применения реактивных элементов или с использованием ограниченного числа конденсаторов малой ёмкости для вспомогательных целей.

**Вентиль (в узком смысле)** — это цифровая схема, реализующая базовую логическую операцию (булеву функцию).

**Вентиль (в широком смысле)** — цифровая схема, реализующая произвольную логическую функцию.

	IEC	ANSI
Дизъюнкция		
Конъюнкция		



При конструировании интегральных логических схем схема техники в качестве базы использует NOR'ы (отрицание OR, НЕ-ИЛИ), NAND'ы (отрицание AND).

На схемах для удобства совокупность бинарных элементов могут заменять одним соответствующим многовходовым элементом (много стрелочек в задугу).

**Утв.** Любую логическую функцию можно задать двумя принципиально разными способами:

- Таблица истинности
- Подходящее алгебраическое выражение

Количество функций от  $k$  переменных —  $2^{2^k}$  (количество различных строк задаёт различные входные данные, и для каждой строки функция может иметь или 0, или 1).

**Теорема:** Любую логическую функцию можно перевести к дизъюнктивной нормальной форме.

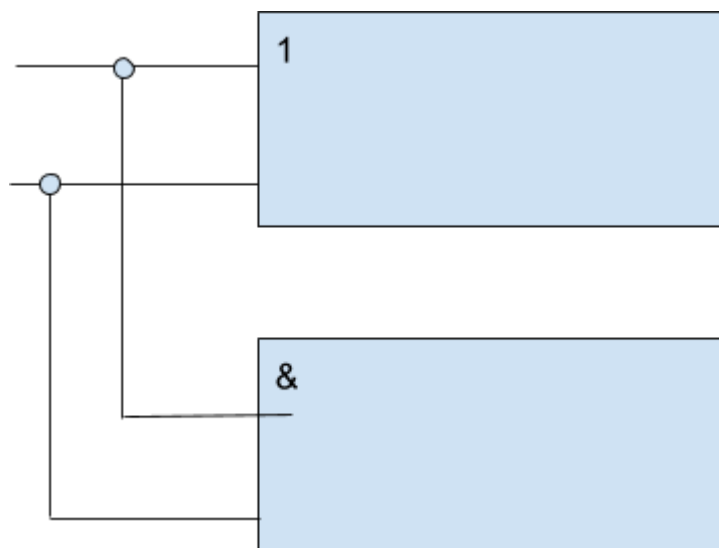
**Пример:**

$$f(a) \equiv af(1) + \bar{a}f(0)$$

$$f(a, b) = af(1, b) + \bar{a}f(0, b) = a(bf_{11} + \bar{b}f_{10}) + \bar{a}(bf_{01} + \bar{b}f_{00})$$

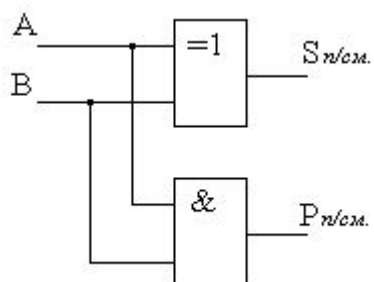
$A$	$B$	$xor$
0	0	0
0	1	1
1	0	1
1	1	0

$$A \text{ xor } B = \bar{A}B + A\bar{B} = (A + B)\overline{AB}$$

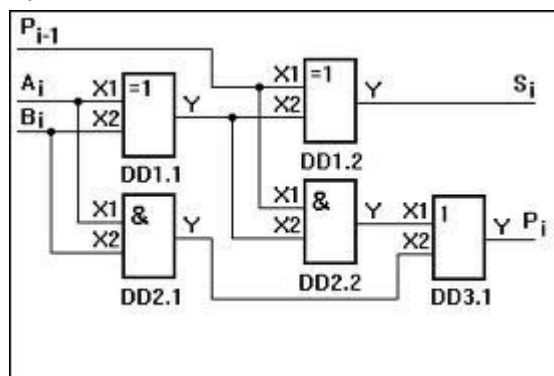


### Домашнее Задание

Полусумматор:



Сумматор:



$$p = ab$$

$$s = a \text{ xor } b$$

### NOR

- **AND:**  $(A \text{ nor } A) \text{ nor } (B \text{ nor } B)$
- **OR:**  $(A \text{ nor } B) \text{ nor } (A \text{ nor } B)$
- **NOT:**  $A \text{ nor } A$

## NAND

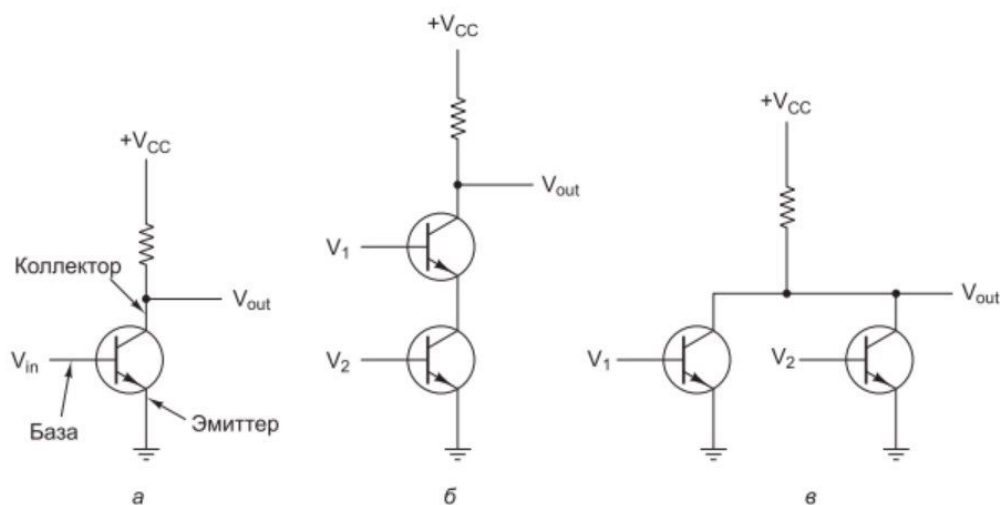
- **AND:**  $(A \text{ nand } B) \text{ nand } (A \text{ nand } B)$
- **OR:**  $(A \text{ nand } A) \text{ nand } (B \text{ nand } B)$
- **NOT:**  $A \text{ nand } A$

**VLIW (Very Long Instruction Word)** — Архитектура команд, которая рассчитана на параллелизм. Каждая команда хранит информацию о том, что должен делать каждый элемент процессора.

**Замечание 1:** Многоразрядный сумматор строится на одном из двух принципов:

- Поразрядно-последовательный —  $n$  1-разрядных сумматоров
- Блочно-параллельный — 2 параллельно работающих поразрядно-последовательных сумматора,

**Замечание 2:** Для реализации инвертора достаточно одного биполярного транзистора. Соответственно операции *nor* и *nand* можно реализовать через него.

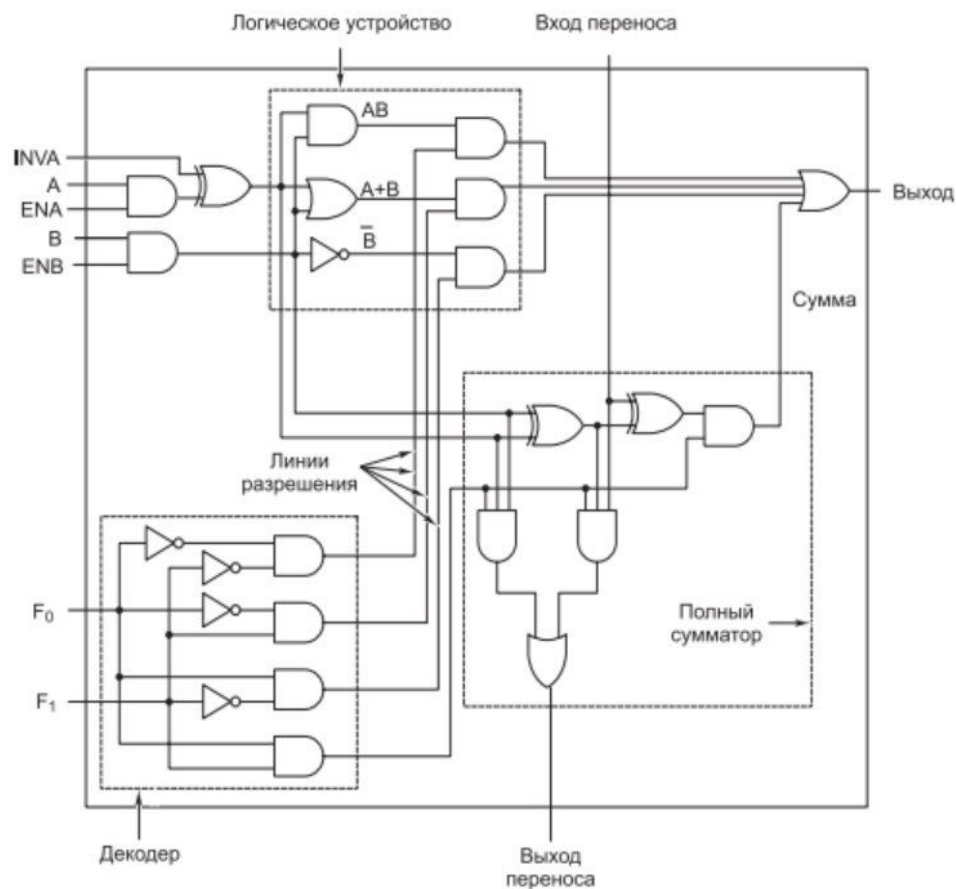


**Рис. 3.1.** Транзисторный инвертор (а); вентиль НЕ И (б); вентиль НЕ ИЛИ (в)

**Дополнение 1:** Все цифровые схемы можно условно поделить на 2 больших группы:

1. Управляющие элементы и вычислительные блоки: некоторое устройство, имеющее конечное число входов и выходов. Значение на выходе зависит от входа — реализуется логика управления или конкретная вычислительная операция. Представители:
  - а. Мультиплексоры и демультиплексоры.
  - б. Шифраторы и дешифраторы
  - с. Компараторы.
  - д. Сдвигатели.
  - е. Сумматоры.
  - ф. АЛУ(Арифметико-логические устройства).
2. Элементы памяти. Представители:





**Рис. 3.17.** Одноразрядное АЛУ

## Бистабильный элемент

**Дополнение 2:** Все цифровые схемы делятся на два класса:

- Композиционные
- Последовательные (накопительные, секвенциальные)

Значение выходных сигналов может также зависеть от некоторой накапливаемой в устройстве информации.

Есть композиционный блок и блок памяти.

**Дополнение 3:** Все процессы, протекающие внутри вычислительной машины можно разделить на два класса::

- ## 1. Синхронные

2. Асинхронные  
DDR, DDR2, DDR3 — отличия.

**Дополнение 4:** При конструировании отдельных схем наряду с позитивной логикой используется инверсная логика. В основном, в шинных интерфейсах и в управляющих системах контроллеров.

- Позитивная логика:
  - Есть сигнал — 1
  - Нет сигнала — 0
- Инверсная логика:
  - Нет сигнала — 1
  - Есть сигнал — 0

#### Позитивная логика

$A \setminus B$	$0B$	$5B$
$0B$	$0B$	$5B$
$5B$	$5B$	$5B$

#### Инверсная логика

$A \setminus B$	1	0
1	1	0
0	0	0

ДЗ: IEEE-754

## Стандарт представления данных в RAM.

### Числовые данные

- Числовые данные. В рамках данного формата, машина обрабатывает данные из ограниченного диапазона.
- Представление целых чисел:
  - Знаковые:
    - Явнознаковое — есть бит с информацией о знаке. Есть два нуля.
    - Biased — обрабатываем отрицательные числа, но не работаем с ними (работаем с соответствующими им, то есть с их смещениями). Смещение на половину размера числа.
    - Дополнительный код.
  - Беззнаковые
  - Ширина поля

#### Свойства

- Ограниченность диапазона
- В общем случае результатом арифметической операции может являться недоступное число (выход за пределы допустимого множества).
- Базовые законы арифметики не работают:
  - Дистрибутивность
  - Ассоциативность

## Вещественные числа

### Форматы хранения:

- Вещественное число с 10 точкой
  - Точка фиксируется справа от самой правой — целые числа, домноженные на константу.
  - Слева от самой левой —  $0 \leq |x| < 1$ .
  - В середине.

Представление числа в памяти может не совпадать с числом.

Первый стандарт — IEEE-754 1985. IEEE-854.1987.

$$X = M \cdot 10^P$$

При рассмотрении нормализованных двоичных чисел, хранящихся в памяти вычислительной машины, используется другое условие для мантииссы (IEEE 754).

**Замечание.** Большинство современных процессоров целую часть мантииссы не хранит (принцип мнимой единицы).

**Дополнение к замечанию:** Многие современные процессоры, явно декларирующие соответствие стандартов IEEE-754, на самом деле, в вычислениях его не применяют.

При большом количестве вычислений с большими числами может накопиться большая погрешность.

1. Правило преобразования
2. Стандартизация вычислений (Exception)

## Домашнее задание:

IEEE-754 — стандарт, описывающий формат хранения чисел с плавающей точкой. Используется в программных и аппаратных реализациях арифметических действий.

IEEE-754 описывает формат чисел с плавающей точкой: мантиисса, знак числа и показатель степени.

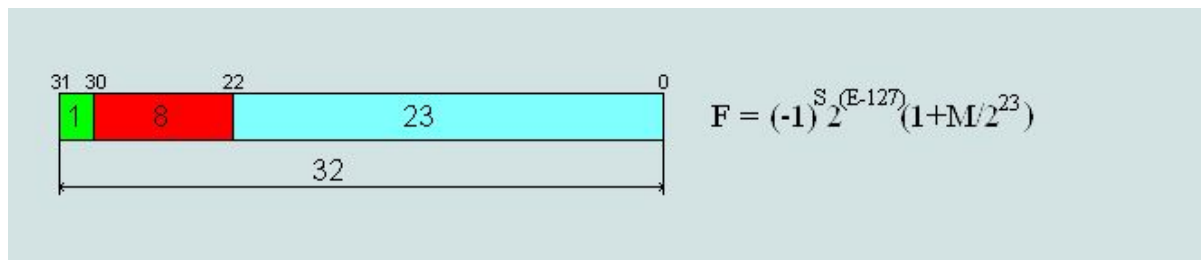
Представление положительного и отрицательного нуля, положительной, отрицательной бесконечностей, а также NaN'ов.

Структура IEEE-754

Размер числа составляет 4 байта.

Поля:

- Один бит — знак
- 8 бит — экспонента со смещением (от -126 до 127)
- 23 бита — дробная часть мантиисы (целая часть всегда единица)



Значения:

- Денормализованные числа: Экспонента — 8 нулей, целая часть равна 0
- Плюс и минус бесконечности: Экспонента — 8 единиц, мантииса равна 0
- NaN: Экспонента — 8 единиц, мантииса ненулевая.

## Характеристические свойства машинного вещественного числа.

1. Диапазон таких чисел всегда ограничен, что является следствием ограниченности поля, отводимого под хранение порядка.
  - a. Порядок хранится в смещённом виде.
  - b. Практически всегда крайние значения этого поля не используются для представления нормализованных чисел.
2. В общем случае в памяти хранится не само значение, а некоторое его приближённое значение, тем самым генерируется погрешность представления, что является следствием ограниченности поля для хранения мантиисы.
3. Множество машинных нулей не пусто, при чём они есть как относительные, так и абсолютные.
4. Машинные вещественные числа заполняют вещественную ось крайне неравномерно.

Домашнее задание:

- Пункт 3: BCD — двоичные десятичные цифры
- Примеры первых трёх символьных кодировок (5-7 бит)
  - Baudot code (5 бит). Создана в 1870, запатентована в 1874.
  - Fieldata (6-7 бит). В 1950-х.
  - IBM's Binary Coded Decimal (BCD) (6 бит). В 1959 году.

## Двоично-десятичные числа (BCD)

Каждая десятичная цифра заменяется полубайтом.

Форматы BCD:

- **Зонный:** цифра занимает младшую часть (4 бита). Зоны — старшую (24 бита).

- **Плотный (упакованный):** в один байт записывается информация о 2 цифрах. Плюс знак на поле бита.
- **FPU (в x86)** — Floating Point Unit. 80-битные регистры, а также есть поддержка 18-разрядных десятичных чисел.
- **Упакованные числа:** для реализации подхода SIMD — один операнд обрабатывает большое количество наборов аргументов.

#### 1. Pentium MMX.

- а. Тип данных — 64-bit Integer.
- б. Упакованные данные
- в. Упакованные слова
- г. Упакованные двойные слова

#### 2. MMX Регистры

- 1) D... Type
- 2) Command-57
- 3) XMM<sub>0</sub> - XMM7
- 4) 3DNow!

Домашнее задание:

- Описать последнее на данный момент SSE (не SSE2, SSE3).

Для выхода из ситуации организовали консорциум, разработавший и предложивший стандарт Юникода. В нем предполагалось объединить знаки всех языков мира в одной большой таблице. Кроме того, определялись кодировки. Сначала ребята посчитали, что 65 535 посадочных мест должно хватить всем, ввели UCS-2 — кодировку с фиксированной 16-битной длиной кодов. Но пришли азиаты с многотомными азбуками, и расчеты рухнули. Кодовую область увеличили вдвое, UCS-2 уже не смогла бы справиться, появилась 32-битная UCS-4. Ощутимыми преимуществами кодировок UCS являлись постоянная кратная двум длина кодов и простейший алгоритм кодирования, и то, и другое способствовало скорости обработки текста компьютером. Но при этом была и неоправданная, чересчур расточительная трата места: представьте, что в ASCII 00010101, то в UCS-2 00000000 00010101, а UCS-4 уже 00000000 00000000 00000000 00010101. С этим нужно было что-то делать.

Развитие Юникода повернуло в сторону кодировок с переменной длиной получаемых кодов. Представителями стали UTF-8, UTF-16 и UTF-32, последняя условно-досрочно, так как на данный момент она идентична UCS-4. Каждый символ в UTF-8 занимает от 8 до 32 бит, причем есть совместимость с ASCII. В UTF-16 16 или 32 бита, UTF-32 — 32 бита (если бы пространство Юникода расширили еще вдвое, то уже 32 или 64 бита), с ASCII эти две не дружат. Количество занимаемых байтов, зависит от позиции символа в таблице Юникода. Очевидно, наиболее практичная кодировка — UTF-8. Именно благодаря своей совместимости с ASCII, небольшой прожорливости до памяти и достаточно простым правилам кодирования, она является наиболее распространенной и перспективной кодировкой Юникода. Ну, а в завершение красивая схема преобразования кода символа в UTF-8:

1	2	3	4	5	6	$\Sigma$
0.8	0.5	$1.1 + 0.6 = 1.7$	1	0.7	0.4	5.1

**ISO-646** — 5-битный стандарт кодирования символов.

**BCDIC** — двоично-десятичный декодер информации. Возникла из перфокарт, была 6-тибитная.

**FELDATA** — набор 6- и 7-битных протоколов для обработки и передачи данных, позже один из них был использован в UNIVAC-1100.

**EBCDIC** - тот же BCDIC, только 8-битный

**ASCII** — 7-битный код. Стандарт 128 символов. 0x1B - код клавиши Esc.

**Y2K**

**USASCII**

Контроль передачи данных был произведён при помощи последнего бита.

Чтобы сделать единообразную кодировку для всех символов, пытаются расширить диапазон количества битов на хранение символа

CJK - China, Japan, Korea. Азиатские кодировки.

CP-437

CP-866

CP-1251

UNICODE - четырёхуровневая иерархия (U+XXXX).

UNICODE 2.0 - расширил возможное множество различных чисел  $= 2^{20} + 2^{16} - 2^{11}$

(U+XXXX(X)(X)).

10FFFF - число, с которого начинаются 6-значные.

UNICODE можно интерпретировать, как четырёхслойную структуру.

1. Абстрактное множество символов
2. Множество закодированных символов
3. Формы кодировки символа
4. Схемы кодировки символа

Code Unit - количество битов на символ.

Code Point - число в соответствие символу.

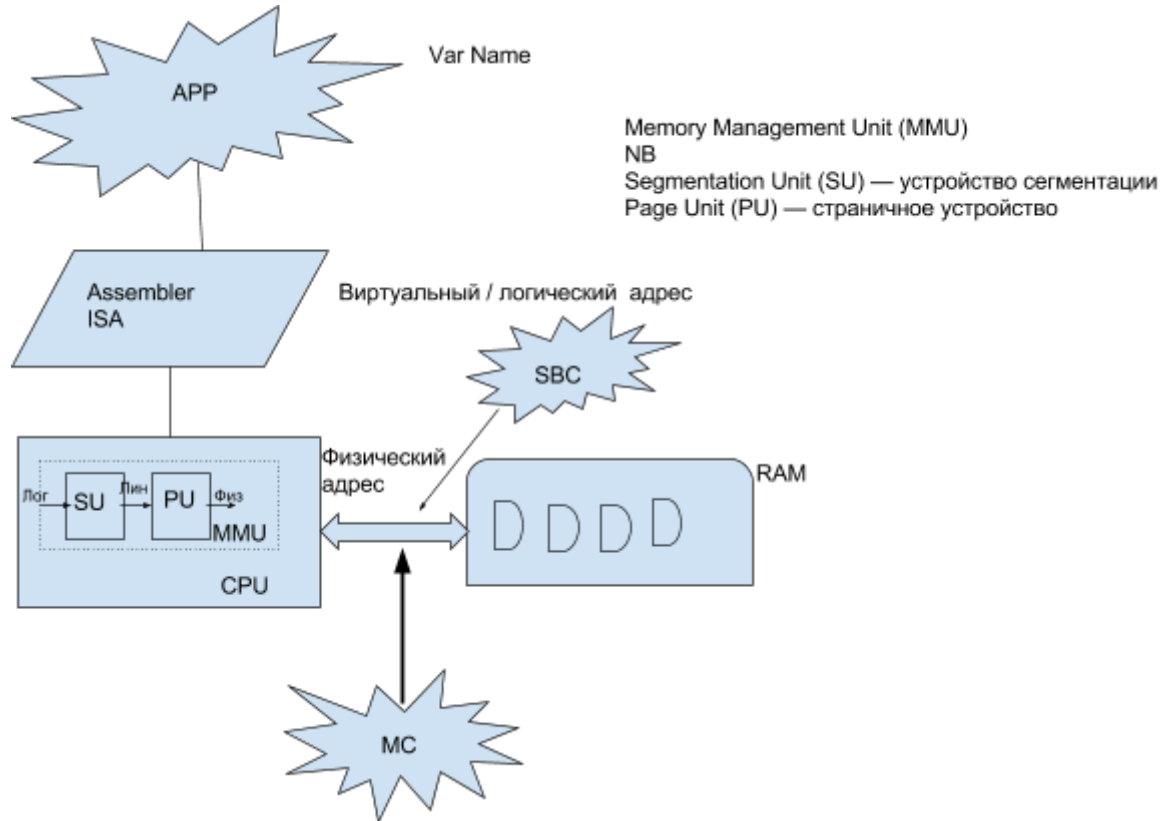
UTF - уровень code unit'ов (8/16/32).

UTF-32. Под code point отводится 32-битное поле.

**Многоязыковая плоскость** — набор символов unicode с зафиксированными значениями байтов, кроме последних двух.

**BOM** — маркер последовательности байтов, один из символов unicode. Указывает порядок байтов и кодировку. Необязателен, ставится в начале файла.

## Архитектура RAM



**MMU (Memory Managment Unit)** — расположен в процессоре.

Функция: получает логический адрес и возвращает соответствующий ему физический.

В состав входит:

- **Устройство сегментации (SU).** Логический адрес линейный адрес.
- **Страничное устройство (PU).** Линейный адрес физический адрес.

**Замечание.** В ситуациях, когда механизм страничного преобразования отключен, то логический адрес совпадает с физическим

**SBC (System Bus Contolter)** — контроллер системной шины. Между процессором и памятью.

Шина состоит из 3 семантических компонент:

1. Шина данных
2. Шина адреса
  - a. Памяти
  - b. Ввода / вывода
3. Шина управления

**Замечание 1:** Шина управления кроме непосредственно управляющих сигналов содержит подсистему индикации состояния.

**Замечание 2:** Не совсем правильно включать в шину управления сигналы питания и тактовый импульс.

### Физическое устройство шины:

1. Провода
2. Разъёмы
3. Протокол: набор соглашения о том, как элементы шины обмениваются информацией между собой.
4. Контроллер
5. Спецификация шины: технологические характеристики.

## Параграф 1. Логические модели адресации.

С точки зрения схем преобразования адресов и механизмов управления этим процессом можно выделить 3 модели памяти:

1. FLAT
2. SEGMENT
3. PAGE

**Опр:** Плоская: всё адресное пространство — линейно упорядоченный набор базовый ячеек.

**Замечание 1:** в настоящее время в чистом виде встречается редко, но достаточно часто используется смешанная модель — сегментно-плоская.

**Замечание 2:** Большинство современных 64-разрядных устройств реально использует для адресации не всю шину адреса, а только первые  $k$  бит.

**Опр.** Сегментная: возникла при реализации многозадачных операционных систем. Каждому процессу выдаётся своё адресное пространство, изолированное от других процессов.

Сегмент — ненулевой фрагмент памяти, выделенный конкретному процессу или задаче для эксклюзивного использования. Каждый адрес имеет 2 части:

1. Сегментация, определяющая базовый адрес сегмента в линейном пространстве.
2. Смещение, показывающее смещение от начала смещения.

Как правило, адрес записывается в формате *сегмент:смещение*.

Характеристики сегмента:

1. Содержание
  - a. Данные
  - b. Код
  - c. Участок стека
2. Размер
  - a. Предопределённый
  - b. Динамический
3. Атрибуты
  - a. Права доступа к сегменту

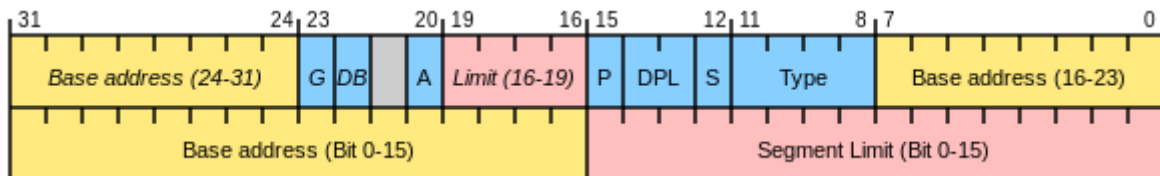
Правила формирования адреса:



1. Сегментная часть адреса может задавать базовый адрес в явном виде или опосредственно.
  - a. В первом случае достаточно часто на базу сегмента накладывается дополнительное ограничение в виде выравнивания на границу. Чтобы можно было отбрасывать нули.
  - b. Опосредованное задание, как правило, реализуется с помощью механизма селекторов и дескрипторов. То есть, для каждого сегмента создаётся специальная системная структура — дескриптор. Содержит информацию о конкретном сегменте:
    - i. Базовый адрес (base)
    - ii. Верхний предел (limit) — информация о максимально возможно размере
    - iii. Права доступа к данному сегменту (AR + States: Access right).

ДЗ:

- Структура дескриптора x86
- Разница между дескриптором данных и стека



**Базовый адрес** — указатель на начало сегмента в линейной памяти.

**Верхний предел** — размер сегмента (в байтах).

**Синие поля** — флаги прав доступа к сегменту.

Все адреса сегмента выравниваются по границе параграфа.

В мультизадачных средах, построение сегментного адреса базируется на двух объектах - дескрипторе и селекторе. Дескриптор - описание отдельного сегмента. Селектор - указатель на конкретный дескриптор.

Дескрипторы объединяет следующая структура:

- 1) База
- 2) Размер сегмента (База + 1). Максимальное значение сегмента (но к сегменту стека относится по-другому)
- 3) Права доступа и, возможно, текущее состояние

Для оптимизации управления сегментами, дескрипторы объединяются в системные структуры - таблица дескрипторов (Глобальная таблица дескрипторов (GDT) и локальные таблицы дескрипторов (LDT)).

LDT строится для каждой задачи отдельно. LDT по каждой задаче изолирована от других. GDT доступна отовсюду. Для хранения информации для текущей LDT используются специальные системные регистры. Один для GDT. LDT загружается при старте каждого нового процесса.

В некоторых архитектурах сам селектор содержит однозначные указания на глобальность или локальность.

Селекторы хранятся в селекторных регистрах. Селектор показывает как сместится по дескриптору, там нашли базовый адрес и вытащили его.

Почти во всех архитектурах сегментные регистры имеют связанные с ними теневые регистры, которые в общем случае хранят либо полный дескриптор сегмента, либо, как минимум, его базу и значение предела.

Теневые регистры управляются на уровне операционной системы. Значение теневого регистра соответствует состоянию исполняемой задачи.

С помощью теневых регистров ускоряется вычисление линейного адреса дескриптора. За счёт теневых регистров не нужно ходить по таблице дескрипторов. При загрузке нового регистра, теневые загружаются автоматически.

## Работа с сегментной моделью в IA32

Схема задеиствуется, когда работает в защищённом режиме процессора (появился в 286, но полноценная реализация в 386 - первый 32-разрядный процессор Intel).

Переключение из обычного в защищённый режим производилось с CR0. В 386 переключение было программное, а в 286 перезагрузкой.

Системные регистры таблицы дескрипторов:

- LDTR локальный адрес
- GDTR глобальный адрес
- IDTR адрес таблицы прерываний (все таблицы прерываний - глобальные)

TSS - структурированная область памяти, которая позволяет производить мультизадачность. Сохраняем состояние в TSS, затем LDTR и GDTR загружаются новыми адресами.

Откуда взялось 64 терабайта виртуальной памяти в архитектуре IA32.

В разных режимах работы процессора, эти характеристики (размер сегмента) могут быть разными

- Режим реальной адресации (64к)
- Защищённый режим ( $2^{20}$  размер сегмента. 8Гб). **Бит гранулярности** показывает в каких единицах измеряется сегмент.

Значение смещения может определяться весьма разнообразно:

- Константа в команде
- Имя регистра, в котором хранится соответствующая константа

- Арифметическое выражение, действующее несколько регистров. Пример к правилу: В архитектуре IA32, в общем случае, для формирования смещения могут быть задействованы явная константа, два регистра и коэффициент умножения. Результат называет эффективным или исполнительным адресом

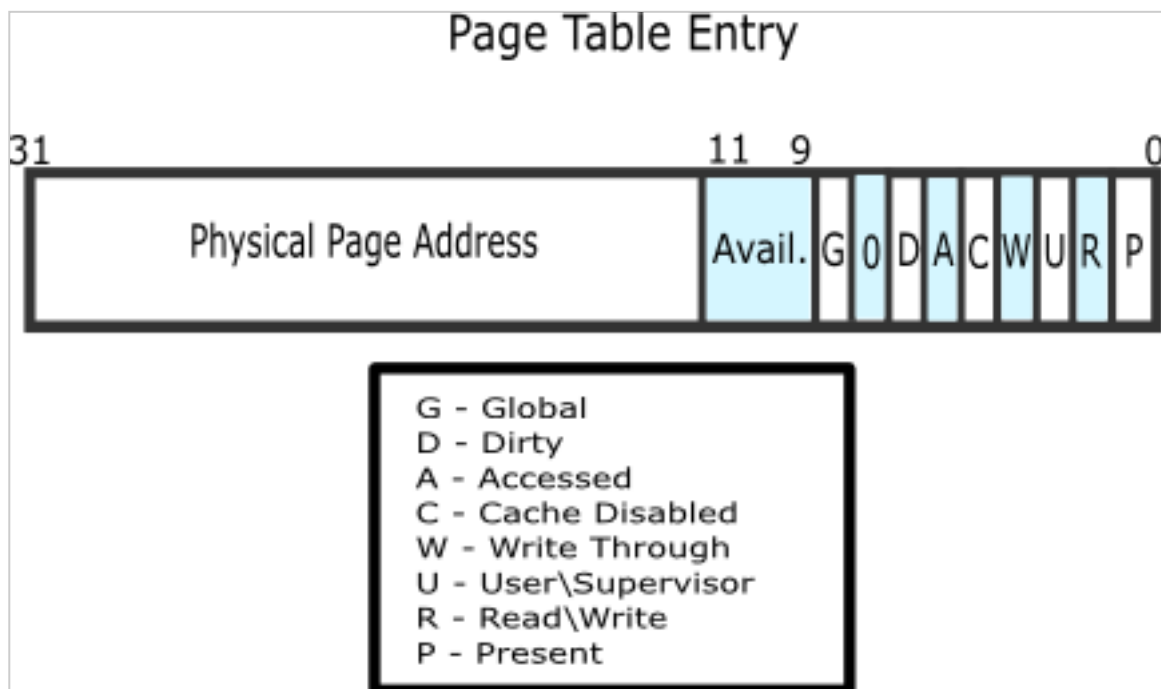
referenced by the page directory entry. Therefore if you wish to make a page a user page, you must set the user bit in the relevant page directory entry as well as the page table entry.

- R, the 'Read/Write' permissions flag. If the bit is set, the page is read/write. Otherwise when it is not set, the page is read-only. The WP bit in CR0 determines if this is only applied to userland, always giving the kernel write access (the default) or both userland and the kernel (see Intel Manuals 3A 2-20).
- P, or 'Present'. If the bit is set, the page is actually in physical memory at the moment. For example, when a page is swapped out, it is not in physical memory and therefore not 'Present'. If a page is called, but not present, a page fault will occur, and the OS should handle it. (See below.)

The remaining bits 9 through 11 are not used by the processor, and are free for the OS to store some of its own accounting information. In addition, when P is not set, the processor ignores the rest of the entry and you can use all remaining 31 bits for extra information, like recording where the page has ended up in swap space.

Setting the S bit makes the page directory entry point directly to a 4-MiB page. There is no paging table involved in the address translation. **Note: With 4-MiB pages, bits 21 through 12 are reserved!** Thus, the physical address must also be 4-MiB-aligned.

## Page Table



A Page Table Entry

In each page table, as it is, there are also 1024 entries. These are called page table entries, and are **very** similar to page directory entries.

*Note: Only explanations of the bits unique to the page table are below.*

The first item, is once again, a 4-KiB aligned physical address. Unlike previously, however, the address is not that of a page table, but instead a 4 KiB block of physical memory that is then mapped to that location in the page table and directory.

The Global, or 'G' above, flag, if set, prevents the TLB from updating the address in its cache if CR3 is reset. Note, that the page global enable bit in CR4 must be set to enable this feature.

If the Dirty flag ('D') is set, then the page has been written to. This flag is not updated by the CPU, and once set will not unset itself.

The 'C' bit is 'D' bit above.

Документация:

[http://developer.amd.com/wordpress/media/2012/10/24593\\_APM\\_v21.pdf](http://developer.amd.com/wordpress/media/2012/10/24593_APM_v21.pdf)

## Механизмы защиты сегментной модели

Механизмы защиты сегментной модели основываются на 2 классах объектов:

1. Дескрипторы сегментов
2. Уровень привилегий

Иерархия привилегий затрагивает 2 набора объектов:

1. Машинные команды
2. Дескрипторы сегментов и дескрипторы сегментных объектов

Для выполнения команды анализируется легитимность данной операции и выполнение определённых ограничений для операндов.

В архитектуре IA-32 можно выделить 3 набора ограничений:

1. Ограничения дескрипторов данных
  - a. Существование сегмента
  - b. Наличие дескриптора в памяти
  - c. Допустимость соответствующего обращения
2. Ограничение по операндам
  - a. Выход за границы сегмента
3. Ограничение по командам перехода (не пытается ли программа залезть в чужой сегмент)
  - a. jmp
  - b. call
  - c. int
  - d. return

Свопинг плохо работает для сегментов из-за произвольного размера (а у страниц фиксированный).

## Страничная модель памяти

Основное назначение страничной модели

1. Возможность работы процесса с виртуальной памятью

Система привелегий PU (2) беднее, чем SU (4).

Страниц много, поэтому описатели страниц объединяются в таблицы. Описатели занимают одну страницу.

Механизм страничной трансляции строится на принципе зонирования адреса: линейный адрес разбивается на фиксированное количество зон, каждая зона участвует в формировании физического адреса.

Всего описателей  $2^{20}$  штук.

Практически во всех современных архитектурах страницы организованы в многоуровневые иерархические структуры, что позволяет держать в памяти таблицы только, которые используются исполняемыми в данный момент задачами.

**Пример (IA-32):**

Линейный адрес разбивается на 2 части: смещение — 12 бит. Старшая — ещё на 2 участка по 10 бит.

Старшая часть адреса даёт номер раздела. Для него строится каталог страниц, содержит описатели всех страниц, которые принадлежат данному разделу.

Средняя — нужную страницу.

Младшая — базовый адрес (и добавляем смещение).

Всё линейное пространство делится на разделы. Каждый раздел — на страницы.

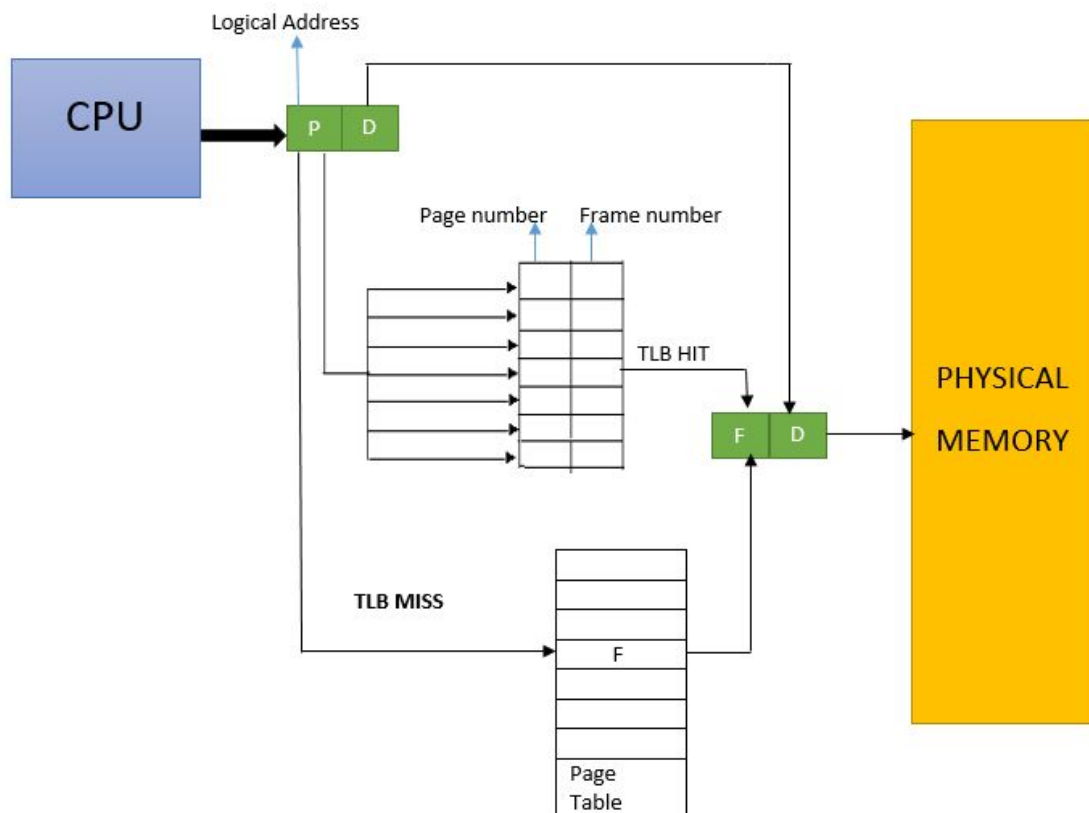
**ITANIUM** — единственный представитель IA-64.

**Правила:**

- Подавляющее большинство современных 64-разрядных ОС используют больше двух уровней иерархий для управления страницами.
- Многие архитектуры с течением времени меняют первоначальное ограничение на размеры страниц. Например, в IA-32, начиная с процессора Pentium, появилась возможность обрабатывать страницы размером 4 MB. В P6 появилась возможность использовать страницы 2MB.
- Многие архитектуры используют встроенный в MMU специализированный кэш, ускоряющий механизм страничной трансляции: в нём хранится информация о недавно использованных страницах (старшая часть линейного адреса, базовый физический адрес). Как правило, эти кэши строятся по принципу ассоциативной памяти. Первая реализация в IA-32 появилась в Intel-80386. Назывался TLB.

ДЗ: Структура TLB.

[http://intel80386.com/386htm/s10\\_06.htm](http://intel80386.com/386htm/s10_06.htm)



Практически во всех современных процессорах используется несколько разных структур TLB.

В современных процессорах реализована возможность использования страниц разных размеров, при чём она поддерживается разными схемами зонирования адреса.

Дополнения к посегментной защите в архитектуре IA-32.

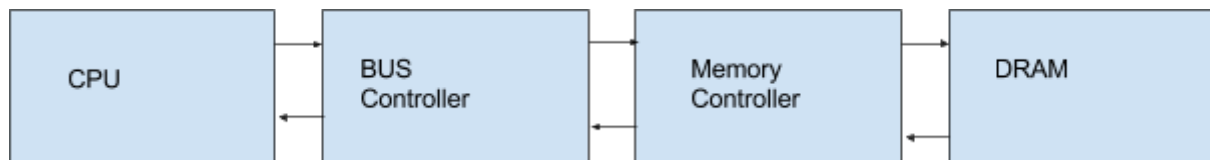
- **DPL** — **D**escriptor **P**rivilege **L**evel
- **CPL** — **C**urrent **P**rivilege **L**evel. Младшие 2 бита селектора кодового регистра.
- **RPL** — **R**equest **P**rivilege **L**evel. Уровень, на котором мы хотим работать с запрашиваемым сегментом.
- **EPL** — **E**ffective **P**rivilege **L**evel.  $EPL = \max(\{CPL, RPL\})$ .

## Физическая организация DRAM

1. В микросхемах динамической памяти для хранения одного бита информации используется конденсатор, снабжённый запирающим транзистором. Если на платах есть заряд, то хранится 1, иначе 0. По физике конденсатора всегда существуют токи утечки. Поэтому надо постоянно заряжать (регенерировать).
2. В процессе развития этих микросхем регенерацией управляли программисты, специально выделенные микросхемы (DMA 18.2), контроллер памяти, сами микросхемы.
3. Операция чтения является деструктивной. Чтобы избежать потери, микросхема снабжается специальным буфером.

4. Логически микросхема динамической памяти представляет из себя совокупность 2 систем:
  - a. Ядро памяти, реализованное в виде двумерной матрицы.
  - b. Интерфейсная обвязка (схемы, обеспечивающие взаимодействие ядра микросхемы с внешней шиной).
5. Считывание данных происходит сразу из одной целой строки.
6. Изначально информация записывается в чувствительный усилитель, а только потом идёт в буфер. Он усиливает те сигналы, которые к нему пошли.

### Схема организации передачи данных



Активизация RAS говорит, что на уровне строки, а не столбец.

Слова с картинки:

1. **RAS** — **R**ow **A**ccess **S**trobe
2. **CAS** — **C**olumn **A**ccess **S**trobe (**CAC** - **C**olumn **A**ccess **T**ime)
3. **RCD** — Время между доступом к RAS и к CAS.
4. **RP** — время цикла обращения.
5. **AC** — время цикла обращения к памяти.

Формула памяти - конструкция из 4 цифр, разделённых дефисами (6-4-x-x, 6-3-x-x, 5-4-x-x, etc)

Первая цифра - время доступа к произвольной ячейки, с которой мы начинаем.

Не все циклы в машине пакетируются. У нас два адресных пространства.

Эффективная тактовая частота - это удвоенная частота такта для выполнения двух операций за один такт

Если мы обрабатываем последующие ячейки, то для каждой последующей мы не будем менять строку, по которой мы идём - в этом заключается оптимизация  
За счёт введения специального буфера появляется возможность начать подготовку раньше

Внутри микросхемы добавили двухбитовый счётчик

Строка кэша в 4 раза шире данного процессора.

Если по шине считается 32 бита, то из них возьмётся столько, сколько нужно по шине процессора. Но существуют разные режимы процессора, поэтому может случиться так, что мы считываем 2 байта из 32 битов, остальные 16 отбрасываем



CDRAM.

- Ширина шины данных стала 64 бита
- Счётчик в BEDO был расширен, чтобы можно было обрабатывать циклы произвольной длины.
- Многобанковые системы
- В обязательном порядке наличие SPD

## Кэш-память

Возникла при разработке системы ATLAS.

Между процессором и оперативной памятью создаётся дополнительная среда, которая, с одной стороны, работает намного быстрее оперативной памяти, а с другой стороны устроена таким образом, что мы не можем заменить всю оперативную память на кэш-память.

**Кэш-память** - это некоторая промежуточная память небольшого объема, работающая гораздо быстрее динамической, основанная на микросхемах статической памяти и являющаяся программно-недоступной (прозрачной для программистов).

Управляется центральным процессором и контроллером кэша.

Почему этой памяти мало? Дорогая. Почему она дорогая? Это статические микросхемы, а не динамические.

Что из себя представляет ячейка статической памяти? Она реализует логику работы триггера. А что такое триггер? Триггер - это бистабильный логический элемент. А что такое бистабильный? Значит у него есть два состояния, в каждом из которых он сохраняется бесконечно долго, пока есть электричество.

FSE - частота системной шины. Но это неправда. Это либо частота системной шины, либо частота системной шины, увеличенная в 2 или 4 раза. BSB.

Кэш первого уровня всегда "живёт" внутри процессора (на кристалле процессора). Кэш второго уровня находится "где-то поблизости", но не на кристалле. Например, на системной плате. Также, у него есть отдельная шина для связи с ядром процессора.

Intel вынудили сделать QPE.

# Характеристики кэша

## Архитектура

1. Линейная
2. Полностью ассоциативная
3. Множественно ассоциативная

## Политика записи

1. **Сквозная** — записывается и в кэш и в память.
2. **Обратная** — записывается в кэш, а память пишется только по истечению времени, или когда необходимо (переключение на другое устройство).

## Механизм замещения(в Direct Map нет)

1. Random
2. FIFO
3. Least Used
4. LRU (дольше всего не использовался), в большинстве процессоров используется не чистый LRU, а псевдо LRU.

## Механизмы когерентности

- MESI (Modify Exclusive)
- MOESI
- Когерентность уровней кэша
- Межядерная когерентность

## Семантическая однородность

- Harvard — разделение на кэш кода и кэш данных
- Princeton — однородный (совмещённый) кэш

## Связь уровней (как хранится информация на 1 и 2 уровнях)

- Inclusive - что хранится в 1 хранится и на 2
  - Exclusive - что хранится на 1 не хранится на 2
- Разница в механизме замещения.

## Блокируемый и неблокируемый

# Архитектура машинных команд (ISO)

Домашнее задание:

[http://ethw.org/Milestones:First\\_RISC\\_\(Reduced\\_Instruction-Set\\_Computing\)\\_Microprocessor\\_or\\_1980-1982](http://ethw.org/Milestones:First_RISC_(Reduced_Instruction-Set_Computing)_Microprocessor_or_1980-1982)

1.

**PATTERSON** — важный сторонник концепции микропроцессорных архитектур типа RISC и автор самого термина «RISC». С 1980 по 1984 годы он руководил проектом Berkeley RISC, в рамках которого были созданы процессоры RISC I и RISC II.

**HENNESSY** — The MIPS project, led by Prof. John Hennessey at Stanford, featured important attention to the role of the compiler in making best use of RISC processor resources. The first working chip resulting from that project came about a year after RISC-I at Berkeley.

**SEQUIN** — observed that compilers for high-level computer languages, such as C, rarely utilized the added instructions. They thought that overall performance could be improved by optimizing the combination of processor function and memory on a single chip. Better overall performance at a much lower cost might be achieved by simplifying the processor, thereby allowing more chip area to be devoted to memory. Thus the goal was defined as a RISC, or reduced instruction set computer.

**DITZEL** — The case for the reduced instruction set computer.

**J. COCKE (IBM-801)** — The IBM work begun in 1975, led by Dr. John Cocke, was aimed the control requirements for an electronic telephone central switch. That project influenced by Dr. Cocke's recognition that compilers of that era rarely made use of complex high-level language instructions.

**HARVARD** — Гарвардская архитектура, используется в RISC

**PRINCETON** — Принстонская архитектура, используется в CISC

Какие архитектуры машинных команд возникли, благодаря их усилиям.

2.

Архитектура MERCED

IA-64 (also called Intel Itanium architecture) is the instruction set architecture (ISA) of the Itanium family of 64-bit Intel microprocessors. The first Itanium processor, codenamed **Merced**, was released in 2001.

HP researchers investigated a new architecture, later named Explicitly Parallel Instruction Computing (EPIC), that allows the processor to execute multiple instructions in each clock cycle. EPIC implements a form of very long instruction word (VLIW) architecture, in which a single instruction word contains multiple instructions. With EPIC, the compiler determines in advance which instructions can be executed at the same time, so the microprocessor simply executes the instructions and does not need elaborate mechanisms to determine which instructions to execute in parallel.

By the time Itanium was released in June 2001, its performance was not superior to competing RISC and CISC processors.[21] Itanium competed at the low-end (primarily four-CPU and smaller systems) with servers based on x86 processors, and at the high end with IBM's POWER architecture and Sun Microsystems' SPARC architecture. Intel repositioned Itanium to focus on high-end business and HPC computing, attempting to

duplicate x86's successful "horizontal" market (i.e., single architecture, multiple systems vendors).

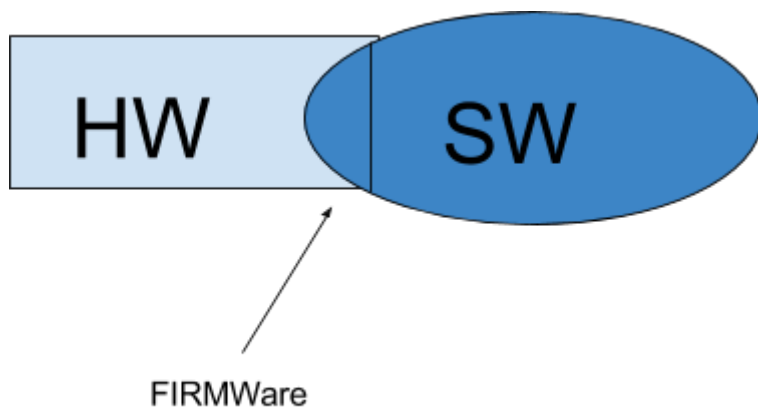
1	2	3	4	5	$\Sigma$
0.6	0.5	0.2	0.1	0.7	2.1

## ISA

1. Basic
2. Классификация
  - a. Историческая
  - b. Структурно-семантическая (сложность)
    - i. CISC
    - ii. Load / Store
      1. RISC
        - a. VLIW
        - b. EPIC
          - i. ROSC — разновидность стековой архитектуры.
    - iii. VLIW
      1. EPIC
  - c. Способ доступа к операндам
    - i. Аккумуляторная
    - ii. Регистровая
    - iii. Стековая
    - iv. С ограниченным доступом памяти (**LOAD / STORE ARCHITECTURE**)
3. Характеристики Машинных Команд
  - a. Структура
  - b. Формат
  - c. Адресность (0-4)
  - d. Функция (5-7)
  - e. Безопасность

**Архитектура набора команд** — логическая прослойка между аппаратной частью и системным ПО.

Одна из основных функций этого объекта — обеспечение баланса между возможностями разработчиков HD (hardware) и и потребностями разработчиков SW (software): операционных систем и компиляторов.



Semantic GAP

HLL v.s. M.I.

Людям удобно писать код, используя выконагруженные семантические структуры, а машине, наоборот, удобно обрабатывать “лёгкие” команды. Так появилась ISA.

Стековые архитектуры использовались в:

- GAP
- IGNITION
- Patriot
- ЭЛЬБРУС-...
- CJIP

## Классификация по неоднородности и сложности набора команд

CISC — исторически первая архитектура.

Предпосылки к её появлению:

- В 60-х годах 16 килобайт стоили 500\$. Поэтому хотелось экономить память, поэтому набор машинных команд должен уметь реализовывать достаточно сложные структуры (оператор цикла и т.д.)
- Программистам было удобнее писать более сложные структуры.
- Компиляторы не были достаточно хорошо развиты.  
Представители: IBM360(1964 год), DEC VAX.

Характеристики CISC:

1. Набор команд вычислительно плотный.  
В наборе много команд, некоторые реализуют сложную логику.
2. Переменная длина команд.
3. Малое количество регистров общего назначения: 8-12 штук.
4. Большое количество способов адресации.
5. Отсутствие предопределённого времени выполнения большинства команд.
6. Очень многие команды могли напрямую обращаться в оперативную память (но это тратит дополнительный такт).



# Оглавление

1. Устройство простейшего компьютера и адресация
2. Принципы фон Неймана
3. УУ и АЛУ. Типы команд. Измерение производительности компьютера.
4. [Системы счисления. Двоичная, восьмеричная и шестнадцатеричная арифметика.](#)
5. [Цифровая логика и операции над битами](#)
6. Простейшие способы оптимизации выполнения команд. CISC и RISC. Принципы RISC.
7. Методы работы с внешними устройствами. Типы прерываний и структура обработчика.
8. Представление данных в ЭВМ. Форматы данных. Представление целых чисел.
9. Представление данных в ЭВМ. Форматы данных. Представление чисел с плавающей точкой.
10. [Представление данных в ЭВМ. Форматы данных. Символьные данные. Массивы. Строки. Стек.](#)
11. Представление данных в ЭВМ. Форматы данных. BCD. Структуры. Специальные типы данных.
12. Методы адресации и использование регистров при адресации. Непосредственная, прямая, регистровая и косвенная регистровая адресация.
13. Методы адресации и использование регистров при адресации. Индексная и относительная индексная адресация. Использование стека при адресации.
14. Методы адресации и использование регистров при адресации. Представление адреса в командах перехода. Представление адреса с использованием сегментных регистров.
15. [Три основные архитектуры организации кэшa](#)
16. [Кэш. Типы кэш-памяти по стратегии обновления основной памяти. Механизмы замещения строк. Организация кэш-памяти в современных ЭВМ.](#)
17. Архитектура с общей шиной. Децентрализованный арбитраж.
18. Архитектура с общей шиной. Централизованный арбитраж. Структура приоритетов при централизованном арбитраже.
19. Архитектура ([Архитектура ЦП 8086](#)) с общей шиной. Механизмы обмена данными.
20. Организация конвейера команд. Скалярный, суперскалярный и суперконвейерный вычислитель.
21. Основы схемотехники, базовые элементы, конструирование булевых функций.
22. Предсказание переходов. Регистровые окна и переименование регистров.
23. Классификация Флинна с примерами реализации архитектур.
24. Архитектуры VLIW и EPIC. Особенности спекулятивного исполнения инструкций в архитектуре EPIC.
25. Закон Амдала.
26. Дополнения Ванга и Бриггса к классификации Флинна.
27. Архитектура MIPS
28. Согласование кэшей в мультипроцессорных системах и многоядерных процессорах.
29. Архитектура SPARK
30. Характеристики машинных команд
31. Архитектура системы команд. Три основные классификации, примеры.
32. Сегментная модель памяти. Страничная модель памяти
33. [Режимы работы процессоров INTEL x86. Уровни привилегий в защищенном режиме.](#)
34. Работа процессоров INTEL x86 в защищенном режиме.
35. [Физическая организация DRAM](#)
36. Типы DRAM, схемы пакетных циклов

[Дополнительная информация](#)

[Распределение](#)

# 1. Устройство простейшего компьютера и адресация

В простейшем компьютере можно выделить следующие устройства: процессор, память и устройства ввода-вывода



**Центральный процессор**— электронный блок либо интегральная схема (микропроцессор), исполняющая машинные инструкции (код программ), главная часть аппаратного обеспечения компьютера или программируемого логического контроллера. Иногда называют *микропроцессором* или просто *процессором*. Он вызывает команды из памяти, определяет их тип, а затем выполняет их одну за другой. Процессор управляет работой всех устройств компьютера.

Процессор состоит из нескольких частей. УУ (устройство управления) отвечает за вызов команд из памяти и определение их типа. АЛУ выполняет арифметические и логические операции. Внутри центрального процессора находится память для хранения промежуточных результатов и некоторых команд управления. Эта память состоит из нескольких регистров, каждый из которых выполняет определенную функцию. Обычно все регистры одинакового размера. Каждый регистр содержит одно число, которое ограничивается размером регистра. Регистры считываются и записываются очень быстро, поскольку они находятся внутри центрального процессора.

Самый важный регистр — счетчик команд, который указывает, какую команду нужно выполнять дальше.

**Шина** – в архитектуре компьютерная подсистема, которая передаёт данные между функциональными блоками компьютера. Представляет собой набор параллельно связанных проводов, по которым передаются адреса, данные и сигналы управления. Шины можно разделить на группы в соответствии с выполняемыми функциями. Они могут быть внутренними по отношению к процессору и служить для передачи данных в АЛУ и из АЛУ, а могут быть внешними по отношению к процессору и связывать процессор с памятью или устройствами ввода-вывода.

**Память** – часть компьютера, где хранятся программы и данные. Память может принимать информацию из других устройств, запоминать ее и выдавать по запросу



другим устройствам. Память состоит из ячеек, расположенных линейно друг за другом, каждая из которых может хранить некоторую порцию информации. Каждая ячейка имеет номер, который называется адресом, по адресу можно ссылаться на определенную ячейку. Если память содержит  $n$  ячеек, они будут иметь адреса от 0 до  $n-1$ . Все ячейки памяти содержат одинаковое число битов. Если ячейка состоит из  $k$  битов, она может содержать любую из  $2^k$ -комбинаций.

**Бит** – основная единица памяти, двоичный разряд. Бит может содержать 0 или 1. Эта самая маленькая единица памяти.

**Байт** – минимальный адресуемый блок памяти (ячейка), обычно 8 бит.

Для представления чисел размер байта не является достаточным (Смотря каких, на самом деле. Вроде как есть типы данных, хранящие однобайтовые числа). Поэтому обычно байты объединяются в пары: нулевой с первым, второй с третьим, четвертый с пятым т.д. Такая пара байтов с соседними адресами, из которых меньший адрес четный, называется **словом**. С точки зрения архитектуры, машинное слово – это минимальный объем данных, которым могут обмениваться между собой различные узлы. Размер слова обычно соответствует разрядности регистра данных или шины. Такая единица, как слово, необходима, поскольку большинство команд производят операции над целыми словами. Биты в слове нумеруются с нуля и справа налево.

**Слово** - машинно-зависимая и платформозависимая величина, измеряемая в битах или байтах, равная разрядности регистров процессора и/или разрядности шины данных.

В ранних ЭВМ размер слова совпадал с минимальным размером адресуемой информации (разрядностью данных, расположенных по одному адресу); в современных машинах минимальным адресуемым блоком информации называется байт, а слово состоит из нескольких байтов. (wiki)

Машинное слово определяет следующие характеристики аппаратной платформы:

- разрядность данных, обрабатываемых процессором
- разрядность адресуемых данных (разрядность шины данных);
- максимальное значение беззнакового целого типа, напрямую поддерживаемого процессором: если результат арифметической операции превосходит это значение, то происходит переполнение;
- максимальный объем оперативной памяти, напрямую адресуемой процессором.

Для 32-битных процессоров x86: исторически машинным словом считается 16 бит, реально — 32 бита.

**Двойное слово** – образовано из двух слов(как ни странно) с соседними адресами, из которых меньший адрес кратен четырем(а в общем случае - двум длинам слова)

## 2. Принципы фон Неймана

Согласно фон Нейману, ЭВМ состоит из следующих основных блоков:

- Устройства ввода/вывода информации

- Память компьютера
- Процессор, состоящий из **устройства управления** (УУ - интерфейс между АЛУ и внешним миром) **арифметико-логического устройства** (АЛУ выполняет команды). АЛУ и УУ определяют действия, подлежащие выполнению, путем считывания команд из оперативной памяти.

1) **Binary coding (Двоичное кодирование)** Все данные в памяти хранятся в двоичной коде.

2) **Линейность и однородность памяти** Память представляет собой линейную упорядоченную систему ячеек (ячейки пронумерованы от 0 до  $2^n-1$ ). Линейность -> Однородность.

Однородность в узком смысле: в каждый момент времени время доступа к произвольной ячейке памяти не зависит от номера. Аналогично по записи. RAM (Random access memory)

Однородность в широком смысле: программа хранится в той же памяти, что и данные.

3) **Принцип программного управления** В узком смысле: процессор работает под управлением программы. Программа работает последовательно (автоматически). В широком смысле: машина работает автоматически, под управлением программы путём последовательного выполнения её команд, хранящихся в памяти.

4) (На лекции не упоминалось) **Принцип жесткости архитектуры**

Неизменяемость в процессе работы топологии, архитектуры, списка команд.

**Счетчик команд** – один из регистров процессора. Счетчик команд служит для автоматической выборки команды из последовательных ячеек памяти.

**Регистры** – это определенные участки памяти внутри самого процессора, который используется для промежуточного хранения информации, обрабатываемой процессором.

Некоторые регистры содержат только определенную информацию. Есть специальный **регистр флагов** (16-битовый). Этот регистр содержит информацию, которая используется побитно, а не в качестве 16-битового числа. Биты флагового регистра имеют значение для процессора по-отдельности. Некоторые из этих бит содержат коды условий, установленные последней выполненной командой. Программа пользуется этими кодами для управления своим выполнением. Программа может тестировать коды условий и на основе полученных значений выбирать последовательность выполнения. Другие биты в регистре флагов показывают состояние процессора при выполнении текущей команды. При условном переходе выполняется проверка комбинации флагов и затем принимается решение менять или нет IP. **Команды переходов** – команды, смысл которых – изменение счетчика команд, они содержат адрес следующей команды. **Безусловный переход** (jump) – изменяет IP (не доходит до АЛУ). **Условный переход** – проверка условия и переход. **Другие команды:** арифметические, прерывания, логические;

**Память** – часть компьютера, где хранятся программы и данные. Память может принимать информацию из других устройств, запоминать ее и выдавать по запросу другим устройствам. Память состоит из ячеек, расположенных линейно друг за другом, каждая из которых может хранить некоторую порцию информации. Каждая ячейка имеет номер, который называется адресом. По адресу можно ссылаться на определенную ячейку. Если память содержит  $n$  ячеек, они будут иметь адреса от 0 до  $n-1$ . Все ячейки памяти содержат одинаковое число битов. Если ячейка состоит из  $k$  битов, она может содержать любую из  $2^k$ -комбинаций.

Большинство команд можно разделить на две группы: команды типа регистр-память и типа регистр-регистр. Команды первого типа вызывают слова из памяти, помещают их в регистры, где они используются в качестве входных данных АЛУ. Другие команды этого типа помещают регистры обратно в память. Команды второго типа вызывают два операнда из

регистров, помещают их во входные регистры АЛУ, выполняют над ними какую-нибудь арифметическую или логическую операцию и переносят результат обратно в один из регистров.

### 3. УУ и АЛУ. Типы команд. Измерение производительности компьютера.

Центральный процессор включает в себя устройство управления (УУ) и арифметико-логическое устройство (АЛУ). АЛУ – выполняет сложение, вычитание и другие простые операции над входными данными и помещает результат в выходной регистр. Этот выходной регистр может помещаться обратно в один из регистров. Он может быть сохранен в памяти, если это необходимо. Входные и выходные регистры есть не у всех компьютеров. Регистр-регистр (операции и чтение/запись из регистра). УУ – управляет всеми остальными устройствами ЭВМ путём посылки управляющих сигналов, подчиняясь которым остальные устройства производят определённые действия, предписанные этими сигналами. УУ имеет собственные регистры (счетчик команд, регистр команд и т.д.). В регистрах хранятся, в том числе, команды для исполнения процессором.

**Типы команд: Регистр-память** – вызывают слова из памяти, помещают их в регистры, где они используются в качестве входных данных АЛУ, либо помещают регистры обратно в память.

**Регистр-регистр** – вызывают два операнда из регистров, помещают их во входные регистры АЛУ, выполняют над ними какую-нибудь арифметическую или логическую операцию и переносят результат обратно в один из регистров.

1) **Команды перемещения данных** Копирование данных из одного места в другое – одна из самых распространенных операций. Под копированием мы понимаем создание нового объекта с точно таким же набором битов, как у исходного объекта.

2) **Бинарные операции** Бинарные операции берут два операнда и получают из них результат. Все архитектуры команд содержат команды для сложения и вычитания целых чисел. Команды умножения и деления целых чисел также имеются практически во всех случаях. Следующая группа бинарных операций содержит булевы команды.

Существует 16 булевых функций от двух переменных. Обычно присутствуют И, ИЛИ и НЕ.

3) **Унарные операции** Унарные операции используют один операнд и производят один результат. Сдвиг, циклический сдвиг, прибавление 1.

4) **Сравнения и условные переходы** Практически все программы должны проверять свои данные и на основе результатов изменять последовательность команд, которые нужно выполнить. Команды условного перехода проверяют какое-либо условие и совершают переход в определенный адрес памяти, если условие выполнено. Сравнение слов, сравнение с 0, и т.д.

5) **Команды вызова процедур** Процедура — это группа команд, которая выполняет определенную задачу и которую можно вызвать из нескольких мест программы.

6) **Управление циклом** Часто возникает необходимость выполнять некоторую группу команд фиксированное количество раз, поэтому некоторые машины содержат команды для облегчения этого процесса. Все эти схемы содержат счетчик, который увеличивается или уменьшается на какую-либо константу каждый раз при выполнении цикла. Кроме того, этот счетчик каждый раз проверяется. При выполнении определенного условия цикл завершается.

#### 7) **Команды ввода-вывода**

**Производительность** – скорость, с которой процессор производит операции. **Тактовый генератор** – источник импульсных сигналов в компьютере, который синхронизирует каждую операцию процессора. Все импульсы одинаковы по длительности. Интервалы между последовательными импульсами также одинаковы. Временной интервал между началом одного импульса и началом следующего называется временем такта. **Тактовая частота** – скорость, с которой микропроцессор выполняет команды. Центральному процессору (CPU) необходимо фиксированное количество тактов системных часов (или тактовых циклов) для выполнения каждой команды. Чем быстрее часы,

тем больше команд процессор исполняет в секунду. **MIPS** (Millions of Instructions per Second – «м и л л и о н ы к о м а н д в с е к у н д у») – величина, которая показывает сколько миллионов инструкций в секунду выполняет процессор. Тактовая частота выражается в **мегагерцах (MHz)**, 1 MHz = 1 миллиону циклов в секунду. Характеризует число основных операций компьютера, производимых в секунду, измеряется в герцах. Различные шины и чипы на материнской плате компьютера могут иметь различную тактовую частоту.

## 4. Системы счисления. Двоичная, восьмеричная и шестнадцатеричная арифметика.

*Ключевые слова:* Преобразование чисел, примеры умножения и деления.

*Примечание: в принципе можете вообще не читать текст, а просто посмотреть на примеры, на них все наглядно объяснено)*

### Алгоритм

Сначала про преобразование чисел из десятичной в  $k$ -ичную и наоборот. Если  $k$  - это степень двойки (как нужно в билете), то рекомендуется для вычислений использовать двоичную систему счисления, как промежуточную. То есть для перевода числа из десятичной в восьмеричную нужно сначала перевести число из десятичной в двоичную, а потом из двоичной в восьмеричную. Для того, чтобы перевести число из восьмеричной или шестнадцатеричной системы счисления в двоичную нужно просто каждую цифру числа заменить на ее представление в двоичной системе счисления.

Для обратного преобразования достаточно разбить двоичное число на блоки длины 3 и 4 для восьмеричной и шестнадцатеричной систем счисления соответственно (т.к.  $8 = 2^3$  и  $16 = 2^4$ ) и заменить каждый блок на цифру в нужной нам системе счисления.

Теперь про переводы между двоичной и десятичной системами счисления. Для перевода из десятичной системы счисления в двоичную, как и наоборот, нужно запомнить такое правило: целая и дробная части переводятся по отдельности, независимо друг от друга. Для перевода целой части в другую систему счисления нужно выполнять следующий алгоритм:

Пусть  $x$  - целая часть переводимого числа,  $b$  - основание системы счисления, в которую мы переводим,  $y$  - **строковое** представление  $x$  в новой системе счисления, тогда перевод выглядит так:

```
while x != 0:  
    y = (x % b) + y  
    x /= b
```



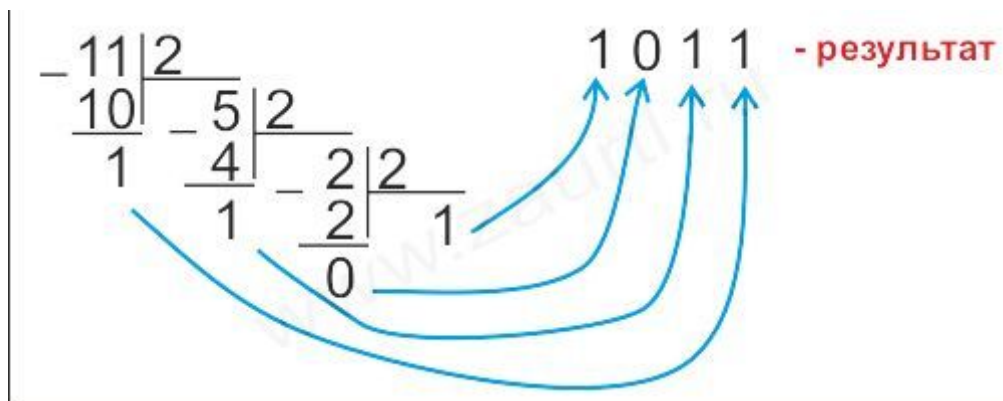
Для перевода дробной части воспользуемся теми же обозначениями за исключением того, что теперь  $x$  и  $y$  будут представлять дробные части переводимых чисел:

```
while x != 0:
    x *= b
    digit = floor(x)
    x -= digit
    y = y + digit
```

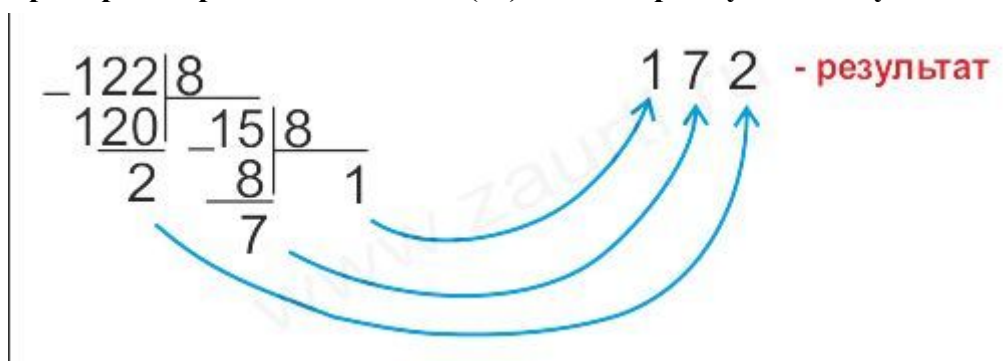
## Примеры

Примеры (в примерах двоичная система счисления не используется, как промежуточная - так тоже можно, но это уже сложнее):

**Пример 1. Перевести число 11(10) в двоичную систему счисления.**



**Пример 2. Перевести число 122(10) в восьмеричную систему счисления.**



**Пример 3. Перевести число 500(10) в шестнадцатеричную систему счисления.**

$$\begin{array}{r|l}
 -500 & 16 \\
 \hline
 496 & 31 \quad 16 \\
 \hline
 4 & 16 \quad 1 \\
 \hline
 & 15
 \end{array}
 \begin{array}{l}
 \nearrow 1 \\
 \nearrow F \\
 \nearrow 4
 \end{array}
 \text{ - результат}$$

Пример 4. Перевести число 0,625(10) в двоичную систему счисления.

$$\begin{array}{r}
 * 0,625 \\
 \hline
 2 \\
 \hline
 1,25
 \end{array}
 \nearrow
 \begin{array}{r}
 * 0,25 \\
 \hline
 2 \\
 \hline
 0,5
 \end{array}
 \nearrow
 \begin{array}{r}
 * 0,5 \\
 \hline
 2 \\
 \hline
 1,0
 \end{array}
 \nearrow 0,0$$

$$\begin{array}{c}
 \downarrow \\
 1
 \end{array}
 \quad
 \begin{array}{c}
 \downarrow \\
 0
 \end{array}
 \quad
 \begin{array}{c}
 \downarrow \\
 1
 \end{array}
 \rightarrow 0,101_{(2)}$$

направление чтения

Пример 4. Перевести число 0,6(10) в восьмеричную систему счисления.

$$\begin{array}{r}
 * 0,6 \\
 \hline
 8 \\
 \hline
 4,8
 \end{array}
 \nearrow
 \begin{array}{r}
 * 0,8 \\
 \hline
 8 \\
 \hline
 6,4
 \end{array}
 \nearrow
 \begin{array}{r}
 * 0,4 \\
 \hline
 8 \\
 \hline
 3,2
 \end{array}$$

$$\begin{array}{c}
 \downarrow \\
 4
 \end{array}
 \quad
 \begin{array}{c}
 \downarrow \\
 6
 \end{array}
 \quad
 \begin{array}{c}
 \downarrow \\
 3
 \end{array}
 \rightarrow 0,463_{(8)}$$

направление чтения

точность 3 знака после запятой

Пример 5. Перевести число 0,7(10) в шестнадцатеричную систему счисления.

$$\begin{array}{r}
 * 0,7 \\
 \hline
 16 \\
 \hline
 11,2
 \end{array}
 \nearrow
 \begin{array}{r}
 * 0,2 \\
 \hline
 16 \\
 \hline
 3,2
 \end{array}
 \nearrow
 \begin{array}{r}
 * 0,2 \\
 \hline
 16 \\
 \hline
 3,2
 \end{array}
 \nearrow
 \begin{array}{r}
 * 0,2 \\
 \hline
 16 \\
 \hline
 3,2
 \end{array}$$

$$\begin{array}{c}
 \downarrow \\
 11
 \end{array}
 \rightarrow B
 \quad
 \begin{array}{c}
 \downarrow \\
 3
 \end{array}
 \quad
 \begin{array}{c}
 \downarrow \\
 3
 \end{array}
 \quad
 \begin{array}{c}
 \downarrow \\
 3
 \end{array}
 \rightarrow 0,B333_{(16)}$$

направление чтения

точность 4 знака после запятой

Пример 6. Перевести число  $101,11_{(2)}$  в десятичную систему счисления.

$$\overset{2}{1}\overset{1}{0}\overset{0}{1},\overset{-1}{1}\overset{-2}{1}_{(2)} \rightarrow_{(10)} = 1 \cdot 2^2 + 0 \cdot 2^1 + 1 \cdot 2^0 + 1 \cdot 2^{-1} + 1 \cdot 2^{-2} = 5,75_{(10)}$$

Пример 7. Перевести число  $57,24_{(8)}$  в десятичную систему счисления.

$$\overset{1}{5}\overset{0}{7},\overset{-1}{2}\overset{-2}{4}_{(8)} \rightarrow_{(10)} = 5 \cdot 8^1 + 7 \cdot 8^0 + 2 \cdot 8^{-1} + 4 \cdot 8^{-2} = 47,3125_{(10)}$$

Пример 8. Перевести число  $7A,84_{(16)}$  в десятичную систему счисления.

$$\overset{1}{7}\overset{0}{A},\overset{-1}{8}\overset{-2}{4}_{(16)} \rightarrow_{(10)} = 7 \cdot 16^1 + 10 \cdot 16^0 + 8 \cdot 16^{-1} + 4 \cdot 16^{-2} = 122,515625_{(10)}$$

Пример 9. Записать число  $16,24_{(8)}$  в двоичной системе счисления.

$$\begin{array}{c} 16,24 \text{ - восьмеричное число} \\ \swarrow \quad \downarrow \quad \downarrow \quad \searrow \\ \underbrace{001 \ 110 \ 010 \ 100} \\ 1110,0101 \text{ - двоичное значение} \end{array}$$

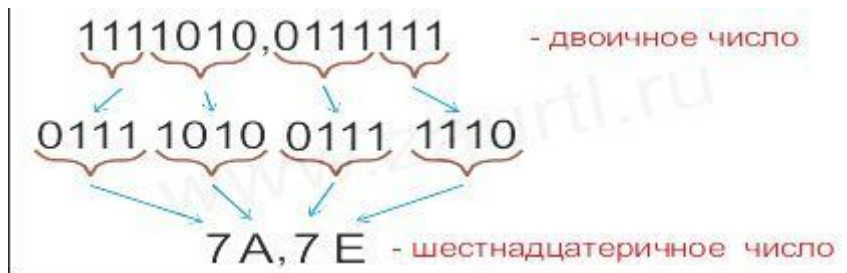
Пример 10. Записать число  $1110,0101_{(2)}$  в восьмеричной системе счисления.

$$\begin{array}{c} 1110,0101 \text{ - двоичное число} \\ \swarrow \quad \downarrow \quad \downarrow \quad \searrow \\ \underbrace{001 \ 110 \ 010 \ 100} \\ \swarrow \quad \downarrow \quad \downarrow \quad \searrow \\ 16,24 \text{ - восьмеричное значение} \end{array}$$

**Пример 11. Записать число 7A,7E(16); в двоичной системе счисления.**



**Пример 12. Записать число 1111010,0111111(2) в шестнадцатеричной системе счисления.**



Статью с примерами и пояснениями можно найти [тут](#).

## 5. Цифровая логика и операции над битами

### Введение

В современном CPU все операции выполняются посредством базовых логических операций {NOT, AND, OR, XOR} и т.д. Однако (согласно дискретной математике) все они могут быть выражены через штрих Шеффера: NAND

**Gate** - логический вентиль - <в узком смысле> цифровая схема реализующая базовую булеву функцию. <в широком смысле> - любую функцию

**Регистр флагов** - регистр CPU в архитектуре x86. Значения его флагов (в том числе) влияют и на АЛ-операции

Точный дамп регистра флагов процессора 8086:

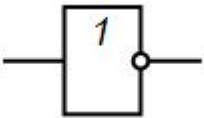
OF	переполнение в результате арифметической операции
DF	флаг направления
IF	флаг прерывания
TF	флаг трассировки
SF	минус в результате арифметической операции
ZF	ноль в результате арифметической операции
AF	вспомогательный флаг
PF	флаг четности
CF	флаг переноса



## Основы цифровой логики

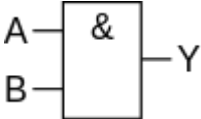
### NOT

$$\text{NOT}(X) = \text{NAND}(X, X)$$

	X	NOT(X)
	0	1
	1	0

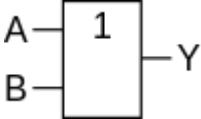
### AND

$$\text{AND}(X, Y) = \text{NOT}(\text{NAND}(X, Y)) = \text{NAND}(\text{NAND}(X, Y), \text{NAND}(X, Y))$$

	X	Y	AND(X, Y)
	0	0	0
	0	1	0
	1	0	0
	1	1	1

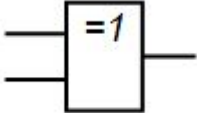
### OR

$$\text{OR}(X, Y) = \text{NAND}(\text{NOT}(X), \text{NOT}(Y)) = \text{NAND}(\text{NAND}(X, X), \text{NAND}(Y, Y))$$

	X	Y	OR(X, Y)
	0	0	0
	0	1	1
	1	0	1
	1	1	1

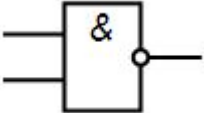
### XOR

$$\text{XOR}(X, Y) = \text{AND}(\text{NAND}(X, Y), \text{OR}(X, Y)) = \dots$$

	X	Y	XOR(X, Y)
	0	0	0
	0	1	1
	1	0	1
	1	1	0

## NAND

$\text{NAND}(X, Y) = \text{NAND}(X, Y)$  🤪

	X	Y	NAND(X, Y)
	0	0	1
	0	1	1
	1	0	1
	1	1	0



## Сумматор

A - первый вход

B - второй вход

iOF - вход переполнения

R - результат

OF - выход переполнения

### Однобитный полусумматор

Построим таблицу истинности для такого сумматора

A	B	R	OF	<p>Заметим, что таблица истинности для суммы в точь соответствует таблице истинности для <math>XOR(A, B)</math></p> <p>В свою очередь - <math>OF - AND(A, B)</math></p> 
0	0	0	0	
0	1	1	0	
1	0	1	0	
1	1	0	1	
<p>A - первый вход B - второй вход R - результат OF - выход переполнения</p>				

## Одноразрядный сумматор

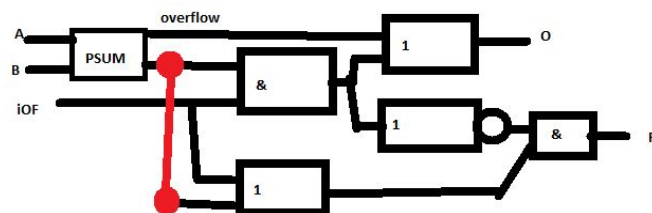
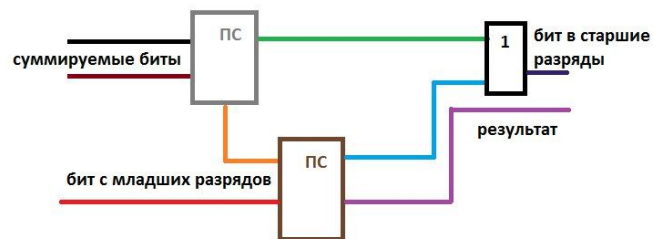
Построим таблицу истинности для такого сумматора

A	B	iOF	R	OF
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

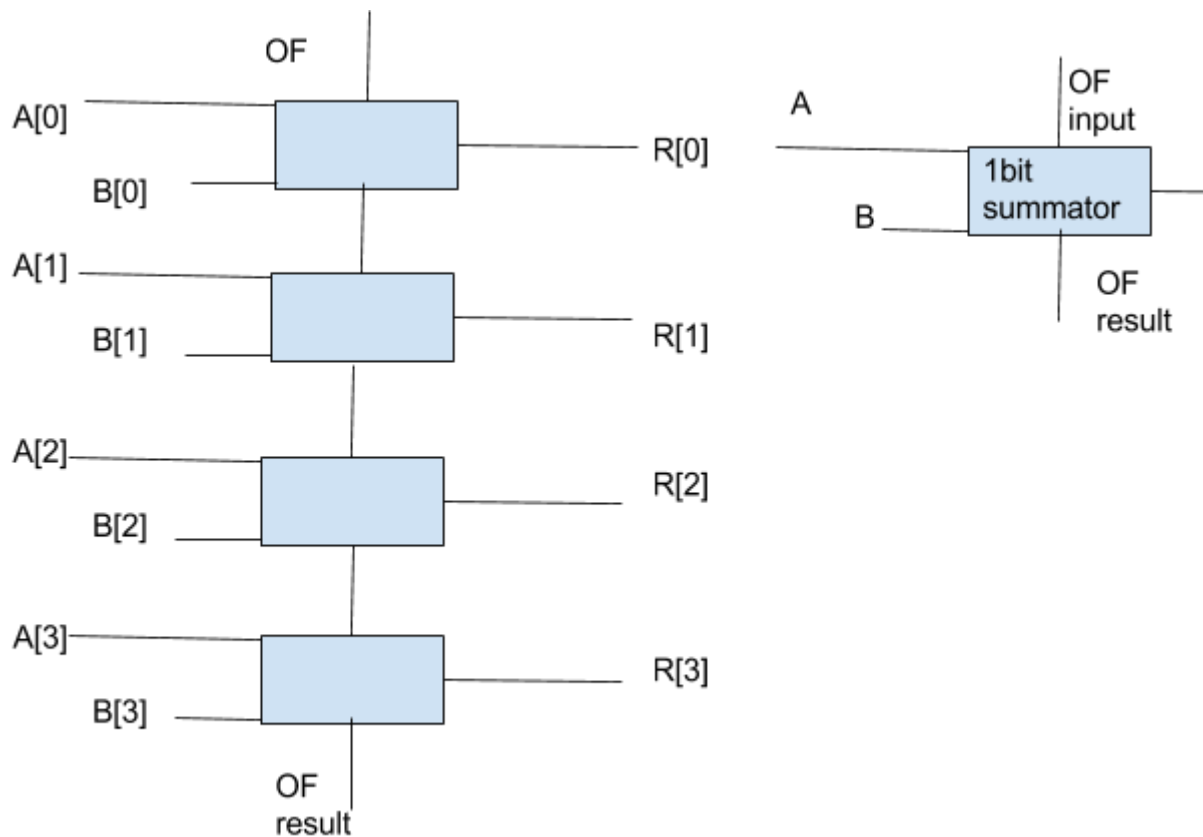
A - первый вход  
B - второй вход  
iOF - вход переполнения  
R - результат  
OF - выход  
переполнения

Вы можете построить ДНФ и получить искомую схему  
Либо можете поверить мне на слово

PSUM и ПС - полусумматор.  
Ниже приведены две разные схемы реализации  
полного сумматора:



## Многобитный сумматор



На примере 4х битного, мы можем понять, что сумматоры стакабельны

Так же существует схема построения многобитного сумматора, когда мы выполняем сложение не байтов, а [нибблов](#). Для старшего ниббла считаем сразу два значения (с поднятым флагом OF и с опущенным). Затем (в зависимости от результата вычисления младшего ниббла) выбираем одно из значений (случилось/не случилось переполнение)

# Вычисления

## Операция сдвига

Операция сдвига эквивалентна умножению на два и выполняется компьютером весьма быстро

$100010101 \rightarrow \text{rol } 3 \rightarrow 100010101000$

$100010101 \rightarrow \text{ror } 2 \rightarrow 1000101$  При этом флаг переполнения выставится в 1

## Сложение беззнаковых

Сложение происходит с использованием сумматора. При этом меняются флаги **OF** - переполнение и **ZF** - признак нуля. Также используется флаг **OF** для входа как переполнение

## Умножение

Умножение происходит через умножение столбиком со сдвигом:

$101 * 1101:$

$(101 \ll 0) * 1 + (101 \ll 1) * 0 + (101 \ll 2) * 1 + (101 \ll 3) * 1$

## Польская нотация

Используется для упрощения декодирования и выполнения последовательности подряд идущих вычислений

Также известна, как префиксная

В чем суть - оператор расположен слева от операндов

Вычисления идут справа налево вплоть до того, как мы встретим оператор.

После чего выполняем оператор над пройденными операндами

Результат заносим в стек справа

Так до тех пор, пока на стеке не останется ни одного оператора

[Подробнее](#)

**Пример:**

1)  $+1+23$

2)  $+15$

3)  $6$

## 6. Простейшие способы оптимизации выполнения команд. CISC и RISC. Принципы RISC.

Можно выделить два способа оптимизации выполнения команд:

- 1) Конвейеризация
- 2) Суперскалярность

Конвейеризация: по идее, выполнение команды может быть разбито на несколько основных этапов, которые можно назвать микрокомандами (считывание, формирование адреса следующей команды, дешифрация, выборка операндов, выполнение операции, запись результата). Каждая операция выполняется за такт. Порядок вызова строго регламентирован. Каждая микрокоманда требуется лишь один раз для каждой команды, т.е. после того как она отработала, в текущей команде она больше использоваться не будет и, следовательно, может быть использована в следующей. При Конвейеризации в каждый момент времени осуществляется параллельная обработка команд, т.е. в каждый момент одна команда считывается, другая декодируется и т.д. Рассмотренная технология обработки команд носит название *конвейерной обработки*. Каждая часть устройства называется *ступенью конвейера*, а общее число ступеней – *длиной линии конвейера*. Команды перехода всё рушат. Конвейерность впервые была использована в IBM 801(?), но широкое применение получила в MIPS.

Суперскалярность: Используется два и более конвейеров. Встретив команду перехода, процессор на первом конвейере продолжает работы на случай, если переход не произойдет, а второй конвейер запускает с адреса, на который переход может произойти. В некоторых системах также используется конвейер данных. У Pentium-а два конвейера команд.

CISC - complex instruction set computer. Большое число различных по формату и длине команд, различные режимы адресации, сложная кодировка инструкций. Присутствует одна внутренняя шина, на процессоре. Задачи регистров строго распределены. Направлена на счёт(?).

RISC - reduced instruction set computer. Система команд имеет упрощенный вид. Все команды одинакового формата с простой кодировкой инструкций. Шины команд и данных разделены. Обращение к памяти происходит через команды загрузки и записи, в остальных случаях используются регистры, т.е. берется из регистра, в него же и записывается (в смысле в регистр). Меньше логических элементов. Большинство команд выполняется за 1 такт. Минимальное число регистров (общего назначения) - 32.

Оптимизация использования регистров:

1) Программная оптимизация выполняется на этапе компиляции программы, написанной на языке высокого уровня. Компилятор стремится распределить регистры процессора таким образом, чтобы разместить в них те переменные, которые в течение заданного периода времени будут использоваться наиболее интенсивно.

2) Аппаратная оптимизация использования регистров в RISC-процессорах ориентирована на сокращение затрат времени при работе с процедурами. Наибольшее время в программах, написанных на языках высокого уровня, расходуется на вызовы процедур и возврат из них. Связано это с созданием и обработкой большого числа локальных переменных и констант. Используются регистровые окна, т.е. блоки регистров отводятся под глобальные переменные, конкретные методы и пр.

Матрица шин (Системная шина и шина данных подключаются к внешнему микроконтроллеру через набор высокоскоростных шин).

Четыре основных принципа RISC-архитектуры:

- \* каждая команда независимо от ее типа выполняется за один машинный цикл, длительность которого должна быть максимально короткой
- \* все команды должны иметь одинаковую длину и использовать минимум адресных форматов
- \* обращение к памяти происходит только при выполнении операций записи и чтения, вся обработка данных осуществляется исключительно в регистровой структуре процессора
- \* система команд должна обеспечивать поддержку языка высокого уровня. (Имеется в виду подбор системы команд, наиболее эффективной для различных языков программирования.)

Черты организации CISC-процессоров:

- 1) большое количество различных машинных команд (сотни), каждая из которых выполняется за несколько тактов центрального процессора
- 2) устройство управления с программируемой логикой
- 3) небольшое количество регистров общего назначения (РОН)
- 4) различные форматы команд с разной длиной
- 5) преобладание двухадресной адресации(?)
- 6) развитый механизм адресации операндов, включающий различные методы косвенной адресации.

Сравнение:

CISC	RISC
Многобайтовые команды	Однобайтовые команды
Малое кол-во регистров	Большое кол-во регистров
Сложные команды	Простые команды
Мало команд за цикл( $\leq 1$ )	Много команд за цикл
Одно исполнительное уст-во	Много исполнительных уст-в

## **7. Методы работы с внешними устройствами. Типы прерываний и структура обработчика.**



## 8. Представление данных в ЭВМ. Форматы данных.

### Представление целых чисел.

#### 1. Беззнаковые (для неотрицательных чисел)

Для беззнакового представления все разряды ячейки отводят под представление самого числа. В одном байте можно представить числа в диапазоне  $[0, 255]$

#### 2. Знаковые

##### Явно знаковые

В представлении числа есть бит информации о знаке, остальные биты под само число. Если число положительное, то в знаковом бите – 0, если отрицательное – 1.

Достоинства:

- о Очевидность
- о Легко считать
- о Симметричный диапазон  $[-127, 127]$

Недостатки:

- о  $\pm 0$

#### Смещенный формат (BIAS)

Хранится линейное преобразование числа. Выбирается константа(смещение) и вместо числа  $x$  хранится число  $x + \text{константа}$ .

На практике константа равна либо (диапазон:  $[-128, 127]$ ), либо (диапазон:  $[-127, 128]$ ).

Все современные процессоры работают со сдвигом .

Достоинства:

- о Нет проблемы двух нулей.

Недостатки:

- о Усложненная реализация, при арифметических операциях нужно учитывать смещение, то есть проделывать на одно действие больше.

#### Комплементарный формат (с дополнительным кодом)

Алгоритм получения дополнительного кода числа:

- если число неотрицательное, то в старший разряд записывается ноль, далее записывается само число (в двоичном виде);
- если число отрицательное, то все биты модуля числа инвертируются, то есть все единицы заменяются на нули, а нули — на единицы, к инвертированному числу прибавляется единица, далее к результату дописывается знаковый разряд, равный единице.

Достоинства:

- о Возможность заменить арифметическую операцию вычитания операцией сложения с дополнительным кодом другого числа.

- о Нет проблемы двух нулей.

Недостатки:

- о Ряд положительных и отрицательных чисел несимметричен, но это не так важно.

Свойства целых чисел:

1. Ограниченный диапазон.
2. В общем случае результатом арифметической операции может быть недопустимое число (из-за ограниченного диапазона)
  - Промежуточный результат может быть семантически не значим (например,  $4/3$ ).
  - Промежуточный результат может быть за рамками диапазона
3. В некоторых случаях базовые законы арифметики не работают (ассоциативность)

## 9. Представление данных в ЭВМ. Форматы данных.

### Представление чисел с плавающей точкой.

*Примечание: рассматривается только содержимое оперативной памяти. Все данные представляются в двоичном виде.*

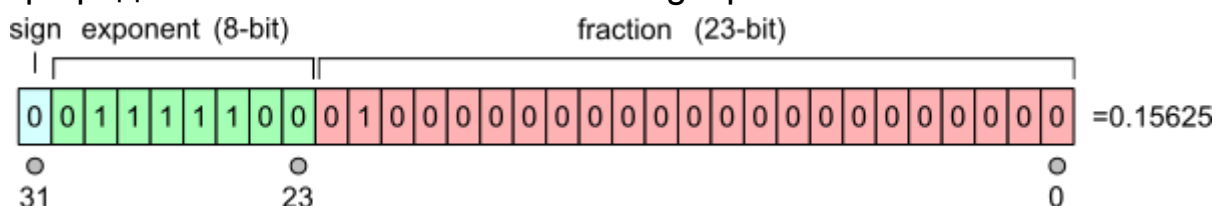
Описание формата IEEE-754.

Числа в формате float IEEE-754 хранятся с помощью числа мантиссы, порядка и знака. Существуют числа одинарной, двойной и расширенной точности.

Precision	size	exponent	mantissa
single	32 bits	8	23
double	64 bits	11	52
extended	80 bits (?)	15	64

Само число хранится в виде  $(-1)^s \cdot 1.M \cdot 2^e - bias$ , где  $s$  - бит знака,  $M$  - мантисса,  $e$  - порядок,  $bias$  - смещение порядка,  $bias(e) = 2^{e-1} - 1$ . Нужно сразу заметить, что т.к. любое число больше нуля в старшем разряде двоичного вида всегда имеет единицу, конкретно эту единицу не хранят, но при переводе всегда подразумевают. Еще одной причиной (а вообще, главной) не хранения единицы является нормализация числа. Если бы мантисса полностью бы записывалась в числе, возникало бы множество представлений одного и того же числа, что снижало бы количество чисел, которые можно представить в формате. Числа такого вида называются нормализованными.

Пример представления числа 0.15625 в single precision.



(Число вычисляется следующим образом: мантисса с учетом неявного бита содержит: 1.01, порядок равен

$01111100_2 - bias = 124_{10} - 127 = -3$ , само число равно

$$1.01 \cdot 2^{-3} = 0.00101 = 0.15625$$

В стандарте предусмотрены специальные значения:  $\pm 0$ ,  $\pm \infty$ , *NaN*

При нулевой мантиссе и порядке число считается равным  $\pm 0$ .

При максимальном порядке (все биты равны 1) и нулевой мантиссе, число считается равным  $\pm \infty$ . А при максимальном порядке, но не нулевой мантиссе возникает NaN (not a number) -

неопределенность. Любая операция с NaN возвращает NaN.

Все операции со специальными значениями совпадают с математическими (с учетом знака, т.е.  $\frac{-1}{+\infty} = -0$ ). Стоит лишь

уточнить, откуда может взяться NaN:

$$+\infty + (-\infty) = \frac{\pm 0}{\pm 0} = \frac{\pm \infty}{\pm \infty} = 0 \cdot (\pm \infty) = \sqrt{x} = NaN \text{ (при } x < 0 \text{)}$$

Также, при нулевом порядке, но не нулевой мантиссе, число считается денормализованным и вместо неявной единицы подразумевается неявный ноль. Это введено для поддержания точности в окрестности нуля.

данные	[младший байт] ... [старший байт]
offset	0                      100500

## Строки

Для (де)кодирования данных необходимо знать как порядок байт, так и кодировку

Примеры кодировок

- ISO-646 - 5 бит, для телеграфов
- BCDIC - 6 бит, перфокарты IBM
- ASCII - 7 бит. Старший (8й) бит использовался как хэш-сумма
- US-ASCII - 8 бит
- ISO-8859 - 8 бит. [0, 127] = ASCII, [128-255] = региональное
- UTF8/16/32 - семейство кодировок, реализующих UNICODE

## Unicode

Стандарт кодирования символов. Описывает инъекцию из множества символов в **N**

Делится на несколько слоев:

1. абстрактный уровень *базовые элементы*
2. coded character set *каждому символу - имя + 16 битное число (code point)*
3. char. encoding forms *code point → code unit (участок памяти фикс. длины)*
4. char. encoding scheme

**BOM** - byte order marker - маркер порядка байтов

Если он находится в начале текста, то по его виду можно сказать, прямой это порядок или обычный

В середине текста трактуется как неразрывный пробел

U+FEFF

Плоскости символов:

0	0	FFFF	базовая многоязычная плоскость
1	10000	1FFFF	доп. многоязычная плоскость
2	20000	2FFFF	доп. иероглифическая плоскость
3	30000	3FFFF	третичная иероглифическая плоскость
4-D	не используется		
E	E0000	EFFFF	доп. плоскость особого назначения
F	F0000	FFFFF	доп. обл. А для частного использования
10	100000	10FFFF	доп. обл. А для частного использования

## UTF

Описывает инъекцию {code point} → {code unit}. В зависимости от цифры меняется размер code unit-а

UTF-8: 1 - 6 байт

UTF-16: 2 байта / суррогатая пара

UTF-32: 4 байта

# Коллекции

## Массив

Множество ячеек, для которой существует биекция на непрерывный участок логического адресного пространства.

Коллекция, содержащая элементы одного размера, такая, что доступ к любому из них выполняется за константу по смещению от логического адреса первого элемента:

data	[first]	[second]	... [100500]
address	B	B+1	B+100499
offset	0	1	100499

## Стэк

Коллекция организованная по принципу первыйПришел - последнийУшел (LIFO)

Удобно представлять как стопку тарелок. Мы можем ложит тарелки сверху, и сверху же их забирать

В памяти это организовано, как сегмент, растущий к началу логического адресного пространства. Указатель на вершину стека (регистр SP).

Такая организация позволяет очень легко организовать две единственных инструкции:

push data	// mov [SP], data; add SP, [SP], 1
pop dest	// mov dest, [SP]; sub SP, [SP], 1

## Очередь

Организована по принципу FIFO, т.е. первыйПришел - последнийУшел

Может использоваться, например, при арбитраже шины данных

## 11. Представление данных в ЭВМ. Форматы данных. BCD. Структуры. Специальные типы данных.

Согласно принципам Фон-Неймана память компьютера однородна, а значит неизвестно, что храниться в конкретных ячейках памяти. Это может быть:

- **Код**
  - CISC
  - RISC
  - VLIW
  - ...
- **Данные**
  - Числа [big endian / little endian]
    - Int [signed(знак / сдвиг) / unsigned]
    - Float [одинарной точности / двойной точности]
    - BCD
    - Упакованные данные
  - Символы [различные кодировки][big endian / little endian]
  - Raw data

Далее, поговорим о данных.

### Двоично-десятичные числа (BCD)

**Двоично-десятичный код** (англ. binary-coded decimal), BCD, 8421-BCD — форма записи рациональных чисел, когда каждый десятичный разряд числа записывается в виде его четырёхбитного двоичного кода.

При помощи четырех бит можно закодировать шестнадцать цифр. Из них используются 10. Остальные 6 комбинаций в двоично-десятичном коде являются запрещенными. Таблица соответствия 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



Двоично-десятичный код также применяется в телефонной связи. В этом случае кроме десятичных цифр кодируются символы '\*' или '#' или любые другие. Для записи этих символов в двоично-десятичном коде используются запрещенные комбинации:

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

Знак хранится как отдельный символ: "+" (1100) для положительных, "-" (1101) для отрицательных. Иногда для положительных место знака "пустое", то есть, там записана последовательность "1111", в таком случае число тоже считается положительным.

В вычислительных машинах нашли применения два формата представления целых десятичных чисел: **упакованный** и **зонный**. В обоих форматах десятичная цифра представляется двоичной тетрадой, то есть заменяется BCD.

**Упакованный формат:**

Байт		Байт			Байт		Байт	
Цифра	Цифра	Цифра	Цифра	...	Цифра	Цифра	Цифра	Знак

**Зонный формат:**

Байт		Байт			Байт		Байт	
Зона	Цифра	Зона	Цифра	...	Зона	Цифра	Знак	Цифра

Наиболее распространён **упакованный формат**, позволяющий не только хранить десятичные числа, но и производить над ними арифметические операции. В данном формате запись числа имеет вид цепочки байтов, где каждый байт содержит коды двух десятичных цифр. Правая тетрада младшего байта предназначается для записи знака числа. Десятичное число должно занимать целое число байт. Если это не выполняется, то четыре старших двоичных разряда левого байта заполняются нулями.

**Зонный формат** распространён, главным образом, в больших универсальных ВМ. В нём под каждую цифру выделяется один байт, где четыре младшие разряда отводятся под код цифры, а в старшую тетраду записывается специальный код — «зона», не совпадающий с кодами цифр и знаков. В IBM 360/370/390 это код 1111. Исключение

составляет байт, содержащий младшую цифру десятичного числа, где в поле зоны хранится знак числа.

**Структуры. Специальные типы данных. Флаги,  
адреса, указатели. Пакеты**

## 12. Методы адресации и использование регистров при адресации. Непосредственная, прямая, регистровая и косвенная регистровая адресация.

Методы адресации - метод определения, откуда брать операнды.

Типы адресаций:

1) Непосредственная - операнд находится в памяти и следует сразу за командой. В данном случае, операнд, как правило, константа.



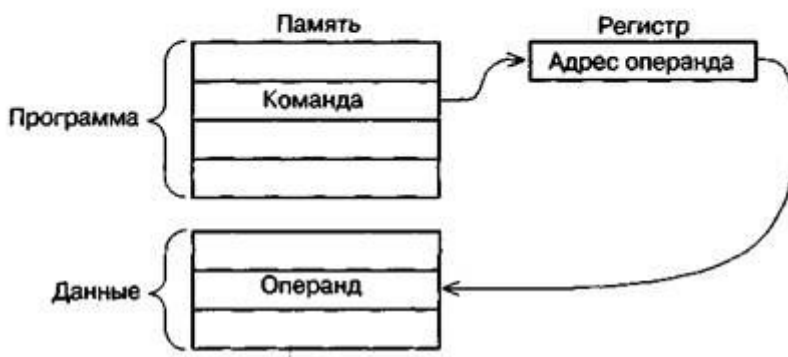
2) Прямая (абсолютная) - за командой лежит код на адрес операнда в памяти



3) Регистровая - операнд лежит в регистре



Рис. 3.3. Регистровая адресация.



4) Косвенная - в регистре лежит адрес на операнд в памяти

(Группы регистров)

Регистры бывают:

- регистры общего назначения
- регистры данных
- регистры адреса
- регистры кодов условий(?)

**Регистры общего назначения** могут использоваться для адресации. По идее, применений много, но в данном случае этого будет достаточно, я думаю.

**Регистры данных** разрешается использовать только для хранения операндов и результатов, но они не принимают участия в вычислении исполнительного адреса.

**Регистры адреса** могут быть в определенной степени универсальными, а могут и предназначаться только для определенного режима адресации. Регистры адреса:

- **Указатель сегмента.** В компьютерах с сегментной организацией оперативной памяти выделяется специальный регистр сегмента, в котором хранится базовый адрес сегмента. Таких регистров может быть несколько, один хранит базовый адрес сегмента операционной системы, а другой (другие) — базовый адрес сегмента текущего процесса.
- **Индексные регистры.** Эти регистры используются в режимах адресации с индексацией, и в некоторых процессорах при обращении к ним автоматически выполняется приращение или уменьшение значения на 1.
- **Указатель стека.** Если в компьютере механизм управления системным стеком программно доступен, то стек размещается в оперативной памяти и выделяется специальный регистр, который содержит текущее значение указателя вершины стека.

Центральный процессор включает в себя устройство управления (УУ) и арифметико-логическое устройство (АЛУ). АЛУ – выполняет сложение, вычитание и другие простые операции над входными данными и помещает результат в выходной регистр. Этот выходной регистр может помещаться обратно в один из регистров. Он может быть сохранен в памяти, если это необходимо. Входные и выходные регистры есть не у всех компьютеров. Регистр-регистр (операции и чтение/запись из регистра). УУ – управляет всеми остальными устройствами ЭВМ путём посылки управляющих сигналов, подчиняясь которым остальные устройства производят определённые действия, предписанные этими сигналами. УУ имеет собственные регистры (счетчик команд, регистр команд и т.д.). В регистрах хранятся, в том числе, команды для исполнения процессором.

Типы команд: Регистр-память – вызывают слова из памяти, помещают их в регистры, где они используются в качестве входных данных АЛУ, либо помещают регистры обратно в память.

Регистр-регистр – вызывают два операнда из регистров, помещают их во входные регистры АЛУ, выполняют над ними какую-нибудь арифметическую или логическую операцию и переносят результат обратно в один из регистров.

1) Команды перемещения данных Копирование данных из одного места в другое — одна из самых распространенных операций. Под копированием мы понимаем создание нового объекта с точно таким же набором битов, как у исходного объекта.

2) Бинарные операции Бинарные операции берут два операнда и получают из них результат. Все архитектуры команд содержат команды для сложения и вычитания целых чисел. Команды умножения и деления целых чисел также имеются практически во всех случаях. Следующая группа бинарных операций содержит булевы команды. Существует 16 булевых функций от двух переменных. Обычно присутствуют И, ИЛИ и НЕ.

3) Унарные операции Унарные операции используют один операнд и производят один результат. Сдвиг, циклический сдвиг, прибавление 1.

4) Сравнения и условные переходы Практически все программы должны проверять свои данные и на основе результатов изменять последовательность команд, которые нужно выполнить. Команды условного перехода проверяют какое-либо условие и совершают переход в определенный адрес памяти, если условие выполнено. Сравнение слов, сравнение с 0, и т.д.

5) Команды вызова процедур Процедура — это группа команд, которая выполняет определенную задачу и которую можно вызвать из нескольких мест программы.

6) Управление циклом Часто возникает необходимость выполнять некоторую группу команд фиксированное количество раз, поэтому некоторые машины содержат команды для облегчения этого процесса. Все эти схемы содержат счетчик, который увеличивается или уменьшается на какую-либо константу каждый раз при выполнении цикла. Кроме того, этот счетчик каждый раз проверяется. При выполнении определенного условия цикл завершается.

#### 7) Команды ввода-вывода

Производительность – скорость, с которой процессор производит операции. Тактовый генератор – источник импульсных сигналов в компьютере, который синхронизирует каждую операцию процессора. Все импульсы одинаковы по длительности. Интервалы между последовательными импульсами также одинаковы. Временной интервал между началом одного импульса и началом следующего называется временем такта. Тактовая частота – скорость, с которой микропроцессор выполняет команды. Центральному процессору (CPU) необходимо фиксированное количество тактов системных часов (или тактовых циклов) для выполнения каждой команды. Чем быстрее часы, тем больше команд процессор исполняет в секунду. MIPS (Millions of Instructions per Second – «миллионы команд в секунду») – величина, которая показывает сколько миллионов инструкций в секунду выполняет процессор. Тактовая частота выражается в мегагерцах(MHz),  $1 \text{ MHz} = 1 \text{ миллиону циклов в секунду}$ . Характеризует число основных операций компьютера, производимых в секунду, измеряется в герцах. Различные шины и чипы на материнской плате компьютера могут иметь различную тактовую частоту.

### **13. Методы адресации и использование регистров при адресации. Индексная и относительная индексная адресация. Использование стека при адресации.**

**Индексная адресация**  $ADD\ R1, \ x(R2)$  где  $x$  фиксированный адрес а в  $R2$  смещение. Часто нужно уметь обращаться к словам памяти по известному смещению. Подобные примеры мы видели в машине IJVM, где локальные переменные определяются по смещению от регистра LV. Обращение к памяти по регистру и константе смещения называется индексной адресацией. В машине IJVM при доступе к локальной переменной используется указатель ячейки памяти (LV) в регистре плюс небольшое смещение в самой команде. Есть и другой способ: указатель ячейки памяти в команде и небольшое смещение в регистре.

**Относительная индексная адресация.** В некоторых машинах применяется режим адресации, при котором адрес вычисляется путем суммирования значений двух регистров и смещения (смещение факультативно). Такой режим называется относительной индексной адресацией. Один из регистров — это база, другой — индекс.

**Стековая адресация** Мы уже отмечали, что очень желательно сделать машинные команды как можно короче. Предельный случай — команды без адресов. Безадресные команды, например IADD, возможны при наличии стека.

**IA-32:** В архитектуре IA-32 на программно-аппаратном уровне поддерживается такая структура, как стек. Для работы со стеком существуют специальные регистры. Регистр указателя стека (Stack Pointer register) ESP / SP — содержит указатель на вершину стека в текущем сегменте стека. Регистр указателя базы стека (Base Pointer register) EBP / BP — применяется для организации произвольного доступа к стеку. Процессоры Intel аппаратно поддерживают сегментную организацию программы. При обращении к данным или стеку неявно используется информация из сегментных регистров. В модели IA-32 имеется шесть 16-битных сегментных регистров: CS, DS, SS, ES, FS, GS. В модели IA-32 поддерживается четыре типа сегментов:

- 1) Сегмент кода — содержит команды программы. Для доступа используется регистр сегмента кода (Code Segment register) CS — содержит адрес сегмента с машинными командами, к которому имеет доступ процессор.
- 2) Сегмент данных — содержит обрабатываемые программой данные. Для доступа используется регистр сегмента данных (Data Segment register) DS — содержит адрес сегмента с данными текущей программы.
- 3) Сегмент стека — представляет собой область памяти, называемую стеком (LIFO — Last-In-First-Out). Для доступа используется регистр сегмента стека (Stack Segment register) SS — содержит адрес сегмента стека текущей программы.
- 4) Дополнительный сегмент данных — неявно предполагается, что обрабатываемые данные находятся в сегменте данных. Но можно задействовать до трех дополнительных сегментов. Для доступа используется регистры дополнительного сегмента данных (Extension Data Segment



register) ES, FS, GS.

## **14. Методы адресации и использование регистров при адресации. Представление адреса в командах перехода. Представление адреса с использованием сегментных регистров. Примеры команд перехода в x86.**

**Команда перехода** – команда передачи управления в другое место.

Представление адреса для них зависит от архитектуры.

(\*) Например в SPARC реализована схема динамического прогнозирования направления ветвлений программы, основанная на двухбитовой истории переходов и обеспечивающая ускоренную обработку команд условного перехода.

Для удобства, будем обсуждать команды архитектуры x86, попутно затрагивая их методы адресации.

### **Безусловный переход**

Команда безусловного перехода имеет следующий синтаксис:

JMP <операнд>

*Операнд* указывает адрес перехода. Существует два способа указания этого адреса, соответственно различают *прямой* и *косвенный* переходы.

### **Прямой переход**

Если в команде перехода указывается метка команды, на которую надо перейти, то переход называется *прямым*.

Пример:

jmp L

...

L: mov eax, x

Вообще, любой переход заключается в изменении адреса следующей исполняемой команды, т.е. в изменении значения регистра EIP. Казалось

бы, в команде перехода должен задаваться именно адрес перехода. Однако в команде прямого перехода задаётся не абсолютный адрес, а разность между адресом перехода и адресом команды перехода. Действие команды перехода заключается в прибавлении этой величины к текущему значению регистра EIP. Операнд команды перехода рассматривается как поле со знаком, поэтому при сложении его со значением регистра EIP значение в этом регистре может как увеличиться, так и уменьшиться, т.е. возможен переход и вперёд, и назад.

Запись в команде перехода не абсолютного, а относительного адреса перехода позволяет уменьшить размер команды перехода. Абсолютный адрес должен быть 32-битным, а относительный может быть и 8-битным, и 16-битным.

## Косвенный переход

При *косвенном* переходе в команде перехода указывается не адрес перехода, а регистр или ячейка памяти, где этот адрес находится. Содержимое указанного регистра или ячейки памяти рассматривается как абсолютный адрес перехода. Косвенные переходы используются в тех случаях, когда адрес перехода становится известен только во время работы программы.

Пример:

```
jmp ebx
```

## Команды сравнения и условного перехода

Команды условного перехода осуществляют переход, который выполняется только в случае истинности некоторого условия. Истинность условия проверяется по значениям флагов. Поэтому обычно непосредственно перед командой условного перехода ставится команда *сравнения*, которая формирует значения флагов:

```
CMR <операнд1>, <операнд2>
```

Команда сравнения эквивалентна команде SUB за исключением того, что вычисленная разность никуда не заносится. Назначение команды CMP – установка и сброс флагов. Что касается команд условного перехода, то их достаточно много, но все они записываются единообразно:

Jxx <метка>

Все команды условного перехода можно разделить на три группы.

В **первую** группу входят команды, которые обычно ставятся после команды сравнения. В их мнемокодах указывается тот результат сравнения, при котором надо делать переход.

**Важно:** не надо учить всю таблицу наизусть! Достаточно просто понять принцип, по которому образуется имя команды(семантический), и уметь привести пару примеров, если Пьянзин попросит.

Мнемокод	Название	Условие перехода после команды CMP op <sub>1</sub> , op <sub>2</sub>	Значения флагов	Примечание
JE	Переход если равно	op <sub>1</sub> = op <sub>2</sub>	ZF = 1	Для всех чисел
JNE	Переход если не равно	op <sub>1</sub> ≠ op <sub>2</sub>	ZF = 0	
JL/JNGE	Переход если меньше	op <sub>1</sub> < op <sub>2</sub>	SF ≠ OF	Для чисел со знаком
JLE/JNG	Переход если меньше или равно	op <sub>1</sub> ≤ op <sub>2</sub>	SF ≠ OF или ZF = 1	
JG/JNLE	Переход если больше	op <sub>1</sub> > op <sub>2</sub>	SF = OF и ZF = 0	
JGE/JNL	Переход если больше или равно	op <sub>1</sub> ≥ op <sub>2</sub>	SF = OF	
JB/JNAE	Переход если ниже	op <sub>1</sub> < op <sub>2</sub>	CF = 1	Для чисел без знака
JBE/JNA	Переход если ниже или равно	op <sub>1</sub> ≤ op <sub>2</sub>	CF = 1 или ZF = 1	
JA/JNBE	Переход если выше	op <sub>1</sub> > op <sub>2</sub>	CF = 0 и ZF = 0	
JAЕ/JNB	Переход если выше или равно	op <sub>1</sub> ≥ op <sub>2</sub>	CF = 0	

Пример(нахождение максимума из двух чисел):

```

mov eax, x
cmp eax, y
jge L
mov eax, y
L: mov z, eax

```

Во **вторую** группу команд условного перехода входят те, которые обычно ставятся после команд, отличных от команды сравнения, и которые реагируют на то или иное значение какого-либо флага.

Мнемокод	Условие перехода	Мнемокод	Условие перехода
JZ	ZF = 1	JNZ	ZF = 0
JS	SF = 1	JNS	SF = 0
JC	CF = 1	JNC	CF = 0
JO	OF = 1	JNO	OF = 0
JP	PF = 1	JNP	PF = 0

Рассмотрим пример: пусть a, b и c – беззнаковые переменные размером 1 байт, требуется вычислить  $c = a * a + b$ , но если результат превосходит размер байта, передать управление на метку *ERROR*.

```

mov al, a
mul al
jc ERROR
add al, b
jc ERROR
mov c, al

```

И, наконец, в **третью** группу входят две команды условного перехода, проверяющие не флаги, а значение регистра ECX или CX:

JCXZ <метка> ; Переход, если значение регистра CX равно 0

JECXZ <метка> ; Переход, если значение регистра ECX равно 0

Однако эта команда выполняется достаточно долго. Выгоднее провести сравнение с нулём и использовать обычную команду условного перехода.

## Представление адреса с использованием сегментных регистров

Физически **сегмент** представляет собой область памяти, занятую командами и (или) данными, адреса которых вычисляются относительно значения в соответствующем сегментном регистре.

Каждая программа содержит 3 типа сегментов:

- Сегмент кодов – содержит машинные команды для выполнения. Обычно первая выполняемая команда находится в начале этого сегмента, и операционная система передает управление по адресу данного сегмента для выполнения программы. Регистр сегмента кодов (CS) адресует данный сегмент.
- Сегмент данных – содержит данные, константы и рабочие области, необходимые программе. Регистр сегмента данных (DS) адресует данный сегмент.

Эти регистры комбинируются с указателем команд для получения ссылки на текущую команду. Выборка команды происходит из памяти по адресу, заданному парой регистров сегментный:IP.

Комбинация сегментного регистра с регистром смещения, порождающая физический адрес, записывается как сегмент : смещение, например, CS:IP. Значение сегмента предшествует двоеточию, значение смещения следует за ним. Такая запись используется как для регистров, так и для абсолютных величин. Вы можете записывать адреса как CS:100, так и DS:BX.

- Сегмент стека - содержит адреса возврата как для программы (для возврата в операционную систему), так и для вызовов подпрограмм (для возврата в главную программу), а также используется для передачи параметров в процедуры. Регистр сегмента стека (SS) адресует данный сегмент. Команды PUSH, POP, CALL и RET обрабатывают данные в стеке, находящиеся в месте, определяемом парой регистров SS:SP.

## 15. Три основные архитектуры организации кэша

*Ключевые слова:* Прямое отображения, полностью ассоциативный, ассоциативный по множеству

Начнём с базовых определений. Для того, чтобы упростить общение с оперативной памятью, кэш-контроллер оперирует не байтами, а блоками данных, которые соответствуют *размеру пакетного цикла чтения или записи*. Программно кэш-память представляется как совокупность блоков данных фиксированного размера. Каждый такой блок называется **кэш-строкой** (или кэш-линейкой).

Кэш-строка полностью заполняется за один пакетный цикл чтения. Если процессор обратился к одному байту памяти, кэш-контроллер все равно инициирует полный цикл обращения к основной памяти и запрашивает весь блок целиком. Техническая деталь: адрес первого байта кэш-строки кратен размеру пакетного цикла обмена.

Объем кэша много меньше объема оперативной памяти, поэтому каждой кэш-строке соответствует множество ячеек кэшируемой памяти. Ввиду этого, нужно сохранять не только содержимое ячейки, которая кэшируется, но еще и ее адрес. Поэтому каждая кэш-строка имеет специальное поле - **тэг**. В тэге хранится линейный и (или) физический адрес первого байта кэш-строки. Отсюда и вывод, что **кэш-память - ассоциативная**.

Допустим, что теперь речь зашла о практике. Процессор обращается к ячейке с некоторым адресом. Сначала всегда сработает кэш-контроллер: есть ли данные соответствующие этому адресу в кэш-памяти? Формально, мы ищем нужный тэг.

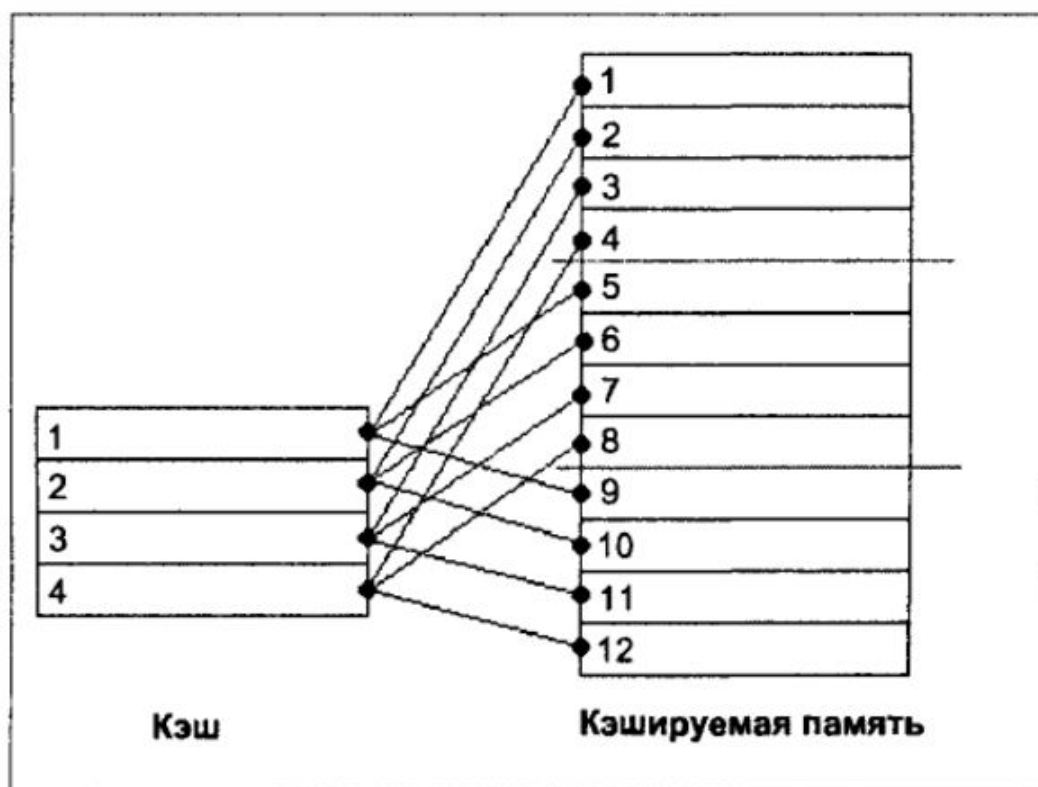
Просмотр тэгов может быть либо **параллельным**, либо **последовательным**. В первом случае получаем сложное и дорогое устройство. А если последовательный, то получаем, что мы просто

просматриваем все тэги каждой кэш-строк. Медленно, поэтому так не делается. Существуют несколько решений такой проблемы.

## Кэш прямого отображения

Проблема поиска решается так: пусть каждая ячейка памяти соответствует одной, строго определенной кэш-строке. Тогда как каждая кэш-строка соответствует множеству строго определенных ячеек кэшируемой памяти.

Здесь и всюду далее фигурирует такое выражение, как кэшируемая ячейка. На рисунках изображено множество таких ячеек, потому что алгоритмы выбора того, какие ячейки кэшировать собирают именно множество, сама выборка ячеек определяются посредством *сложных алгоритмов: загрузка по требованию, спекулятивная загрузка.*



Если представить, что кэш состоит из 4 кэш-строк (как на рисунке), то получим следующую связку (указана стрелками на рисунке): первая строка кэша - с первой ячейкой кэшируемой памяти, а потом уже с пятой и девятой. Тогда можно выделить следующую формулу:

$$N = (\text{Address} / \text{Cache line size}) \% (\text{Cache size} / \text{Cache line size})$$

N - номер кэш линейки.

Address - адрес ячейки кэшируемой памяти.

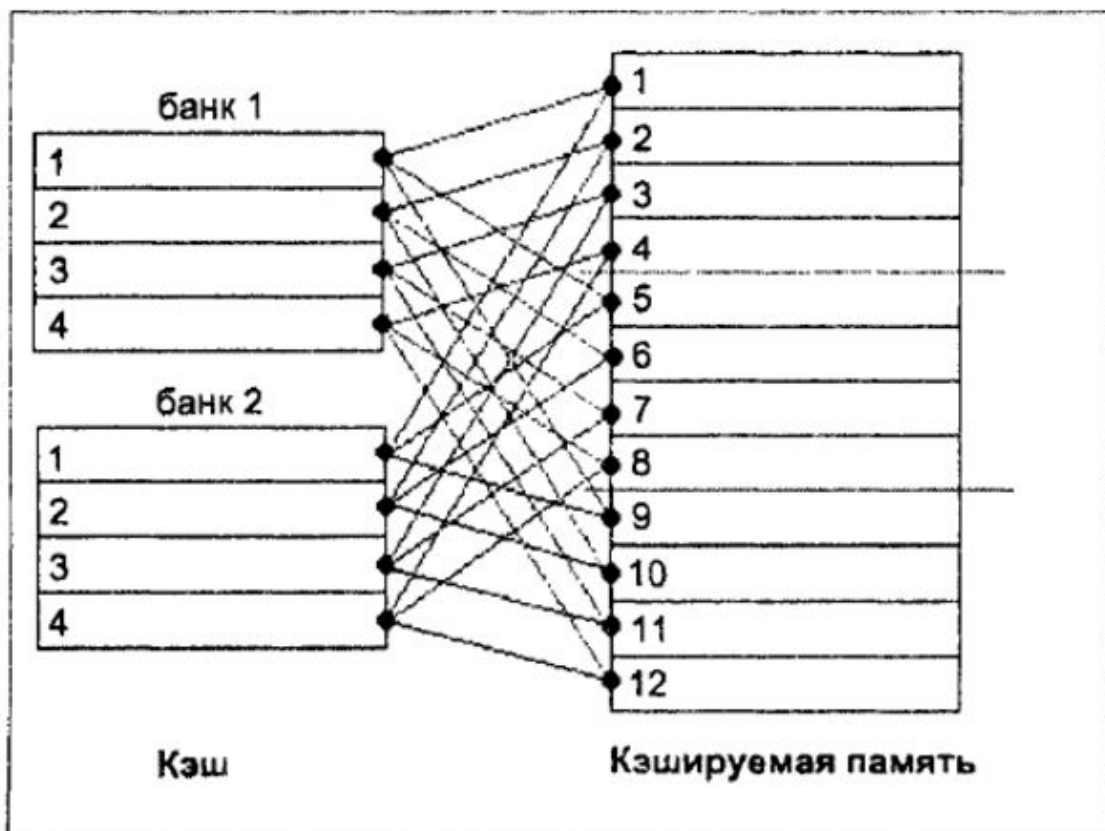
Cache line size - длина кэш-строки в байтах

Cache size - размер кэша в байтах

Но и тут есть проблема. Если процессор обратится к 1, 4 и 9 ячейке, то, несмотря на то, что в кэше куча свободных кэш-строк, они все равно будут пустыми, так как очередная ячейка будет замещать ту, что уже помещен туда. Эта ситуация называется работой кэша **вхолостую** (trashing).

**Кэш, ассоциативный по множеству (или, что часто встречается в книгах наборно-ассоциативный кэш).**

Является улучшением кэша прямого отображения. Состоит из набора банков; каждый банк представляет собой кэш прямого отображения.



У нас есть банки, каждый из которых независим от другого. На рисунке их два. Тогда, если процессор запросит 1 кэшируемую ячейку, то она окажется в 1 банке, а 5 уже во 2.



Количество банков кэша называется **ассоциативностью**. Видно, что при росте количества банков эффективность кэша возрастает.

Большинство современных компьютеров используют кэш с 8 или 16 банками.

### **Полностью Ассоциативный кэш**

В идеале, при наивысшей степени дробления в каждом банке будет только одна кэш-строка, и тогда любая ячейка кэшируемой памяти будет сохранена в любой строке кэш-памяти.

**Полезная ссылка:** <http://iproc.ru/parallel-programming/lection-7/>

## 16. Кэш. Типы кэш-памяти по стратегии обновления основной памяти. Механизмы замещения строк. Организация кэш-памяти в современных ЭВМ.

Пространство памяти отображения данных в кэше разбивается на **строки** - блоки фиксированной длины (например, 32, 64 или 128 байт). Каждая строка кэша может содержать непрерывный выровненный блок байт из оперативной памяти. Какой именно блок оперативной памяти отображен на данную строку кэша, определяется тегом строки и алгоритмом отображения (предыдущий билет). С каждой строкой кэша связано поле тэга. **Тэг** (tag – пометка, отметка) – признак, используемый при поиске требуемых данных.

**Механизм сохранения информации в кэш-памяти.** При включении микропроцессора в работу вся информация в его кэш-памяти недостоверна. При обращении к памяти микропроцессор сначала проверяет, не содержится ли искомая информация в кэш-памяти. Для этого сформированный им физический адрес сравнивается с адресами ячеек памяти, которые были ранее кэшированы из ОЗУ в КП. При первом обращении такой информации в кэш-памяти, естественно, нет, и это соответствует **кэш-промаху**. Тогда микропроцессор проводит обращение к оперативной памяти, извлекает нужную информацию, использует ее в своей работе, но одновременно записывает эту информацию в кэш. Если бы в кэш-память заносилась только востребованная микропроцессором в данный момент информация, то, скорее всего, при следующем обращении вновь произошел бы кэш-промах: вряд ли следующее обращение произойдет к той же самой команде или к тому же самому операнду.

**Кэш-попадания** (присутствие запрашиваемой информации в кэше) происходили бы лишь после того, как в КП накопится достаточно большой фрагмент программы, содержащий некоторые циклические участки кода, или фрагмент данных, подлежащих повторной обработке. В настоящее время существует тенденция к использованию разделенной кэш-памяти (split cache), когда команды хранятся в одной кэш-памяти, а данные — в другой. Такая архитектура также называется гарвардской (**Harvard architecture**), поскольку идея использования отдельной памяти для команд и отдельной памяти для данных впервые воплотилась в компьютере Mark III, который был создан Говардом Айкеном (Howard Aiken) в Гарварде. Современные разработчики пошли по этому пути, поскольку сейчас широко распространены конвейерные архитектуры, а при конвейерной организации должна быть возможность одновременного доступа и к командам, и к данным (операндам). Разделенная кэш-память позволяет осуществлять параллельный доступ, а общая — нет. К тому же, поскольку команды обычно не меняются во время выполнения программы, содержание кэша команд не приходится записывать обратно в основную память. Между разделенной кэш-памятью и основной памятью часто помещается кэш-память второго уровня. Как правило, все содержимое кэш-памяти первого уровня находится в кэш-памяти второго уровня, а все содержимое кэш-памяти второго уровня — в кэш-памяти третьего уровня.

Кэш-контроллер обязан обеспечивать **коггерентность (coherency)**- согласованность кэш-памяти с основной памятью. Допустим, к некоторой ячейке

памяти, уже модифицированной в кэше, но еще не выгруженной в основную память, обращается периферийное устройство (или другой процессор) - кэш-контроллер должен немедленно обновить ОП, иначе оттуда прочтутся "старые" данные.

Стратегии обновления ОП:

1. метод сквозной записи (Write Through – WT): обновляется слово, хранящееся в основной памяти. Если в кэш-памяти существует копия этого слова, то она также обновляется. Если же в кэш-памяти отсутствует копия этого слова, то либо из основной памяти в кэш-память пересылается строка, содержащая это слово (метод WTWA (write thru with allocation)– сквозная запись с распределением), либо этого не допускается (метод WTNWA (write thru not with allocation)– сквозная запись без распределения). При исполнении любой из модификаций при методе сквозной записи нет необходимости копировать удаляемый из кэш-памяти блок в ОП, так как его копия в ОП поддерживается актуальной.
2. метод обратной записи (Write Back – WB): запись данных производится в кэш. Запись же в основную память производится позже (при вытеснении или по истечению времени). Модификация содержимого блока в кэш-памяти при обращении по записи не является причиной изменения копии этого блока в ОП. При обращении по записи к блоку, отсутствующему в кэш-памяти, обязательно осуществляется пересылка блока из ОП в кэш-память с последующей его модификацией только в кэш-памяти. Метод WB требует копирования блока из кэш-памяти в ОП только в момент удаления блока из кэш-памяти, то есть когда этот блок становится кандидатом на удаление.

Стратегия WT требует больших временных затрат, однако содержимое блоков ОП полностью согласовано с содержимым блоков кэш-памяти. Стратегия WB требует значительно меньших затрат времени на реализацию, однако, на какое-то время копии некоторых блоков в ОП становятся не актуальными по сравнению с их модификациями в кэш-памяти. Однако, к этим блокам может потребоваться обращение со стороны устройств ввода/вывода, поэтому при использовании двухуровневой кэш-памяти для первого уровня обычно используется стратегия WT, а для второго уровня – WB.

Способы выбора позиции записи (стратегии замещения):

1. Random - не предполагается выполнения анализа предыстории блоков, находящихся в кэш-памяти. Согласно стратегии RAND блок – кандидат на удаление – выбирается случайным образом (т.е. удален может быть любой из допустимого множества блоков кэш-памяти). Реализация принципа случайного выбора может быть осуществлена с помощью счетчика, содержимое которого инкрементируется с каким-либо периодом. Частным случаем такого счетчика может быть системный таймер. Значение в счетчике, в принципе, является случайным по отношению к процессу замещения блоков в кэш-памяти и может использоваться как номер блока – кандидата на удаление.

2. FIFO - первым пришёл – первым вышел: среди всех строк, являющихся объектами замещения, выбирается та, которая самой первой была переслана в кэш-память. Физическая реализация этой стратегии может быть осуществлена с использованием принципа циклического буфера, в котором блоки выстраиваются в порядке их поступления в кэш. Блок – кандидат на удаление выбирается из вершины буфера.
3. LFU - Least Frequency Used - удалению из допустимого множества блоков кэш-памяти подлежит блок с наименьшей частотой обращений. Для реализации этого принципа необходимо каждый блок кэш-памяти снабдить счетчиком обращений, значение которого инкрементируется при обращении к блоку. Тогда при необходимости записать новый блок из ОП на место одного из блоков кэш-памяти анализируются значения счетчиков всех блоков из допустимого множества. Удалению подлежит блок с наименьшим значением счетчика обращений. Процесс выбора блока – кандидата на удаление завершается обнулением (сбросом) счетчиков обращений всех блоков кэш-памяти.
4. LRU - Least Recently Used- выбирается строка, к которой наиболее долгое время не было обращений. Общая реализация этого метода требует сохранения «бита возраста» для строк кэша и за счет этого происходит отслеживание наименее использованных строк (то есть за счет сравнения таких битов). В подобной реализации, при каждом обращении к строке кэша меняется «возраст» всех остальных строк.

В общем случае современный кэш на каждом уровне состоит из трех семантических блоков (конкретнее рассмотрим на МП i486):

1. Блок данных: содержит 8 Кбайт данных и команд. Он разделен на 4 массива (направления), каждый из которых состоит из 128 строк. Строка содержит данные из 16 последовательных адресов памяти начиная с адреса, кратного 16. Индекс массивов блока данных, состоящий из 7 бит, соответствует 4 строкам КП, по одной из каждого массива. Четыре строки КП с одним и тем же индексом называются множеством.
2. Блок тегов: имеется один тег длиной 21 бит для каждой строки данных в КП. Блок тегов также разделен на 4 массива по 128 тегов. Тег содержит старшие 21 бит физического адреса данных, находящихся в соответствующей строке КП.
3. Блок состояния (блок достоверности и LRU ): содержится по одному 7-разрядному значению для каждого из 128 множеств строк КП: 4 бита достоверности ( V ) по одному на каждую строку множества и 3 бита ( B0 ... B2 ), управляющие механизмом LRU. Биты достоверности показывают, содержит ли строка достоверные ( V = 1 ) или недостоверные ( V = 0 ) данные. При программной очистке КП и аппаратном сбросе процессора все биты достоверности сбрасываются в 0.

### **Многоуровневая память (multilevel memory):**

Организация памяти, состоящая из нескольких уровней запоминающих устройств с различными характеристиками и рассматриваемая со стороны

пользователей как единое целое. Для многоуровневой памяти характерна страничная организация, обеспечивающая «прозрачность» обмена данными между ЗУ разных уровней.

Современные технологии позволяют разместить КЭШ-память и ЦП на общем кристалле. Такая внутренняя КЭШ-память строится по технологии статического ОЗУ и является наиболее быстродействующей.

Емкость ее обычно не превышает 64 Кбайт. Попытки увеличения емкости обычно приводят к снижению быстродействия, главным образом, из-за усложнения схем управления и дешифрации адреса.

Общую емкость КЭШ-памяти ЭВМ увеличивают за счет второй (внешней) КЭШ-памяти, расположенной между внутренней КЭШ-памятью и ОЗУ. Такая система известна под названием двухуровневой, где внутренней КЭШ-памяти отводится роль первого уровня (L1), а внешней - второго уровня (L2). Емкость L2 может быть значительной (до 1 МБ).

При доступе к памяти ЦП сначала обращается к КЭШ-памяти первого уровня. В случае промаха производится обращение к КЭШ-памяти второго уровня. Если информация отсутствует и в L2, выполняется обращение к ОЗУ и соответствующий блок заносится сначала в L2, а затем и в L1. Благодаря такой процедуре часто запрашиваемая информация может быть быстро восстановлена из КЭШ-памяти второго уровня. Для ускорения обмена информацией между ЦП и L2 между ними часто вводят специальную шину, так называемую шину заднего плана, в отличие от шины переднего плана, связывающую ЦП с основной памятью.

Количество уровней КЭШ-памяти не ограничивается двумя. В некоторых ЭВМ можно встретить КЭШ-память третьего уровня (L3). Характер взаимодействия очередного уровня с предшествующим аналогичен описанному для L1 и L2. Таким образом, можно говорить об иерархии КЭШ-памяти. Каждый последующий уровень характеризуется большей емкостью, меньшей стоимостью, но и меньшим быстродействием, хотя оно все же выше, чем у ЗУ основной памяти.

Все современные процессоры имеют как минимум двухуровневую структуру кэшпамяти, а большинство процессоров Intel — трехуровневую кэшпамять.

Причем в случае процессоров Intel кэши всех уровней размещены на кристалле процессора.

**Многовходовая память** (*multiport storage memory*) — устройство памяти, допускающее независимое обращение с нескольких направлений (входов - процессоров), причём обслуживание запросов производится в порядке их приоритета.

## 17. Архитектура с общей шиной. Децентрализованный арбитраж.

**Шина** - провода по которым происходит обмен информацией между устройствами.

**Архитектура с общей шиной** - архитектура, при которой все устройства (память, процессор, устройства ввода/вывода) общаются посредством одного канала связи.

Устройства подключаются к шине через **контроллер устройства**.

Контроллер реализует всю сложную логику взаимодействия с шиной.

Так как обмен информацией производится по шине с помощью электрических сигналов, то в каждый момент времени только два устройства могут выполнять такой обмен. Обычно одно из этих устройств является **задатчиком** (инициатором обмена данными), а другое – **исполнителем** (ведомым).

При использовании общей шины несколькими устройствами могут возникать конфликты - **коллизии**, когда много устройств хотят одновременно обмениваться данными (хотят стать **задатчиками шины**). Чтобы предотвратить хаос, который может при этом возникнуть, нужен специальный механизм — так называемый **арбитраж шины**.

Ясно, что этот самый арбитраж можно производить по разным алгоритмам. В данном билете мы подробно осветим **децентрализованный арбитраж**. Способов проводить децентрализованный арбитраж тоже несколько. Есть децентрализованный арбитраж с **последовательным** и **параллельным подключением**. Далее я опишу децентрализованный арбитраж с параллельным подключением.

Но сначала небольшой ликбез про шины:

Шина, хоть и разделяемая, - это не один провод. Их там несколько, просто они используются все вместе. Некоторые из них служат непосредственно для передачи данных. Другие - для служебных целей. Часто эти провода или наборы проводов называются **линиями**.

### Децентрализованный арбитраж:

В составе шины имеется группа арбитражных линий. Она нужна для определения **задатчика**, если есть несколько желающих. Эта группа реализована по схеме “монтажного или”. Схема “монтажного или” позволяет каждому устройству выставлять на общую шину свой номер и при этом каждое другое устройство будет видеть все выставленные на данный момент номера. У каждого устройства есть свой уникальный приоритет (обычно устройств немного, я нашел цифру 4-7). Когда устройство хочет передавать данные - оно выставляет на группу арбитражных линий

свой приоритет. Если устройство видит, что на группе арбитражных линий выставлено устройство с приоритетом старше - оно снимает свой приоритет. Когда на арбитражных линиях остаётся только одно устройство - это устройство понимает что ему можно передавать данные, теперь оно - **здатчик**.

После того как устройство получило доступ к шине - оно ставит в линию busy соответствующее значение, сигнализирующее о том, что данные передаются. Пока они передаются - никто другой не может получить доступ к шине. В связи с этим, вводятся ограничения на время владение шиной.

Существуют также схемы **централизованного арбитража**, когда существует отдельный контроллер, производящий управление доступом к шине. Там тоже будут последовательное и параллельное подключение. И там я уже расскажу про последовательное.

Децентрализованный арбитраж реализуется сложнее централизованного, но при этом является более отказоустойчивым т.к. ошибка в одном устройстве не влияет на всю систему (если только из-за этой ошибки устройство не начинает засорять общий канал).

## 18. Архитектура с общей шиной. Централизованный арбитраж. Структура приоритетов при централизованном арбитраже.

Архитектура с общей шиной - в 17 билете.

**Централизованный арбитраж** - способ управления доступом к общей шине, при котором существует отдельное устройство, называемое арбитром шины, которое регулирует передачу информации.

### Логика работы централизованного арбитража:

У шины есть 2 служебных линии - **линия запроса** и **линия предоставления**. Разрешение конфликтов производится по принципу **приоритетов устройств**: при конфликте арбитром отдается предпочтение устройству с большим приоритетом.

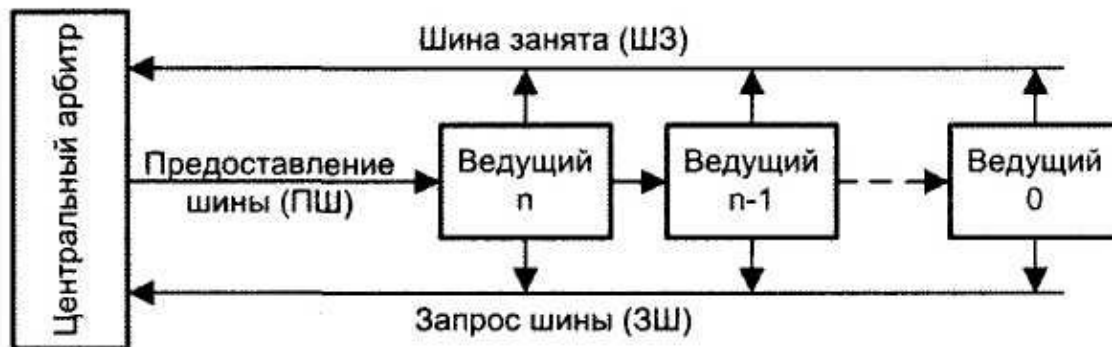
1. Сначала **здатчик** делает так называемый запрос шины, т.е. посылает **арбитру** сигнал о желании начать обмен данными.  
Этот сигнал выставляется на **линию запроса**.
2. Если шина занята, то устройство вынуждено ждать ее освобождения, а если шина свободна, то она сигнализирует об этом через **линию предоставления**. Если устройство свободно - **здатчик** получает ее в монопольное пользование. Это означает, что для остальных устройств шина теперь будет выдавать признак занятости.
3. После захвата шины **здатчик** определяет, готов ли **исполнитель** для обмена данными. Для этого он посылает исполнителю специальный сигнал и ждет ответа.
4. Определив готовность исполнителя, **здатчик** начинает обмен данными.
5. Закончив обмен данными, **здатчик** производит освобождение шины. На этом операция обмена данными между двумя устройствами по общей шине считается завершенной.

Существуют схемы **последовательного** и **параллельного** арбитража.

При последовательном что устройства подключаются к шине [последовательно](#). В этом случае легко реализуется логика приоритетов - чем ближе устройство к шине, тем выше приоритет. Однако, приоритеты в таком случае являются статическими.

На картинке ниже схематично изображена схема подключения устройств при последовательном арбитраже. Видимо, по линии **предоставление шины** на картинке как раз идут сами данные.





В некоторых шинах устраивается несколько уровней приоритета. На каждом уровне приоритета есть своя линия запроса шины и линия предоставления шины. Каждое устройство связано с одним из уровней запроса шины. Если одновременно запрашивается несколько уровней приоритета, арбитр предоставляет шину самому высокому уровню. Среди устройств одинакового приоритета используется система **последовательного опроса**. Если шина занята и происходит запрос на захват шины, то арбитр не отключает устройство от шины а “рекомендует” ему максимально быстро закончить выполнение своей задачи.

Обычно у устройств ввода самый высокий приоритет, затем устройства вывода, затем устройства чтения.

В 17 и 18 билете я писал про последовательный и параллельный арбитраж, но в 17 рассказал только про параллельный, а в 18 только про последовательный.

Если вам интересно как работает параллельный централизованный или последовательный децентрализованный арбитраж - можно почитать об этом [здесь](#).

Ну а я про эти два способа напишу основные преимущества и недостатки:

**Последовательный арбитраж** легко реализуется.

Во-первых, у меня есть предположение, что с физической точки зрения легко реализуются эти приоритеты, когда устройства расположены последовательно в порядке их убывания.

Во-вторых, арбитражу шины легко - у него всего 3 линии.

При **параллельном арбитраже** приходится выделять по отдельной линии на каждое устройство. Это ограничивает возможности для подключения новых устройств. Также, это увеличивает стоимость шины (потому что больше линий нужно), а ещё усложняет логику её работы. С другой стороны, это позволяет устанавливать приоритеты устройств как угодно. В том числе можно менять приоритеты устройств прямо во время работы. Также эта схема кажется ещё и более устойчивой к поломкам, ведь раз у каждого устройства своя выделенная линия - устройства не мешают друг другу.

## **19. Архитектура с общей шиной. Механизмы обмена данными.**

Архитектура с общей шиной - см 17.

Существует два основных механизма обмена данными по системной шине:

1) Прямой доступ к памяти - DMA - способ обмена данными между устройствами и памятью без участия процессора. Для этого существует специальное устройство - DMA контроллер - посредством которого происходит весь процесс обмена. У этого устройства есть регистры, в которые процессор может писать и считывать данные.

2) Программный ввод/вывод - PIO - способ обмена данными, в котором вся информация проходит через процессор. Для осуществления взаимодействия с устройствами процессор может читать, записывать информацию в специальные регистры контроллера устройства. Существует пара способов реализации доступа к этим регистрам:

\* порты ввода/вывода - перед началом работы каждому регистру устройства назначается один из портов ввода вывода. В дальнейшем процессор может получить доступ к этому регистру, обратившись по адресу конкретного порта: OUT PORT, REG; запись в регистр процессора содержимого порта PORT

\* отображение всех управляющих регистров на общее адресное пространство. Эти адреса исключаются операционной системой из адресного пространства. Так осуществляется защита управляющих регистров от пользовательских процессов. Удобное обобщение управляющих регистров на общее адресное пространство все таки вызывает ряд проблем:

1) Кэширование - нужно запрещать кэшировать адресное пространство управляющих регистров

2) еще какие то, я не понял =( LOL

## 20. Организация конвейера команд. Скалярный, суперскалярный и суперконвейерный вычислитель.

Выполнение каждой команды складывается из ряда последовательных этапов, суть которых не меняется от команды к команде. Чтобы увеличить быстродействие процессора и максимально использовать его возможности, используется конвейерный принцип обработки информации.

Этот принцип подразумевает, что в каждый момент времени процессор работает над различными стадиями выполнения нескольких команд, причем на выполнение каждой стадии выделяются отдельные аппаратные ресурсы. По очередному тактовому импульсу каждая команда в конвейере продвигается на следующую стадию обработки, выполненная команда покидает конвейер, в него поступает новая.

В **скалярном конвейере** на разных ступенях обработки находятся команды с разными кодами операций, но обрабатываются одни и те же данные.

Конвейеризация не сокращает время, требуемое для завершения отдельной инструкции, которое также называется **латентностью**. Например, конвейер с пятью стадиями по-прежнему затрачивает 5 тактовых циклов для завершения инструкции. Конвейеризация повышает пропускную способность инструкций, а не время выполнения, или латентность, отдельно взятой инструкции, что является важным показателем, поскольку в реальной программе выполняются миллионы инструкций.

Набор инструкций MIPS был разработан для выполнения с использованием конвейера. В классическом варианте выполнение MIPS-инструкций занимает пять шагов, значит, MIPS-конвейер имеет пять стадий. Это означает, что на один тактовый цикл приходится до пяти инструкций. Таким образом, операционный блок делится на пять частей, где каждая из них названа в соответствии со стадией выполнения инструкции:

1. IF: извлечение инструкции (Instruction fetch).

2. ID: декодирование инструкции и чтение файла регистров (Instruction decode and register file read). (Постоянный формат MIPS-инструкций позволяет вести одновременное чтение и декодирование)
3. EX: выполнение операции или вычисление адреса (Execution or address calculation).
4. MEM: обращение к памяти данных (доступ к операнду) (Data memory access).
5. WB: обратная запись результата в регистр (Write back).

При конвейеризации создаются ситуации, когда следующая инструкция не может выполняться в следующем тактовом цикле. Такие ситуации называются *конфликтами* и делятся на три типа: структурные, конфликты данных и конфликты управления.

**Структурный конфликт** означает, что оборудование не может поддержать комбинацию инструкций, которую мы хотим выполнить за один и тот же тактовый цикл.

Например, если в машине есть единственный конвейер памяти для команд и данных, то, когда одна команда содержит обращение к памяти за данными, оно будет конфликтовать с выборкой более поздней команды из памяти.

Чтобы разрешить эту ситуацию, можно просто приостановить конвейер на один такт, когда происходит обращение к памяти за данными. Подобная приостановка часто называется “конвейерным пузырем” или просто пузырем, поскольку пузырь проходит по конвейеру, занимая место, но не выполняя никакой полезной работы (например, если возникла задержка на ступени считывания команды, то в следующем такте блок декодирования от него ничего не получит, а далее блок сохранения результатов ничего не получит от блока выполнения). Таким образом, скорость конвейера определяется скоростью самой медленной его ступени.

Борьба с конфликтами такого рода проводится путем увеличения количества однотипных функциональных устройств, которые могут одновременно выполнять одни и те же или схожие функции. В запоминающих устройствах в микропроцессорах с этой целью разделяют

кэш-память для хранения команд и кэш-память данных, а также используют многопортовую схему доступа к регистровой памяти, при которой к регистрам можно одновременно обращаться по нескольким каналам для записи и считывания информации. Например, в микропроцессоре Itanium к блоку регистров общего назначения допускается одновременное обращение на выполнение 8 операций чтения и 6 операций записи. В процессоре Pentium 4 для обработки целочисленных данных предусмотрено 4 АЛУ. При этом появляется возможность *параллельной обработки* информации в нескольких конвейерах.

Процессоры, имеющие в составе более одного конвейера, называются **суперскалярными**.

Недостатком суперскалярных микропроцессоров является необходимость синхронного продвижения команд в каждом из конвейеров. Для правильной работы при возникновении задержки в одном из конвейеров (например, если одной команде на какой-то стадии требуется больше тактов) должны приостанавливать свою работу и другие, иначе нарушится исходный порядок завершения команд программы. Но такие приостановки снижают быстродействие процессора.

Разрешение этой ситуации состоит в том, чтобы дать возможность выполняться командам в одном конвейере вне зависимости от ситуации в других конвейерах, а *аппаратные средства* микропроцессора должны гарантировать, что результаты выполненных команд будут записаны в приемник в том порядке, в котором команды записаны в программе. Это обеспечивается путем использования **принципа неупорядоченного выполнения команд**.

Блок выборки и декодирования выбирает команды из памяти и заносит их в *буфер команд*. По мере готовности операндов и исполнительного блока соответствующего типа команды извлекаются из буфера для обработки. Порядок их исполнения может отличаться от предписанного программой.

Результаты этапа выполнения команды сохраняются в специальном *буфере восстановления последовательности команд*. Запись результата очередной команды из этого буфера в приемник результата проводится лишь после того, как выполнены все предшествующие команды и записаны их результаты. Так команды максимально используют возможности всех конвейеров, обеспечивая максимальную производительность процессора.

**Конфликт данных** возникает тогда, когда запланированная инструкция не может быть выполнена в нужный тактовый цикл, поскольку данные, в которых она нуждается, еще недоступны. То есть когда выполнение одной команды зависит от результата выполнения предыдущей команды (такие команды называются **зависимыми по данным**; команды, которые могут выполняться в конвейере одновременно без приостановок, предполагая, что конвейер имеет достаточно ресурсов (структурные конфликты отсутствуют) - **параллельные** или **независимые**).

Пусть команда  $i$  предшествует команде  $j$ .

Существует несколько типов конфликтов по данным:

1. Конфликты типа RAW (Read After Write): команда  $j$  пытается прочесть операнд прежде, чем команда  $i$  запишет на это место свой результат. При этом команда  $j$  может получить некорректное старое значение операнда.
2. Конфликты типа WAR (Write After Read): команда  $j$  пытается записать результат в приемник прежде, чем он считается отсюда командой  $i$ . При этом команда  $i$  может получить некорректное новое значение операнда. Этот тип конфликтов, как правило, не возникает в системах с централизованным управлением потоком команд, обеспечивающих выполнение команд в порядке их поступления, так как последующая запись всегда выполняется позже, чем предшествующее считывание. Особенно часто конфликты такого рода могут возникать в системах, допускающих выполнение команд не в порядке их расположения в программном коде.

3. Конфликты типа WAW (Write After Write): команда  $j$  пытается записать результат в приемник прежде, чем в этот же приемник будет записан результат выполнения команды  $i$ , то есть запись заканчивается в неверном порядке, оставляя в приемнике значение, записанное командой  $i$ , а не  $j$ . Этот тип конфликтов присутствует только в конвейерах, которые выполняют запись со многих ступеней (или позволяют команде выполняться даже в случае, когда предыдущая приостановлена).

**Конфликт управления**, также называется конфликтом условного перехода, возникает, когда нужная инструкция не может быть выполнена в нужном тактовом цикле конвейера по причине того, что извлеченная инструкция не является той инструкцией, которая нужна, то есть поток адресов инструкций не соответствует ожиданиям конвейера.

Эффективность конвейера находится в прямой зависимости от того, с какой частотой на его вход подаются объекты обработки.

**Суперконвейеризация** позволяет улучшить производительность процессора за счет повышения частоты, с которой команды подаются на конвейер и перемещаются по нему. В обычном конвейере эта частота ограничена временем обработки в самой “медленной” ступени конвейера. В суперконвейере возможность увеличения частоты достигается путем выявления “медленных” ступеней и разбиения их на  $k$  меньших ступеней таким образом, чтобы время обработки в каждой из них не превышало аналогичного показателя для остальных ступеней конвейера.

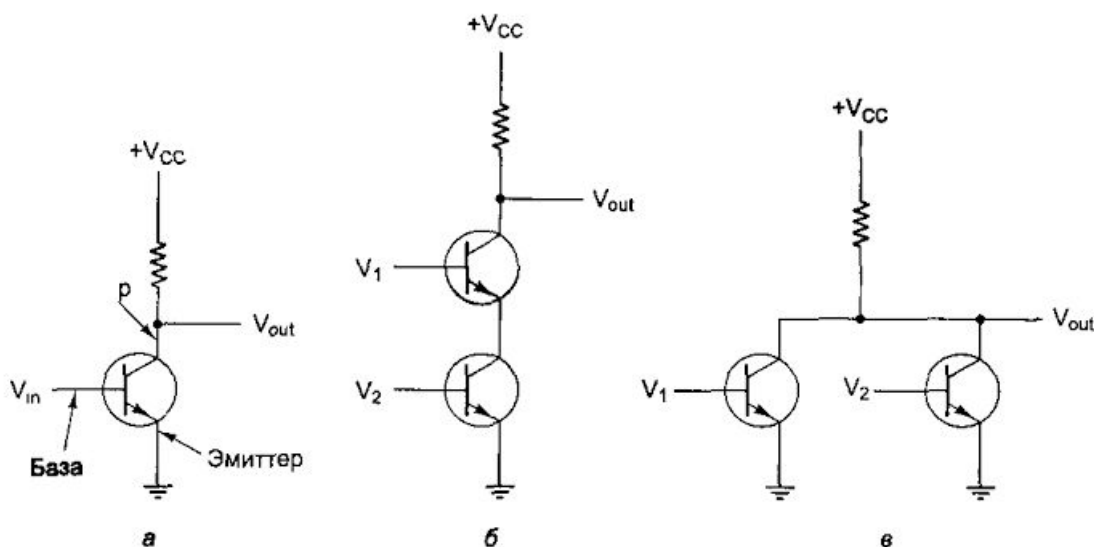
Показателем для причисления процессора к суперконвейерным служит число ступеней в конвейере команд. К суперконвейерным относят процессоры, где таких ступеней больше шести. Первым серийным суперконвейерным процессором считается MIPS R4000, конвейер команд которого включает в себя восемь ступеней. Суперконвейеризация здесь стала следствием разбиения этапов выборки команд и выборки операнда.



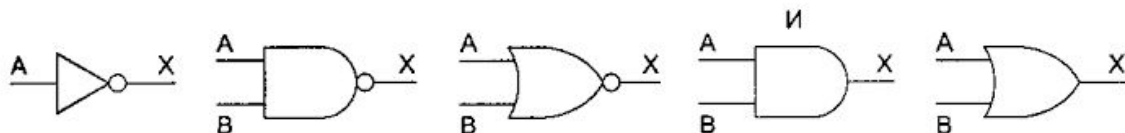


## 21. Основы схемотехники, базовые элементы, конструирование булевых функций.

**Цифровая схема** — это схема, в которой есть только два логических значения. Обычно сигнал от 0 до 1 В представляет одно значение, а сигнал от 2 до 5 В — другое значение. Напряжение за пределами указанных величин недопустимо. Крошечные электронные устройства, которые называются вентилями, могут вычислять различные функции от этих двузначных сигналов. Эти вентили формируют основу аппаратного обеспечения, на которой строятся все цифровые компьютеры.



(Физические реализации схем НЕ, НЕ-И, НЕ-ИЛИ)



(схематичное изображение НЕ, НЕ-И, НЕ-ИЛИ, И, ИЛИ)

С нуля на транзисторах можно сконструировать только схемы НЕ, НЕ-И, НЕ-ИЛИ, а схемы И, ИЛИ получаются с помощью сочетаний схем НЕ и НЕ-И, НЕ и НЕ-ИЛИ. Поэтому понятно, что схемы НЕ-И и НЕ-ИЛИ требуют меньше транзисторов, чем И и ИЛИ (конкретно, НЕ-И и НЕ-ИЛИ - по два транзистора, а И и ИЛИ - по три). Более

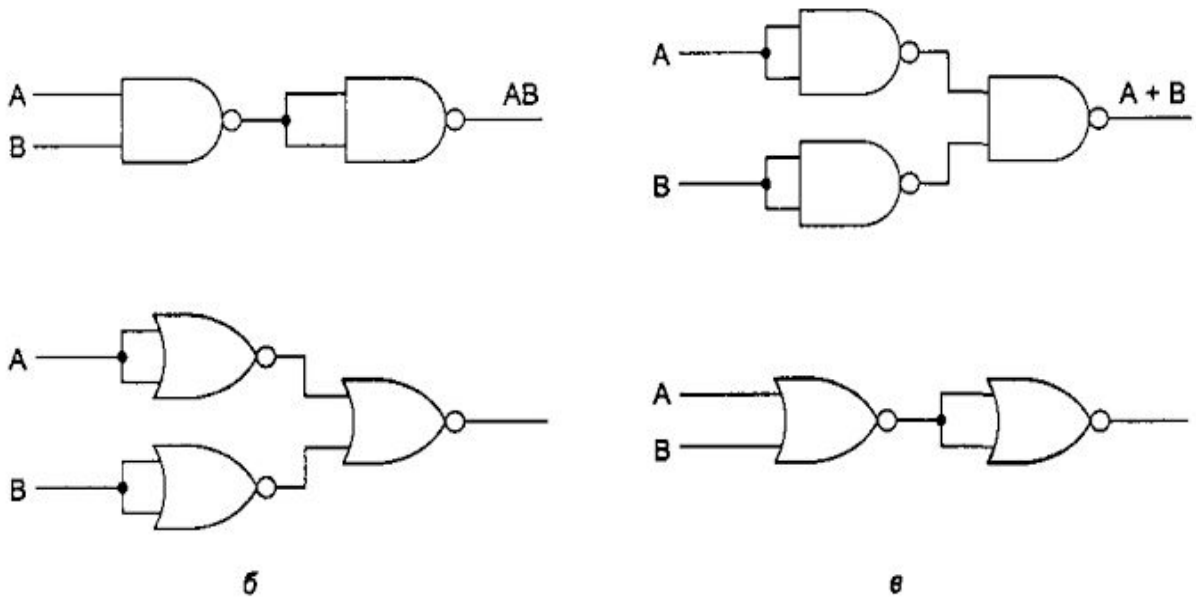
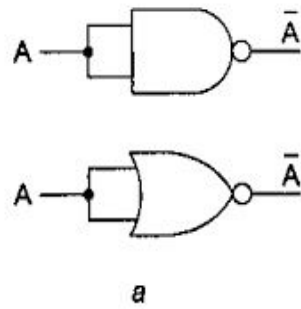
того, через только НЕ-И или только НЕ-ИЛИ выражается любая булева функция. Поэтому в большинстве ЭВМ в качестве базы используются эти вентили.

Понятно, что любую булеву функцию можно записывать с помощью таблицы истинностей. А теперь опишем алгоритм построения любой булевой функции:

- 1) Составить таблицу истинности для данной функции.
- 2) Обеспечить инверторы, чтобы порождать инверсии для каждого входного сигнала.
- 3) Нарисовать вентиль И для каждой строки таблицы истинности с результатом 1.
- 4) Соединить вентили И с соответствующими входными сигналами.
- 5) Вывести выходы всех вентилях И в вентиль ИЛИ.

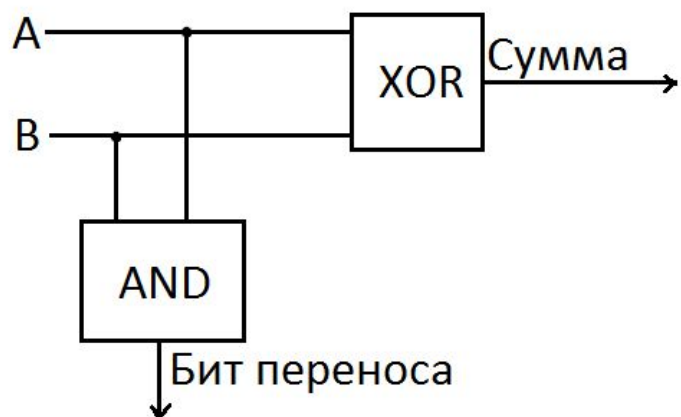
(Это просто является алгоритмом приведения к ДНФ на схемах)

Здесь приведен алгоритм реализации любой булевой функции на схемах НЕ, ИЛИ, И, но как мы сказали, лучше бы все это сделать на НЕ-ИЛИ и НЕ-И. Приведем реализацию НЕ, И, ИЛИ на схемах НЕ-И, НЕ-ИЛИ:



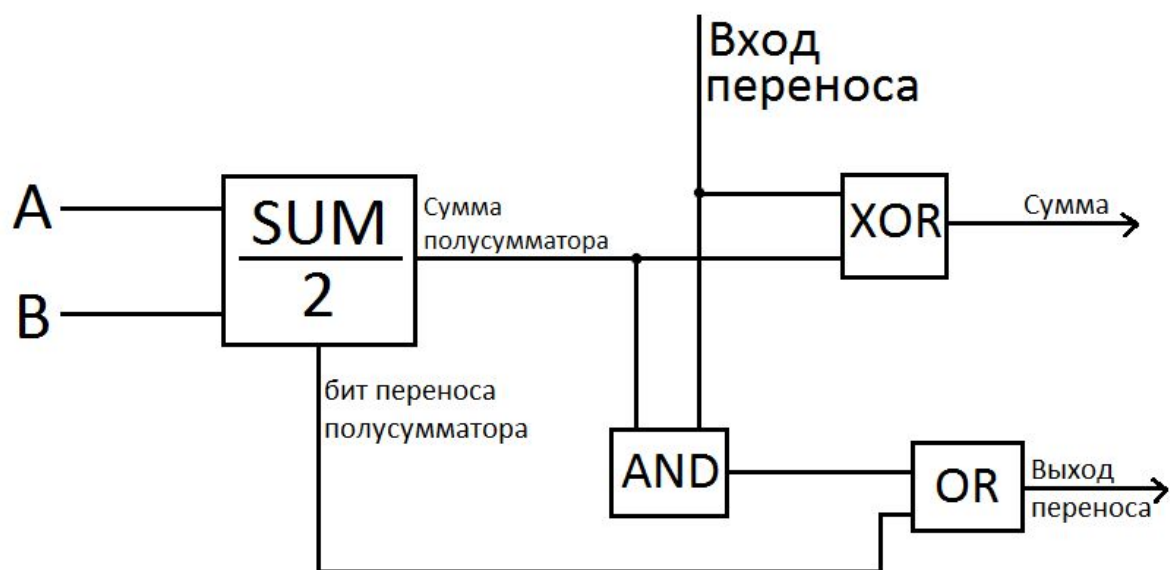
Однобитный полусумматор:

A	B	Сумма	Перенос
0	0	0	0
0	1	1	0
1	0	1	0
1	1	0	1



Однобитный полусумматор подходит только для сложения младших битов, потому что у него нет бита переноса на вход.

Однобитный сумматор на полусумматоре:



Однобитный сумматор на двух полусумматорах:

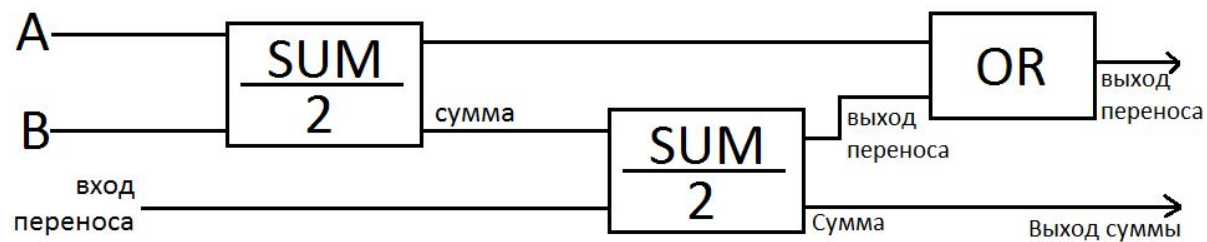
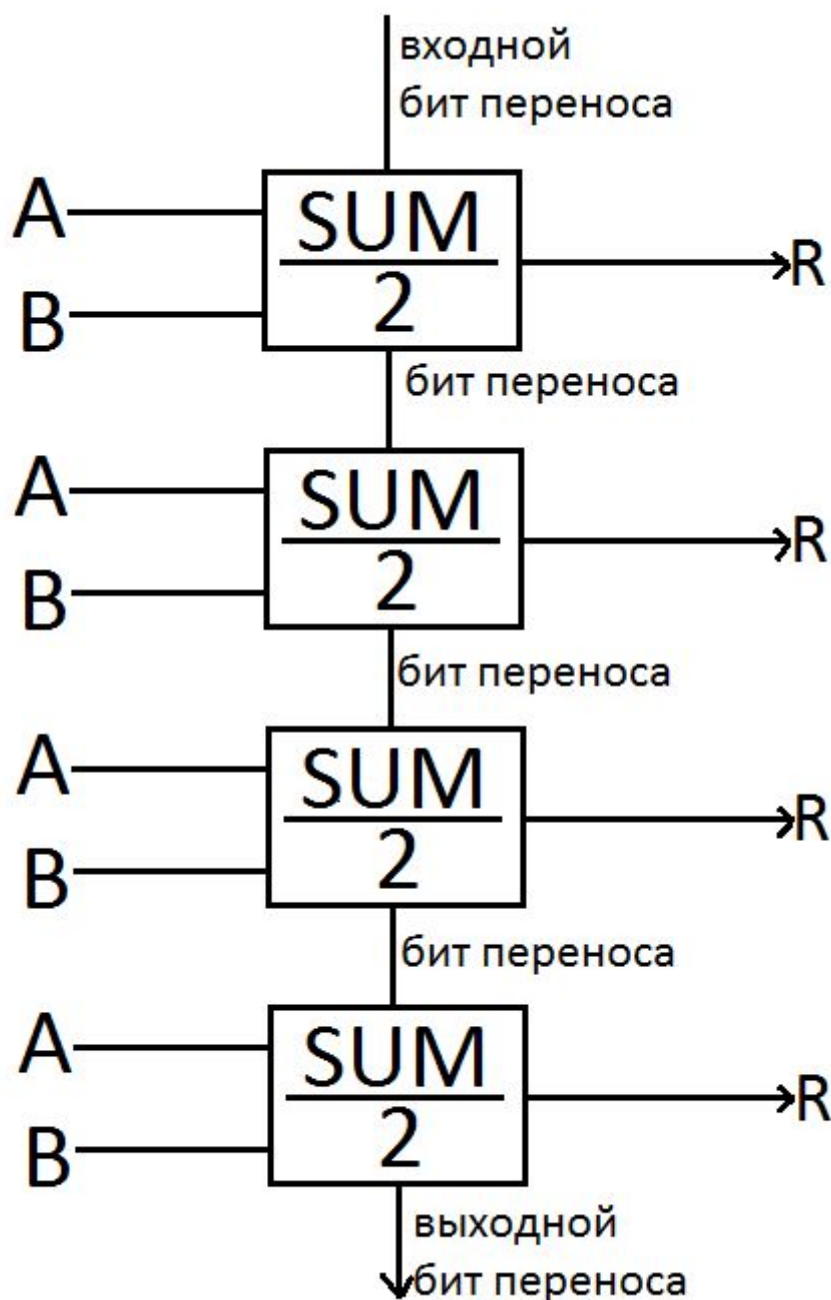


Таблица истинностей сумматора:

A	B	Вход переноса	Сумма	выход переноса
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

Однобитный сумматор подходит для сложения любых битов.

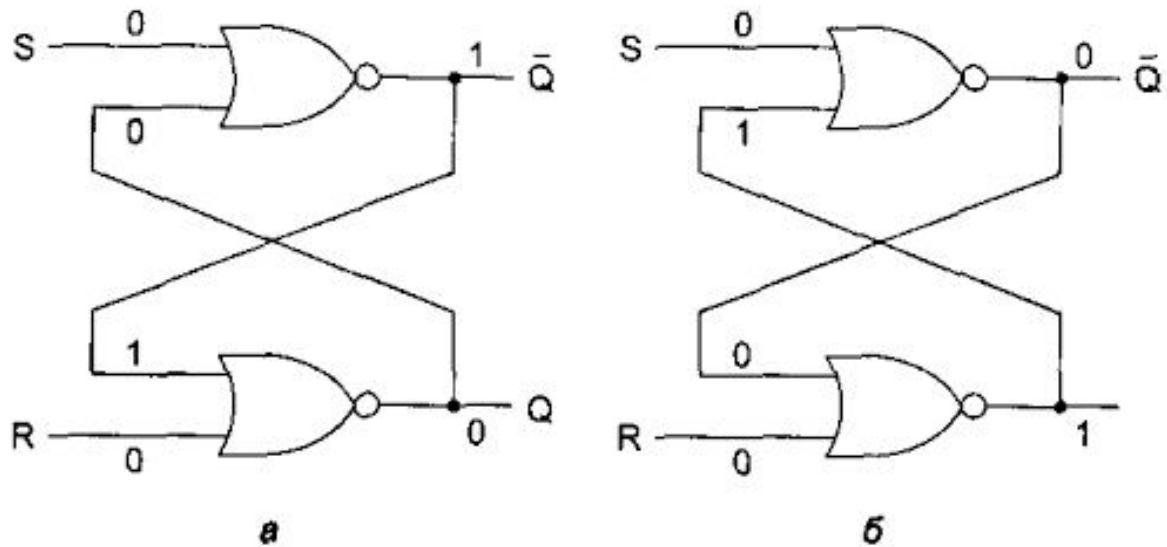
Пример полного четырехразрядного сумматора:



A, B - биты на вход соответствующего полусумматора, R - результат соответствующего полусумматора,  $\frac{SUM}{2}$  - полусумматор.

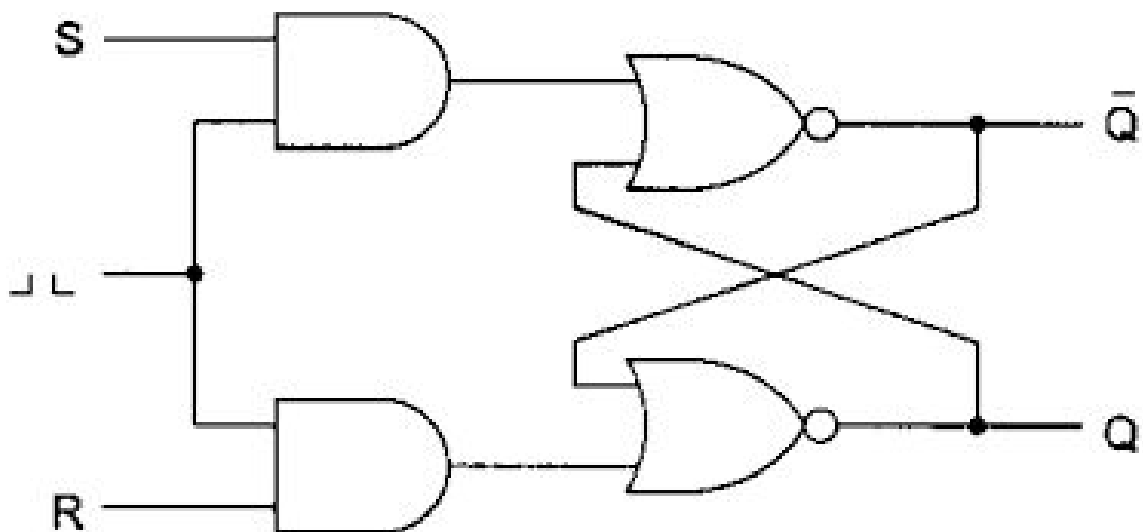
Дается изначальный бит переноса, он учитывается первым полусумматором, на выходе у первого полусумматора также появляется бит переноса. Он передается второму на вход и т.д. В итоге снова остается бит переноса при переполнении.

SR-защелка:



Чтобы создать бит памяти, нужна схема, которая будет запоминать предыдущие входные значения. Для этого придумали защелку, она основана на вентиле NOR. Чтобы установить сигнал, нужно подать 1 на вход S(set). Тогда защелка “Запомнит” состояние, на выходе  $Q=1$ ,  $\bar{Q}=0$ . Также, можно сбросить сигнал, подав 1 на вход R, тогда на выходе будет  $Q=0$ ,  $\bar{Q}=1$ .

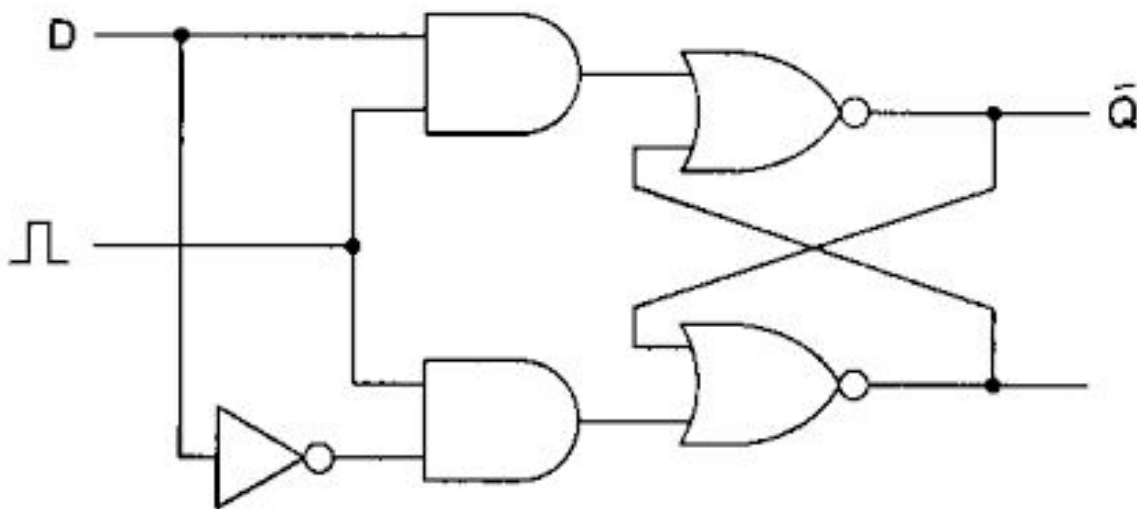
Синхронная SR-защелка:



Иногда нужно, чтобы защелка срабатывала только в определенное время. Поэтому добавлены тактовый генератор и два вентиля И,

которые контролируют это. Пока тактовый генератор не подал сигнал 1, независимо от S и R, защелка не будет менять состояние благодаря И. Когда же тактовый генератор подал сигнал 1, вентили И “исчезнут” для защелки и она будет работать так, будто это обычная SR-защелка.

У SR-защелок есть недостаток: если подать на SR-защелку сигнал  $S=R=1$ , то после того, как входной сигнал перестанет отдаваться защелке (т.е.  $S=R=0$ ), она будет вести себя недетерминированно. Чтобы решить эту проблему ввели D-защелку:



Это усовершенствованная синхронная SR-защелка, в которой проблема совпадения входных сигналов решилась удалением одного из них. При включенном тактовом генераторе (подается сигнал 1), она просто запоминает сигнал D. При выключенном тактовом генераторе сигнал D просто игнорируется.

## **22. Предсказание переходов. Регистровые окна и переименование регистров.**

На вход процессора поступает поток инструкций для их последующего исполнения. Инструкции поступают в том порядке, в котором они содержатся в коде программы, исполняемой в данный момент процессором. Как только на входе процессора появляется очередная порция инструкций для исполнения, ее содержимое анализируется с целью найти точки ветвления в исполняемом потоке инструкций и предсказать наиболее вероятные пути (ветви), по которым пойдет обработка инструкций после этих точек ветвления. Инструкции, принадлежащие ветвям с наибольшей вероятностью выполнения, тут же ставятся в очередь на исполнение.

Основная идея - заставить процессор выполнить инструкции, которые принадлежат ветвям с наибольшей вероятностью выполнения, "вне очереди" - то есть не дожидаться того момента, когда очередь на выполнение дойдет до этих ветвей естественным образом (согласно порядку поступления инструкций на вход процессора и, соответственно, контексту выполняемой программы), а загрузить эти ветви на выполнение раньше этого момента. В современных процессорах для предсказания адреса перехода обычно используют специальную таблицу адресов переходов BTB (Branch Target Buffer). Эта таблица устроена подобно кэшу и содержит адреса инструкций, на которые ранее производились переходы. Спекулятивное выполнение инструкций - это способность процессора исполнить инструкции в порядке, отличном (как правило, с опережением) от порядка во входном потоке инструкций (что определяется кодом исполняемой программы), но завершить и вернуть результаты исполнения инструкций в порядке, соответствующем оригинальному

входному потоку инструкций. Выполнение команды до того, как стало известно, понадобится ли вообще эта команда, называется спекулятивным выполнением. Переименование регистров (register renaming) является аппаратной техникой уменьшения конфликтов из-за регистровых ресурсов. Компиляторы преобразуют языки высокого уровня в ассемблерный код, назначая регистрам те или иные значения. В суперскалярном процессоре операция может потребовать регистр до того, как предыдущая инструкция закончила использование этого регистра. Это состояние не является конфликтом данных, поскольку этой операции не требуется значение регистра, а только сам регистр. Однако, эта ситуация приводит к остановке конвейера до освобождения регистра. Идея разрешения этой проблемы состоит в следующем: берем свободный регистр, переименовываем его для соответствия параметрам инструкции, и даем инструкции его использовать в качестве требуемого ей регистра. Так как каждая часть программы хочет задействовать все регистры, видимые ей, становится существенным предоставить каждой части программы некоторый набор регистров. Фишка заключается в том, чтобы сделать какие-то регистры невидимыми для программы. Команда вызова процедуры скрывает старый набор регистров и предоставляет новый набор, который может использовать вызванная процедура. Однако некоторые регистры переносятся из вызывающей процедуры к вызванной процедуре, что обеспечивает эффективный способ передачи параметров между процедурами. Для этого некоторые регистры переименовываются. В парадигме регистрового окна процессорные регистры общего назначения делятся на глобальные регистры (для хранения глобальных переменных) и регистровый файл, не



видимый целиком никакой программе. Пусть регистровый файл состоит из  $K$  регистров, расположенных по кругу. Каждой программе доступны лишь  $L = L_1 + L_2 + L_3$  регистров ( $L < K$ ), составляющих текущее регистровое окно. Окно делится на три части: \*  $L_1$  регистров параметров (для получения параметров от вызвавшей программы) \*  $L_2$  локальных регистров (для хранения промежуточных данных)

\*  $L_3$  временных регистров (для передачи параметров в процедуру)

## 23. Классификация Флинна с примерами реализации архитектур.

Определения для более хорошего понимания билета:

Конвейерная обработка – одновременное выполнение нескольких различных этапов одной операции на различных ступенях конвейера.

Векторный процессор — это процессор, в котором операндами некоторых команд могут выступать упорядоченные массивы данных — векторы. При выполнении векторной обработки – одна и та же операция применяется ко всем элементам вектора.

Векторные компьютеры - компьютеры, использующие векторные процессоры.

Супер-скалярная обработка - наличие в аппаратуре средств, позволяющих одновременно выполнять две и более скалярные операции, т.е. команд обработки пары чисел (например, процессоры DEC серии Alpha)

Классификация М. Флинна является одной из самых ранних и наиболее известных классификацией архитектур вычислительных систем. В основу классификации положено понятие потока. **Поток** - это последовательность, под которой понимается последовательность данных или команд, обрабатываемых процессором.

Рассматривая число потоков данных и потоков команд, М. Флинн предложил рассматривать следующие классы архитектур: MIMD, SIMD, SISD, MISD.

**SISD** (single instruction stream / single data stream) - одиночный поток команд и одиночный поток данных. К этому классу относятся, прежде всего, классические последовательные машины, или иначе, машины фон-неймановского типа, например, PDP-11 или VAX 11/780. В таких машинах есть только один поток команд, все команды обрабатываются последовательно друг за другом и каждая команда инициирует одну операцию с одним потоком данных. Не имеет значения тот факт, что для увеличения скорости обработки команд и скорости выполнения арифметических операций может применяться конвейерная обработка - как машина CDC 6600 со скалярными функциональными устройствами, так и CDC 7600 с конвейерными попадают в этот класс. В этот класс входят однопроцессорные последовательные компьютеры. Векторно-конвейерные компьютеры также могут быть отнесены к этому классу, если рассматривать вектор как одно неделимое данное для машинной команды. Примерами компьютеров с архитектурой SISD являются большинство рабочих станций Compaq, Hewlett-Packard и Sun Microsystems. Это обычные скалярные, однопроцессорные системы.

**SIMD** (single instruction stream / multiple data stream) - одиночный поток команд и множественный поток данных. В архитектурах подобного рода сохраняется один поток команд, включающий, в отличие от предыдущего класса, векторные команды. Это позволяет выполнять одну арифметическую операцию сразу над многими данными - элементами вектора. Способ выполнения векторных операций не оговаривается, поэтому обработка элементов вектора может производиться либо процессорной матрицей, как в ILLIAC IV, либо с помощью конвейера, как, например, в машине CRAY-1. К этому классу относятся однопроцессорные, векторно-конвейерные

суперкомпьютеры, например Cray – 1 и более современный Cray Y-MP. В этом случае мы имеем дело с одним потоком (векторных) команд, а потоков данных много; каждый элемент вектора входит в отдельный поток данных.

**MISD** (multiple instruction stream / single data stream) - множественный поток команд и одиночный поток данных. Определение подразумевает наличие в архитектуре многих процессоров, обрабатывающих один и тот же поток данных. Однако ни Флинн, ни другие специалисты в области архитектуры компьютеров до сих пор не смогли представить убедительный пример реально существующей вычислительной системы, построенной на данном принципе. Ряд исследователей относят конвейерные машины к данному классу, однако это не нашло окончательного признания в научном сообществе. Будем считать, что пока данный класс пуст.

**MIMD**: (multiple instruction stream / multiple data stream) - множественный поток команд и множественный поток данных. Этот класс предполагает, что в вычислительной системе есть несколько устройств обработки команд, объединенных в единый комплекс и работающих каждое со своим потоком команд и данных. Включает в себя всевозможные мультипроцессорные системы: Cm\*, C.mmp, CRAY Y-MP, Denelcor HEP, BBN Butterfly, Intel Paragon, CRAY T3D и многие другие. Интересно то, что если конвейерную обработку рассматривать как выполнение множества команд (операций ступеней конвейера) не над одиночным векторным потоком данных, а над множественным скалярным потоком, то все рассмотренные выше векторно-конвейерные компьютеры можно расположить и в данном классе.

У этой классификации есть очевидные недостатки:

- в нее четко не вписываются отдельные нашедшие применение архитектуры. Например, векторно-конвейерные компьютеры и компьютеры, управляемые потоками данных;
- класс MIMD очень перегружен: в него вошли все многопроцессорные системы. При этом они существенно отличаются по ряду признаков (числом процессоров, природе и топологией и видами связей между ними, способами организации памяти и технологиями программирования).

## 24. Архитектуры VLIW и EPIC. Особенности спекулятивного исполнения инструкций в архитектуре EPIC.

**Архитектура VLIW (Very Long Instruction Word)** известна с начала 80-х из ряда университетских проектов, но только сейчас, с развитием технологии производства микросхем она нашла свое достойное воплощение.

Идея VLIW базируется на том, что задача эффективного планирования параллельного выполнения нескольких команд возлагается на «разумный» компилятор. Такой компилятор вначале исследует исходную программу с целью обнаружить все команды, которые могут быть выполнены одновременно, причем так, чтобы это не приводило к возникновению конфликтов. В процессе анализа компилятор может даже частично имитировать выполнение рассматриваемой программы. На следующем этапе компилятор пытается объединить такие команды в пакеты, каждый из которых рассматривается так одна сверхдлинная команда. Объединение нескольких простых команд в одну сверхдлинную производится по следующим правилам:

- 1) количество простых команд, объединяемых в одну команду сверхбольшой длины, равно числу имеющихся в процессоре функциональными блоками = ФБ = АЛУ;
- 2) в сверхдлинную команду входят только такие простые команды, которые исполняются разными ФБ, то есть обеспечивается одновременное исполнение всех составляющих сверхдлинной команды.

Длина сверхдлинной команды обычно составляет от 128 до 1024 бит. Такая команда содержит несколько полей (по числу образующих ее простых команд), каждое из которых описывает операцию для конкретного функционального блока.

Во VLIW-архитектуре распараллеливание кода производится на этапе компиляции(код скомпилированный для VLIW'a не будет работать на другой архитектуре, а так же не будет работать должным образом если у нас поменялось кол-во АЛУ), а не динамически во время исполнения. То, что в выполняемой сверхдлинной команде исключена возможность конфликтов, позволяет предельно упростить аппаратуру VLIW-процессора и, как следствие, добиться более высокого быстродействия.

В качестве простых команд, образующих сверхдлинную, обычно используются команды RISC-типа, поэтому архитектуру VLIW иногда называют постRISC-архитектурой. Максимальное число полей в сверхдлинной команде равно числу вычислительных устройств и обычно колеблется в диапазоне от 3 до 20. Все вычислительные устройства имеют доступ к данным, хранящимся в едином многопортовом регистровом файле( модуль микропроцессора, содержащий в себе реализацию регистров процессора). Отсутствие сложных аппаратных механизмов дает значительный выигрыш в быстродействии.

=====

**Архитектура EPIC** - архитектура с явным параллелизмом команд, базовые принципы были разработаны в университете Иллинойса, проект имел название Impact. В начале 1990-х годов были заложены теоретические основы самой архитектуры. Архитектура EPIC имеет следующие основные особенности:

- 1) поддержка явно выделенного компилятором параллелизма. Формат команд имеет много общего с архитектурой VLIW - параллелизм так же явно выделен. В процессоре EPIC существует несколько длинных команд, в которых каждому АЛУ явно сопоставлена операция. Есть сопоставление инструкций из различных тактов выполнения. Пример:
  - а) Пусть инструкция явно выполняется за 1-2 такта, тогда, в инструкции явно специфицированы между какими командами имеется **слот задержки** (фактически "граница" между командами, выполняемыми в разных тактах).  
К каждой длинной команде (128 бит) прилагается небольшой (3-5 битный) **ярлык**, который специфицирует формат команды.
- 2) наличие большого регистрового файла( модуль микропроцессора, содержащий в себе реализацию регистров процессора)
- 3) спекулятивная загрузка данных, заключается в смене порядка обращений к памяти, позволяющая избежать простоев конвейера при загрузке данных из оперативной памяти
- 4) Есть наличие **предикатных регистров**, а так же поддержка предикатно-выполняемых команд, которая позволяет:
  - а) избежать излишних инструкций ветвления, если количество команд в ветвях условного оператора невелико;
  - б) уменьшить нагрузку на устройство предсказания переходов.
- 5) аппаратная поддержка программной конвейеризации - это оптимизированный способ выполнения цикла. Если итерации в цикле могут выполняться независимо, то за счет имеющихся АЛУ они могут исполняться параллельно.
- 6) Используется стек регистров (и **регистровые окна**).
- 7) Для предсказания инструкций используется поддержка компилятора.
- 8) Введена поддержка инструкций циклического выполнения команд без потерь времени на инструкции циклического выполнения.

#### Unpredicated Code

```
cmp a,b
jump EQ
y=3
jump END
EQ: y=4
END:
```

#### Predicated Code

```
cmp.eq p1,p2=a,
p1 y=4
p2 y=3
```

Наиболее интересной в архитектуре EPIC является поддержка спекулятивное выполнение команд и загрузка данных.

Спекулятивное исполнение команд. Большинство команд загрузки данных из памяти выполняется длительное время. Выполнение команды за 1-2 такта возможно только в том случае, если значение содержится в кеш-памяти первого уровня. Время несколько увеличивается, если значение находится в кеш-памяти второго уровня. В случае чтения же данных из микросхем динамической памяти даже при попадании на активную страницу происходит значительная задержка при загрузке, а при смене страницы задержка имеет огромную величину, причём конвейер быстро блокируется, таким образом команд, которые можно выполнить без большой задержки, обычно крайне мало.

При спекулятивном выполнении используется вынесение команд загрузки далеко вперёд инструкций, использующих эти данные, в основном вверх за инструкции условного перехода. При достижении потоком команд места, где необходимо использовать загружаемые данные, вставляется инструкция, проверяющая не произошло ли исключения в процессе загрузки (например, какая-то инструкция произвела запись по этому адресу), если исключение произошло, то вызывается специально написанный восстановительный код, который попросту перезагружает значение в регистр.

=====

## 25. Закон Амдала.

Сформулирован в 1967 году главным архитектором линейки компьютеров [IBM System/360](#) (Пьянзин упоминал про них на паре, так что этот факт очень в тему) Джином Амдалом.

**Закон Амдала** (*об ограниченной горизонтальной масштабируемости*):

Пусть необходимо решить некоторую вычислительную задачу.

Предположим, что её алгоритм таков, что доля  $\alpha$  от общего объёма вычислений может быть получена только последовательными расчетами, а, соответственно, доля  $1 - \alpha$  может быть распараллелена идеально (то есть время вычисления будет обратно пропорционально числу задействованных узлов  $p$ ).

Тогда ускорение, которое может быть получено на вычислительной системе из  $p$  процессоров, по сравнению с однопроцессорным решением не будет превышать величины:

$$S_p = \frac{1}{\alpha + \frac{1-\alpha}{p}}$$

Основным следствием этой формулы является то, что ускорение выполнения программы за счет распараллеливания её инструкций на множестве вычислителей **ограничено** временем, необходимым для выполнения её последовательных инструкций.

Более того, если учесть время, необходимое для передачи данных между узлами вычислительной системы, то зависимость времени вычислений от числа узлов будет иметь строгий максимум. Это накладывает ограничение на масштабируемость вычислительной системы, то есть означает, что с определенного момента **добавление новых узлов** в систему будет **увеличивать** время расчета задачи.

## 26. Дополнения Ванга и Бриггса к классификации Флинна.

*Функциональные устройства процессора: выполняют логические и арифметические операции (АЛУ).*

**Мультипроцессоры** - машины с одним адресным пространством на все процессы. Они же называются

сильно-связными вычислительными системами. Управление обеспечивается одной общей операционной системой. При этом достигаются более быстрый обмен информацией между процессорами, чем между ЭВМ в многомашинных вычислительных системах (комплексах), и более высокая суммарная производительность системы. В таких системах, как правило, число параллельных процессов невелико и управляет ими централизованная операционная система. Процессы обмениваются информацией через общую оперативную память. При этом возникают задержки из-за межпроцессорных конфликтов. При создании больших мультипроцессорных ЭВМ (мэйнфреймов, суперЭВМ) предпринимаются огромные усилия по увеличению пропускной способности оперативной памяти. В результате аппаратные затраты возрастают чуть ли не в квадратичной зависимости, а производительность системы упорно «не желает» увеличиваться пропорционально числу процессоров. Так, сложнейшие средства снижения межпроцессорных конфликтов в оперативной памяти суперкомпьютеров серии CRAY X-MP/Y-MP позволяют получить коэффициент ускорения не более 3,5 для четырехпроцессорной конфигурации системы.

**Мультикомпьютеры** - машины с обменом сообщениями, без общей памяти, состоят из большого числа взаимосвязанных компьютеров, у каждого из которых имеется собственная память. Отдельные компьютеры объединяются либо с помощью сетевых средств, либо с помощью общей внешней памяти (обычно — дисковые накопители большой емкости). Каждая ЭВМ системы имеет свою оперативную память и работает под управлением своей операционной системы. Каждая машина использует другую как канал или устройство ввода-вывода. Обмен информацией между машинами происходит в результате взаимодействия их операционных систем.

Классификация Ванга и Бриггса сохраняет четыре базовых класса (SISD, SIMD, MISD, MIMD) классификации Флинна. К. Ванг и Ф. Бриггс внесли следующие изменения.

В классе MIMD выделяются:

вычислительные системы со слабой связью между процессорами, к которым они относят все системы с распределенной памятью (например, Cosmic Cube) и вычислительные системы с сильной связью (системы с общей памятью), куда попадают такие компьютеры, как C.mmp, BBN Butterfly, Cray Y-MP, Denelcor HEP.

Класс SISD делится на два подкласса:

архитектуры с единственным функциональным устройством, например, серия PDP-11

архитектуры, имеющие в своем составе несколько функциональных устройств - CDC 6600, Cray-1, FPS AP-120B, CDC Cyber 205, FACOM VP-200.

Класс SIMD с учетом способа обработки данных делится на два подкласса:

архитектуры с пословно-последовательной обработкой информации - ILLIAC IV, PEPE, BSP

архитектуры с разрядно-последовательной обработкой - STARAN, ICL DAP.



## 27. Архитектура MIPS

//ещё будет правиться

**MIPS** (Microprocessor without Interlocked Pipeline Stages), - "процессор без блокировок в конвейере". Основная идея, которой руководствовался Джон Хеннеси, со своей командой проектировавший в 1981 году первый MIPS-процессор, такова. Сильно упростив внутреннее устройство CPU и используя очень длинный (по тем временам) конвейер, можно получить процессор, не умеющий выполнять сравнительно сложные инструкции, зато работающий на очень высоких тактовых частотах, позволяющих скомпенсировать потери производительности на эмуляцию этих сложных инструкций. Изначально предполагалось, что MIPS-процессоры не будут аппаратно поддерживать даже операции умножения и деления - благодаря чему можно было обойтись без сложных в реализации блокировок конвейера (Процедура приостановки конвейера, инициируемая, когда процессору встречается "медленно выполняющаяся" операция, которую невозможно выполнить на какой-то из стадий за один такт. В процессорах тех времен такими операциями являлись умножение и деление; в современных процессорах блокировку может вызвать неудачное обращение в оперативную память, не находящуюся в кэше CPU) (отсюда и название архитектуры). Тем не менее даже в самых первых MIPS'ах блокировки в конвейере, равно как и аппаратные инструкции умножения и деления все-таки присутствовали - "в чистом виде" идея оказалась малоприменимой для создания коммерческих процессоров.

В 1984 году Хеннеси с командой покинул Стэнфордский университет и основал компанию MIPS Computer Systems. В 1985 она выпустила первый 32-разрядный MIPS-процессор R2000; в 1988 году - гораздо более быстрый, работающий с виртуальной памятью и поддерживающий многопроцессорность R3000. R3000 стал первым по-настоящему коммерчески успешным MIPS-процессором и использовался в рабочих станциях Silicon Graphics. Вариант MIPS R3000A хорошо известен в народе как центральный процессор приставки Sony PlayStation.

В 1991 году вышел первый 64-разрядный MIPS R4000, легший в основу целого ряда различных процессоров, выпускавшихся по лицензиям другими фирмами. R4000 оказался настолько важен для SGI, что она не колеблясь приобрела испытывавшую тогда финансовые затруднения MIPS Computer Systems и превратила эту компанию в собственное подразделение MIPS Technologies. Тогда же SGI начала продавать лицензии на производство MIPS-процессоров сторонним фирмам, которые взялись разрабатывать свои, улучшенные варианты R4000. Помимо всего прочего, начиная с R4600 и R4700 (разработка Quantum Effects Devices) MIPS-процессоры стали основой для знаменитых маршрутизаторов Cisco, являющихся сегодня неотъемлемой частью большинства крупных сетей, включая интернет. Использовались 64-разрядные MIPS-процессоры и в приставках: R4300 - в Nintendo 64, R5900 - в PlayStation 2. В 1994 году вышел R8000 - первый суперскалярный MIPS-процессор; в 1995-м - R10000, улучшенный во всех отношениях вариант R8000, поддерживавший внеочередное исполнение команд в конвейере. Работая на частоте 200 МГц, R10000 был одним из самых быстрых CPU того времени. На те времена пришелся расцвет архитектуры MIPS - она была столь успешной, что в 1998 году SGI снова сделала из MIPS Technologies отдельную компанию. Правда, SGI сочла дальнейшее развитие MIPS как своей флагманской разработки бесперспективным и решила, когда настанет срок, перевести линейку Silicon Graphics на процессоры архитектуры IA-64 (Intel Itanium).

В итоге дизайн всех последующих MIPS-процессоров основывался на R10000. Изменялись только объем кэш-памяти и постепенно наращивалась тактовая частота. Фактически после прорыва R10000 архитектура MIPS была заброшена, и эти процессоры постепенно утратили лидирующее положение в индустрии. В 2001 году топовым CPU от MIPS Technologies был R14000 с тем же старым ядром R10000 и тактовой частотой всего 600 МГц. Конкуренты в лице, к примеру, более совершенных в технологическом плане AMD Athlon уже достигли частот 1,3-1,4 ГГц, были в несколько раз производительнее, а стоили куда меньше. Так что как "тяжелая высокопроизводительная RISC-архитектура" MIPS к началу нового тысячелетия умерла. Но компания MIPS Technologies процветает до сих пор - за счет лицензирования архитектуры сторонним разработчикам.

Еще в 1999 году MIPS Technologies упростила свою лицензионную политику, предложив всем желающим два варианта MIPS-архитектуры: MIPS32 - для 32-разрядных систем и MIPS64 - для 64-разрядных. Сегодня MIPS - самая популярная высокопроизводительная архитектура, используемая во встраиваемых системах.

## 28. Согласование кэшей в мультипроцессорных системах и многоядерных процессорах.

*Ключевые слова:* Каталог, когерентный кэш. MOESI, MESI

Если бы ячейки памяти были доступны только на чтение, то копия ячейки к кэше и в памяти всегда бы совпадала. Но это не так. Из-за этого получаем несколько проблем: 1) кэш-контроллер должен отслеживать модификацию ячеек кэш-памяти, выгружая в основную память модифицированные ячейки, при их замещении. 2) Нужно отслеживать, кто и когда обращается к ячейкам (включая остальные микропроцессоры в многопроцессорных компьютерах).

То есть, допустим, ячейка в кэш-памяти уже модифицирована, но еще не выгружена в основную память. А к ней обращается периферийное устройство (или другой процессор), то кэш-контроллер должен что-то предпринять, чтобы из основной памяти не считались устаревшие данные.

Ввиду этого, кэш-контроллер обеспечивает когерентность. **Когерентность - согласованность кэш-памяти с основной памятью.**

Поддержка когерентности решается несколькими способами.

- 1) **Сквозная политика.** Самое простое решение: ячейки основной памяти кэшируются только для чтения, а запись осуществляется на прямую, то есть не заходя в кэш. Такое решение очень неэффективно (вполне понятно, почему)
- 2) **Сквозная политика с буферизацией.** Записываемые данные попадают не сразу в основную память, а в специальный *буфер записи*. Его размер порядка 32 байт. В нем данные накапливаются до тех пор, пока буфер не заполнится или не освободится шина. После все содержимое буфера записывается за раз, одним большим блоком. Тем не менее, даже тут видно, что куча времени уходит именно на выгрузку буфера.

3) **Обратная политика записи.** Для отслеживания модификации с каждой ячейкой кэш-памяти (кэш-строкой) связывается специальный флаг, именуемый *флагом состояния*. Если кэшируемая ячейка модифицирована, то кэш-контроллер устанавливает флаг в состояние “грязно” (dirty). Когда к такой ячейке обращается другой процессор или периферийное устройство, кэш-контроллер проверяет, есть ли адрес (с помощью тэга, конечно) в кэш-памяти, и если да, то проверяется его флаг. Если флаг “грязный”, то она загружается в основную память, а флаг становится “чистым”.

Замещение кэш-строк производится так же, как и в последнем алгоритме. При замещении старых строк новыми, контроллер сначала пытается избавиться от чистых строк (экономия времени). И только в случае, когда все ячейки “грязные”, выбирается одна, наименее ценная (это уже определяется алгоритмом политики замещения данных), она и перемещается в основную память, меняясь на новую, “чистую”.

Конечно, все просто, только не в случае, когда у нас есть многоядерный процессор, а уж всё ещё сложнее, когда многопроцессорная архитектура. Если пытаться логично рассуждать, то раз каждое ядро (про процессоры пока не говорим) что-то само рассчитывает, то ему нужно частенько обращаться к данным, с которыми оно имеет дело, значит, у ядра есть необходимость в своем собственном кэше маленького размера (это будет L1). И так, как этот кэш вообще-то независим от остальных, но при этом есть общий кэш внутри процессора (большого размера, но и медленнее -L2), а память, в которой будет вестись запись(читаться), в конечном итоге одна - оперативная, то остро всплывает проблема когерентности. Поэтому и появляется необходимость в следующем протоколе - **MESI (Modified Exclusive Shared Invalid)**.

Далее также будет упомянут и более новый протокол, **MOESI**, появившийся впервые в AMD Athlon (многопроцессорный). Его принцип работы отличается не только наличием дополнительного флага, но и изменением смысла остальных. Логично рассмотреть оба. **Инфа 100% Пьянзин спрашивает различия, сам уточнил.** Пьянзин сказал, что можно не

уточнять, как общаются кэши между процессорами, хотя они это делают через дополнительное устройство на плате, что-то типа общего кэш-контроллера, проверяющего все запросы. Но про детали взаимодействия можно не говорить.

**Modified** - кэш-строка присутствует только в этом кэше и является “грязной” - то есть модифицирована. Требуется записать значение обратно в память, возможно, что и не сейчас, но прежде, чем какой-либо другой кэш попытается прочитать содержимое. Кэш-строка теперь имеет активным только флаг **Invalid**. Если данные будут записаны в память, а после другой кэш потребует чтение, то будет активным флаг **Sharing**.

**Exclusive** - кэш-строка присутствует только в данном кэше, но, важное отличие, она является “чистой” - соответствует тому, что лежит в основной памяти. В любой момент времени может поменять свой статус на **Sharing**, если кто-то отправит запрос на ее чтение. Альтернативно, перейдет в **Modified**, если будет изменена.

**Shared** - означает, что кэш-строка “чистая”, и что она может присутствовать в кэш-строках других кэшей. В любой момент времени может изменить свое состояние и поставить флаг Invalid (идея в том, что именно никогда нельзя гарантировать, будет ли этот флаг, если, конечно, принять во внимание умный кэш).

**Invalid** - строка отсутствует в кэше (очень вероятно), или другой кэш изменил состояние данных, переводя ее у себя в **Modified**, поэтому использование запрещено.

Краткая таблица практического применения протокола:

Статус/свойства	Modified	Exclusive	Shared	Invalid
Эта кэш-линия действительна?	да	да	да	нет
Копия в памяти действительна?	устарела	действительна	действительна	этой строке вообще не соответствует никакая память
Содержится ли копия этой строки в других процессорах?	нет	нет	может быть	может быть
Запись в эту линию осуществляется...	только в эту строку, без обращения к шине	только в эту строку, без обращения к шине	сквозной записью в память с аннулированием строки в кэшах остальных процессоров	непосредственно через шину

Пример работы MESI для многопроцессорных устройств:

Имеем 2 процессора (рисунок ниже).

1. С самого начала все кэш линии имеют состояние Invalid.
2. CPU 1 загружает значение по некоему адресу X и соответствующая адресу X кэш-линия приводится в состояние shared.
3. CPU 2 также загружает данные по адресу X и соответствующая этому адресу кэш-линия в кэше процессора #2 также приводится в состояние shared.
4. CPU 1 собирается модифицировать данные в кэш-линии и посылает команду invalidate процессору CPU 2, чтобы тот удалил X из соответствующей кэш-линии и меняет состояние своей кэш-линии на exclusive.
5. CPU 1 модифицирует данные в соответствующей X кэш-линии и изменяет ее состояние на modified.
6. CPU 2 загружает данные по адресу X, но поскольку CPU 1 модифицировал эти данные, надо сперва отправить ему запрос на чтение. CPU 1 по запросу записывает данные

обратно в память и меняет текущее состояние кэш-линии, в которой находится X.

ОПЕРАЦИЯ	CPU #	CPU 1	CPU 2
1		Invalid/-	Invalid/-
2	1	Shared/X	Invalid/-
3	2	Shared/X	Shared/X
4	1	Exclusive/X	Invalid/-
5	1	Modified/X	Invalid/-
6	2	Shared/X	Shared/X

В процессорах, начиная с AMD Athlon, добавился новый статус **Owner** - владелец. Новый статус не только частично меняет смысл остальных состояний, но и внедрен из-за своей эффективности при многопроцессорном устройстве.

**Modified** - у этого кэша присутствует единственная валидная, то есть самая новая информация о данных.

**Owned** - у этого кэша присутствует какая-то копия данных, но только этот кэш может менять кэш-строку. Если изменения действительно произведены, то кэш должен оповестить все остальные кэши, содержащие копию подобной кэш-строки, о том, что данные изменились. Остальные кэши должны также обновить данные в свой кэш-строке. Что это дает? Во-первых, выигрыш в скорости обмена. Появляется возможность перемещать модифицированные данные по разным кэшам, при этом, нет необходимости постоянно обновлять состояние основной памяти. Кэш-строка данного кэша может перейти в состояние **Modified**, если будут внесены изменения. Но важно, чтобы в этот момент кэш-строки всех других кэшей, которые Sharing эту строку перешли в Invalid. Возможна и такая, модификация, когда кэш-контроллер, тот что общий для всех,

может понять, что строка вообще-то используется везде, но была модифицирована каким-то кэшем. Тогда он поставит временно все копии в **Invalid**, загрузит изменения в основную память, вернет всем тем кэшам, что использовали данные статус, **Sharing** (обновлять данные уже не надо, так как это сделал **Owned**).

**Exclusive** - у данного кэша присутствует единственная копия данных и кэш-строка, которая их содержит “чистая”.

**Shared** - означает, что строка может присутствовать в других кэшах. Важное отличие в том, что строка то может быть модифицирована каким-либо другим кэшем, но при этом все равно иметь установленным этот флаг. И если это так, то значит есть кэш, у кого есть кэш-строка в состоянии **Owned**. В противном случае, информация соответствует последней версии.

**Invalid** - строка отсутствует в кэше. Сперва надо выгрузить данные из оперативной памяти, прежде, чем производить запрос.

Последнее, о чем можно сказать, так это о сущности того, что хранится в кэше - данные или код. Выделяют два типа - *Гарвардскую и Принстонскую* архитектуры (об этом подробно должно быть написано в билете 15). Но если коротко, то в первой данные и код хранятся отдельно (Например, Mark-1), а во второй все вместе. В зависимости от ответа на этот вопрос, как хранятся данные, может меняться набор допустимых флагов. Кэш кода может содержать только два состояния - **Shared** и **Invalid**. Остальные не разрешены, так как модификация кода недопустима. Если конечно, речь не о самомодифицирующемся коде. (Пьянзин разрешил не говорить про детали).



## 29. Архитектура SPARK

**SPARC** (**S**calable **P**rocessor **A**rchitecture — масштабируемая архитектура процессора) — архитектура RISC-микропроцессоров, первоначально разработанная в 1985 году компанией Sun Microsystems.

Архитектура системы команд SPARC опубликована как стандарт IEEE 1754—1994

Разработчик - SunMicrosystems

Разрядность - 64-bit (32 → 64)

Представлена - 1985

Архитектура RISC

Тип Регистр-Регистр

Кодирование СК - Фиксированное

Переходы - Флаги Условий

Порядок байтов - Bi (Big → Bi)

Размер Страницы - 8 KiB

Регистры общего назначения - 31 (G0 = константа ноль; не глобальные регистры)

Регистры Вещественные - 32

По восьмой версии стандарта :

- линейное 32-разрядное адресное пространство, т.е. пронумерованная последовательность ячеек памяти от минимального номера до максимального (0x00000000...0xFFFFFFFF – адресация до 4 Гбайт оперативной памяти);
- небольшое количество простых форматов 32-разрядных команд. Все команды в памяти выровнены по границе 32-разрядных слов. Имеется всего три базовых формата команд, в которых поля кода операции и регистровых операндов всегда находятся в одних и тех же разрядах. Доступ к памяти и ввод/вывод могут осуществляться только командами чтения/записи;
- небольшое количество способов адресации – адрес по памяти вычисляется либо как «регистр + регистр», либо как «регистр + непосредственное значение»;
- трёхадресная регистровая команда – команды большей частью выполняют действия с двумя операндами (двумя регистрами или одним регистром и константой), помещая результат в третий регистр;
- отложенная передача управления – процессор всегда выбирает команду, следующую за командой отложенной передачи управления. Эта команда может быть выполнена или не выполнена в зависимости от состояния «аннулирующего» разряда в команде передачи управления;
- быстрые обработчики прерываний – прерывания собраны в линейную таблицу, их генерация приводит к созданию в регистровом файле нового регистрового окна.
- тегированные команды – команды тегированного сложения / вычитания рассматривают два младших разряда своих операндов в качестве тегов;
- команды межпроцессорной синхронизации – одна команда выполняет непрерываемую операцию «чтения и последующей записи», другая – непрерываемый «обмен содержимого регистра и памяти».
- 

В девятой версии принципиальное отличие:

- Линейное 64-разрядное адресное пространство (позволяет адресовать гипотетически до 16 Эбайт, фактически до 256 Тбайт оперативной памяти).

См. больше здесь

[http://www.elbrus.ru/arhitektura\\_sparc](http://www.elbrus.ru/arhitektura_sparc)



## 30. Характеристики машинных команд

**Машинная команда** - команда, которую исполняет непосредственно процессор. Имеет смысл рассматривать следующие характеристики машинных команд: формат, структура, адресность, способы адресации, функционал, ограничения по доступу. Рассмотрим, что они означают.

### Формат, структура:

Определяют правила кодирования машинных команд. Вообще говоря, понятия “формат” и “структура” довольно близки. Формат определяет, из каких полей состоит команда, а также указывает, какие биты какому полю принадлежат. Структура же определяет функциональное назначение полей.

В общем виде структура машинной команды выглядит так:

Операционная\_часть    Адресная\_часть

В операционной части записан код операции, которую требуется исполнить. Как правило, количество различных кодов около сотни, поэтому на операционную часть выделяют один байт.

Адресная часть содержит адреса операндов, а также адрес, по которому будет помещен результат операции.

### Адресность:

Определяет количество адресов в адресной части, которое может быть от 0 до 4.

4-адресная команда имеет следующий формат:

Код A1 A2 A3 A4

A1, A2 - операнды

A3 - адрес, по которому будет помещен результат

A4 - адрес следующей для исполнения команды

Такой формат применялся только в первых моделях вычислительных машин, в дальнейшем нужда в A4 отпала, так как команды теперь исполняются по порядку следования в памяти.

Соответственно, появились 3-адресные наборы команд в формате Код A1 A2 A3.

Далее тенденция к сокращению адресной части продолжилась:

2-адресные команды принимают только операнды, а результат помещают в фиксированный регистр, например, на место одного из операндов.

1-адресные команды принимают адрес одного операнда, а второй считывают из специального регистра. В тот же регистр записывается результат. Такой регистр называют *аккумулятором*.

Наконец, безадресные команды фиксируют адреса обоих операндов и результата.

### Способы адресации:

Способы, которыми могут вычисляться операнды для команды.

- Регистровая адресация - операнд находится в регистре
- Непосредственная адресация - операнд указывается прямо в команде
- Прямая адресация - указывается адрес операнда в памяти

- Косвенная адресация - указывается адрес, по которому хранится адрес операнда [мы встроили адрес тебе в адрес, чтобы ты мог переходить по адресу, пока переходишь по адресу]

Существуют и другие типы адресации. Операнд может задаваться прямо в коде команды (адресом или значением, их называют *подразумеваемыми*), может получаться как сумма заданного в адресной части и фиксированного смещения (обычно хранящегося в специальном регистре).

#### Функциональные типы:

- Арифметика и логика
- Перемещение данных
- Команды управления (условные переходы, вызовы процедур и др.)
- Команды ввода-вывода
- Вспомогательные команды (например, остановка выполнения программы)

#### Ограничения по доступу:

Как вы, возможно, помните из курса ОС, у процессора есть два режима работы: режим пользователя и режим ядра. Соответственно, команды делятся на категории по уровням доступа:

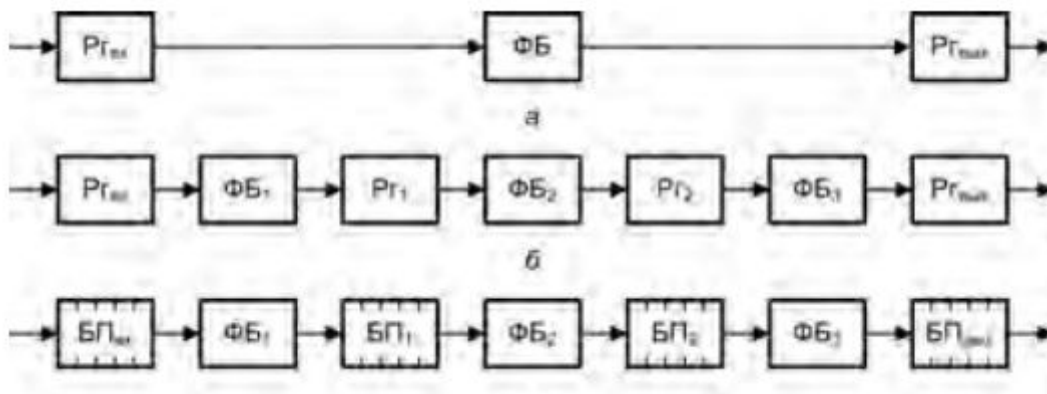
- Общего пользования
- Контекстно доступные
- Максимально привилегированные

## 31. Архитектура системы команд. Три основные классификации, примеры.

В билете номер 6 уже было введено понятие *Конвейера* и суперскалярных систем. В билете номер 20 также были рассмотрены эти понятия. Тем не менее, для хорошего понимания сути всех видов архитектур, которые будут рассмотрены ниже, следует еще раз осветить данную идею.

Введем очень неточное, но интуитивное определение функционального блока. **Функциональным блоком** может называться любая совокупность устройств процессора, которые принимают на вход регистр, производят с содержимым регистра необходимые операции, записывают результат в выходной регистр.

Теперь можно ввести идею конвейера. Легче всего продемонстрировать рисунком:



Рассмотрим классический вариант устройства простейшей ЭВМ (на рисунке а). В функциональный блок заносится содержимое входного регистра, производятся необходимые манипуляции и на выходе получаем содержимое в другом регистре. Если так подумать, то эта операция делается всегда за ***T***, если обозначить за ***T*** - максимальное время обработки в функциональном блоке. То есть новые данные могут быть занесены в первый регистр не ранее, чем через ***T***.

Рассмотрим второй рисунок, где вместо одного функционального блока уже присутствует аж целых три, все они образуют последовательность с промежуточными регистрами. По-прежнему, время перехода из входного регистра в выходной равно  $T$ . Однако если мы добьемся того, что каждый ФБ (далее сокращение от функционального блока), выполняет свои операции в лучшем случае за  $T/3$ , то получим, что данные на входной регистр могут подаваться в 3 раза чаще! Промежуточные регистры нужны не практике в силу того, что ФБ не могут быть максимально сбалансированы во времени, и  $T/3$  - идеальный недостижимый вариант.

На практике применяется и вовсе 3 вариант с изображения. Используются не промежуточные регистры, а промежуточные **буферы памяти**, которые способны хранить множество полученных с некоторого фб результатов одновременно и подавать их на вход следующему по принципу FIFO. Во многих источниках ФБ именуются как ступени конвейера, что еще более интуитивно и понятно при попытке визуализировать работу процессора на конвейерной архитектуре.

Прежде, чем приступить к теме билета, сформулируем ряд терминов:

**Архитектура набора (системы) команд** - промежуточный логический слой между аппаратной частью и системным программным обеспечением.



Теперь можно приступить и к самой классификации (не забудьте при ответе нарисовать или пересказать эту таблицу!). В других вопросах уже освещались детали некоторых архитектур системы команд. Основных, конечно, три: CISC, RISC, VLIW. К перечисленным можно добавить EPIC. Тем не менее, эти названия еще не отражают полной сути того, какие модели заложены в их реализацию. Основная тема вопроса - именно разобрать способы реализации той или иной архитектуры. На рисунке, что выше, допущена неточность (сам Пьянзин об этом упоминал), конкретно в блоке, посвященном стековым архитектурам. Стрелка идет не сразу в ROSC, а через некоторые промежуточные модификации. Они будут упомянуты ниже.

Начнем классификацию по тому же принципу, как это делал Пьянзин. В билете рассмотрим три(четыре, если учитывать историю) вида классификации: семантическую, по месту хранения операндов и по адресности команд (этот пункт относится к более общей теме формата команд.)

## История и Структурно-семантическая классификация.

Суть современных языков программирования высокого уровня (ЯВУ) - облегчить сам процесс программирования. Из-за этого получается, что сложные операторы ЯВУ очень сильно отличаются от простых машинных команд. Следствием такого несоответствия становится недостаточно эффективное выполнение программ на машине. Проблема получила название **семантического разрыва**. Для ее разрешения разработчики выбирают один из трех типов архитектуры команд:

- **CISC архитектура** (Complex Instruction Set Computer) - архитектура с полным набором команд. Про детали устройства в билете 6. В данной архитектуре семантический разрыв устраняется за счет **расширения системы команд**, дополнения ее сложными командами, аналогичными тем, что реализованы в ЯВУ.

Основоположниками архитектуры являются, согласно Цилькеру и википедии, компания IBM, которая начала применять данный подход, начиная с **семейства** машин IBM 360 (это линейка машин, а не конкретная VM!). Сегодня IBM продолжает развитие этой архитектуры, но уже в мощных современных универсальных машинах (мэйнфреймах). В 6 билете рассмотрена краткая таблица сравнения устройства команд в двух основных архитектурах. Основная проблема машин на этой архитектуре - сложность проектирования. Большая нагрузка ложится на устройство управления процессора. Еще раз упомянем про основные моменты и дадим краткие пояснения:

- ❑ множество форматов команд различной разрядности (ввиду разной сложности)
- ❑ разнообразие способов адресации операндов(на самом деле следующий пункт проясняет эту мысль, иногда нужно выгоднее сразу иметь физический адрес, нежели смотреть в таблицу адресов, вычислять линейный, и только потом физический)
- ❑ наличие команд, где обработка совмещается с обращением к памяти (ввиду того, что некоторые команды требуют работы с массивами)



- **RISC-архитектура** (Reduced Instruction Set Computer) - архитектура с сокращенным набором команд. Официально принято считать, что термин RISC впервые был использован Д. Паттерсоном и Д. Дитцелем в 1980 году, хотя Джон Кок еще в начале 1970-х установил, что прогресс в области компиляторов достиг той точки, когда можно упростить набор команд процессора и возложить на компилятор большую часть работы, которая ранее выполнялась самой аппаратурой, тем самым обосновав возможность реализации концепции RISC (читайте подробно в 6 билете, где ДОЛЖНО БЫТЬ НАПИСАНО ПРО ВАЖНОСТЬ РЕГИСТРОВ ОБЩЕГО НАЗНАЧЕНИЯ!!!). Кратко, можно резюмировать следующее:
  - ❑ Уменьшено количество форматов команд и способов указания адресов операндов
  - ❑ Реализация сложных команд за счет последовательности из простых, но быстрых RISC-команд не менее эффективна, чем аппаратный вариант сложных команд в CISC-архитектуре.
- **VLIW** (Very Long Instruction Word) - архитектура с командными словами сверхбольшой длины. Про детали должно быть написано в билете 24. Но если коротко, то концепция VLIW базируется на RISC-архитектуре, но в ней несколько простых RISC-команд объединяются в одну сверхдлинную команду и выполняются параллельно. Параллелизм достигается на этапе компиляции (как именно - 24 билет).

Можно представить все вышеперечисленное в краткой таблице, но будьте готовы рассказать Пьянзину, почему она верна:

Характеристика	CISC	RISC	VLIW
Длина команды	Варьируется	Единая	Единая
Расположение полей в команде	Варьируется	Неизменное	Неизменное
Количество регистров	Несколько (часто специализированных)	Много регистров общего назначения	Много регистров общего назначения
Доступ к памяти	Может выполняться как часть команд различных типов	Выполняется только специальными командами	Выполняется только специальными командами

Следующим принципом, по которому можно классифицировать архитектуру системы команд - по месту хранения операндов.

Пьянзин не давал определение того, что есть операнд и явно дал понять, что операндом может быть не обязательно логический или просто числовой элемент.

Для определенности. Машинные команды оперируют данными, которые в этом случае принято называть ***операндами***. К наиболее общим (базовым) типам операндов можно отнести: адреса, числа, символы и логические данные. Помимо них вычислительная машина обеспечивает обработку и более сложных информационных единиц: графических изображений, аудио-, видео- и анимационной информации (это к теме про MMX и SSE).

### **Классификация архитектур команд по месту хранения операндов и осуществлению доступа к ним.**

Выделяют следующие виды архитектур, рассмотренных с этих позиций:

- 1) Стековая
- 2) Аккумуляторная
- 3) Регистровая
- 4) с выделенным(ограниченным) доступом к памяти (load/store).

Начнем по порядку.

#### **Стековая архитектура.**

Подробного объяснения, как устроена эта структура данных здесь приводиться не будет ввиду предположения, что это всем известно.

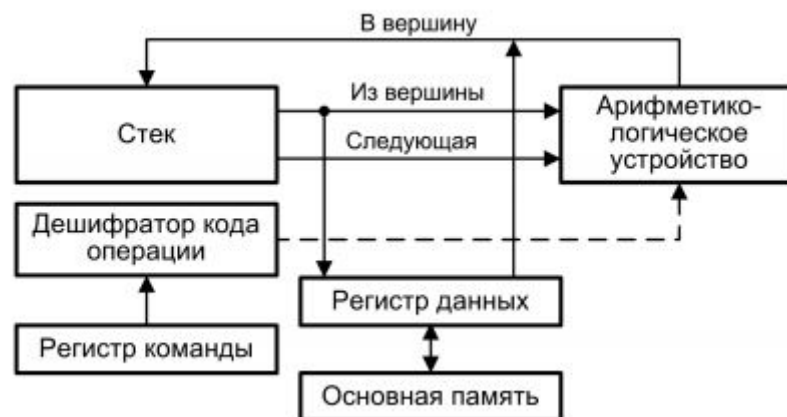
Неформально, стэк образует множество логически взаимосвязанных ячеек по принципу LIFO.

Гораздо более конструктивно привести пример того, как такая архитектура может быть реализована в вычислительной машине.

---

Одна из первых реальных машин, где был реализован такой подход - [B5500](#) - по ссылке исчерпывающий документ, рассказывающий про все детали устройства машины

---



В прикрепленной ссылке про B5500 рассказывается, что несколько элементов стека (верхушка), реализованы в виде дополнительных регистров (более того, если речь идет о числах с плавающей точкой двойной точности, то для этого вводятся еще два регистра, конкретно в последней модификации B6500 ).

Как именно реализован стек, то есть какими средствами - неоднозначный вопрос. Цилькер, из рекомендуемой литературы, приводит такую организацию: в основной памяти отводится участок с наибольшими адресами, а сам стек расширяется в сторону уменьшения (то же самое можно прочесть и в билете 10, в разделе про стек). Внутри процессора располагаются специальные регистры - два регистра для двух первых элементов и, вероятно, еще несколько для следующих. Но остальная часть стека может располагаться (теоретически) даже и на жестком диске.

Сами операции pop и push могут быть реализованы либо на уровне машинных команд, либо на уровне устройства управления (так было в B5500).

Информация **попадает** на вершину либо из АЛУ, либо из основной памяти. Для **записи** в стек содержимого ячейки памяти с текущим

адресом выполняется команда `push [адрес]`, по которой информация считывается из ячейки памяти, заносится в регистр данных, а затем проталкивается в стек. Результат операции из АЛУ заносится в вершину стека автоматически.

**Сохранение** содержимого вершины стека в ячейке памяти с адресом, который лежит на вершине, этот адрес также и в регистре данных, производится командой `pop [адрес]`. По этой команде содержимое верхней ячейки стека подается на шину, с которой и производится запись в ячейку `[адрес]`, после чего все содержимое стека проталкивается на одну позицию вверх.

Для выполнения арифметической или логической операции на вход АЛУ подается информация, считанная из двух верхних ячеек стека (указатели на них лежат в специальных регистрах, поэтому это еще очень быстро). Результат операции заталкивается в вершину стека. Возможен вариант, когда результат сразу же переписывается в память с помощью автоматически выполняемой операции `pop [адрес]`

Одно из достоинств стековой архитектуры в том, что она проста в реализации. Однако, в силу того, что было упомянуто выше, в стековой архитектуре нет произвольного доступа к памяти, а это может быть узким местом в повышении производительности системы. К тому же, компилятору не очень просто создать оптимальный код программы (тоже из-за отсутствия доступа ко всем ячейкам)

---

Среди современных ВМ со стековой АСК особо следует отметить стековую машину IGNITE компании Patriot Scientist, которую ее авторы считают представителем нового вида архитектуры с **безоперандным** набором команд. Для обозначения таких ВМ они предлагают аббревиатуру ROSC (Removed Operand Set Computer).

---

Аккумуляторная архитектура.

Исторически самая первая. Для хранения одного из операндов отводится специальный регистр внутри процессора - **аккумулятор**. В этот регистр будет занесен и результат операции. Так как адрес одного из операндов предопределен (аккумулятор), то в командах обработки нужно указать только адрес - второго. Когда АЛУ посчитает результат того, что хранилось в аккумуляторе и того, что было подано в регистр данных, результат должен быть занесен в аккумулятор. Если результат не является операндом для следующей команды, то содержимое аккумулятора записывается в память.

Для аккумулятора также предусмотрены две команды - load/store. **load x** - информация считывается из ячейки памяти x, выход памяти подключается ко входам аккумулятора, и происходит занесение считанных данных в аккумулятор. **store x** - запись содержимого аккумулятора в ячейку x, при выполнении которой выходы аккумулятора подключаются к шине, после чего информация с шины записывается в память.

Существенным недостатком этой архитектуры является наличие одного регистра данных, который, понятно, требует постоянного обращения в основную память, что многократно снижает производительность.

---

Архитектура базе аккумулятора была популярна в ранних ВМ, таких, например, как IBM 7090, DEC PDP-8

---

Мы подошли к **регистровой архитектуре** - самой распространенной на сегодняшний момент.

Регистровая архитектура.

Возьмем за основу тот факт, что разрядность регистра совпадает с длиной машинного слова (конкретно в MIPS это 32 или 64 бита). К любому

регистру можно обратиться явно указав его номер (это для программистов на ассемблере).

**Регистры общего назначения** - такие регистры, которые предназначены для хранения операндов арифметико-логических инструкций, а также адресов или отдельных компонентов адресов ячеек памяти. Подробно - [тут](#).

В CISC архитектуре, как известно, малое количество регистров общего назначения, 8-32, поэтому достаточно всего лишь пяти разрядов, чтобы определить номер конкретного регистра. Поэтому в адресной части команд обработки допустимо одновременно указать номера сразу нескольких регистров (конкретно трех -2 регистра операнда, 1 регистр результата). в RISC используется большее число регистров этого вида.

Особенностью регистровой архитектуры является то, что операнды могут быть как в основной памяти (в CISC), так и в самих регистрах. Отсюда получаем 3 комбинации: **регистр-регистр, регистр-память, память-память**. Легче всего оценить недостатки и преимущества в виде таблицы из Цилькера, вроде. Единственное, отмечу, что, что в паре (m,n), m - число операндов в памяти, а n - общее число операндов в команде арифметической или логической обработки .

Вариант	Достоинства	Недостатки
Регистр-регистр (0, 3)	Простота реализации, фиксированная длина команд, простая модель формирования объектного кода при компиляции программ, возможность выполнения всех команд за одинаковое количество тактов	Большая длина объектного кода, из-за фиксированной длины команд часть разрядов в коротких командах не используется
Регистр-память (1, 2)	Данные могут быть доступны без загрузки в регистры процессора, простота кодирования команд, объектный код получается достаточно компактным	Потеря одного из операндов при записи результата, длинное поле адреса памяти в коде команды сокращает место под номер регистра, что ограничивает общее число РОН. CPI зависит от места размещения операнда
Память-память (3, 3)	Компактность объектного кода, малая потребность в регистрах для хранения промежуточных данных	Разнообразие форматов команд и времени их исполнения, низкое быстродействие из-за обращения к памяти

Данные полезны уже непосредственно программистам.

Остановимся на одном моменте: операция загрузки регистров из памяти и сохранении их содержимого в памяти. В принципе, все так же, как с **аккумулятором**. Отличие только в этапе выбора нужного регистра. Для этого имеется два специальных регистра - селекторные регистры каждого операнда (второй селекторный регистр является еще и селектором результата). Как это устроено? Достаточно просто:

1. в регистр команды заносится команда.
2. регистр команды пропускается через дешифратор кода операции, чтобы выявить адреса операндов если операция алгебраическая, то сама операция идет сразу в АЛУ.
3. далее каждый адрес операнда идет в селекторы регистров первого и второго операнда.
4. селекторы идут в **массив регистров общего назначения**, где определяются уже сами регистры.
5. полученные два регистра проходят **через три шины** и поступают в АЛУ.
6. АЛУ решает что делать дальше: вернуть результат уже в регистр данных или обратно в массив регистров.

Не случайно выделено, что между АЛУ и массивом регистров три шины. Две из трех шин, расположенных между массивом регистров и АЛУ, обеспечивают передачу в арифметико-логическое устройство операндов, хранящихся в регистрах общего назначения или ячейках памяти. Третья служит для занесения результата в выделенный для этого регистр или ячейку памяти. Эти же шины позволяют загрузить в регистры содержимое ячеек основной памяти и сохранить в ОП информацию, находящуюся в регистрах общего назначения.

Важным плюсом является **компактность машинного кода** (из-за не слишком большого количества регистров, что в CISC, что в RISC), причем в CISC это не совсем справедливо (неверно для длинных команд, которые и не хранятся в регистрах общего назначения), а также **высокая скорость вычислений** (число обращений к памяти существенно уменьшено).

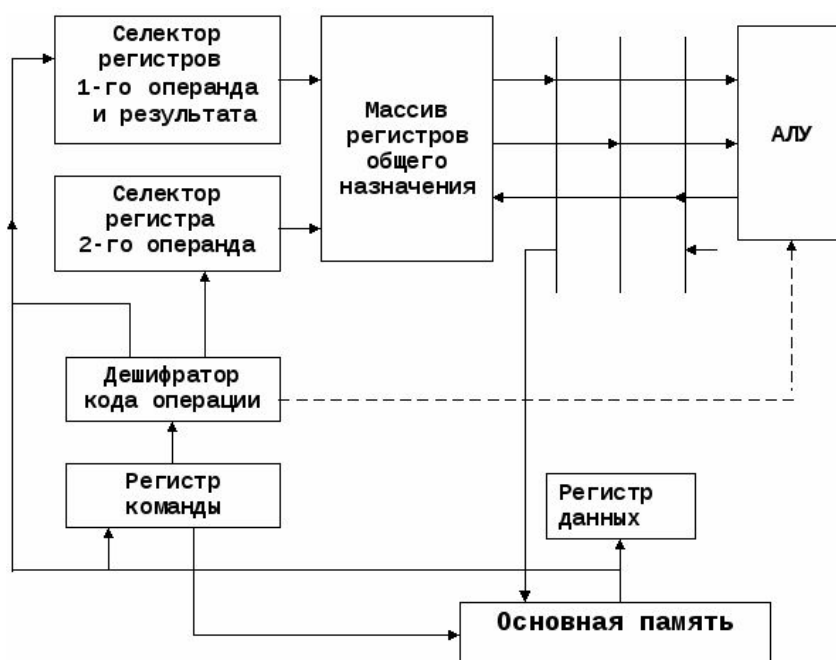
## Архитектура с выделенным доступом к памяти (Load/Store Architecture).

В архитектуре с выделенным доступом к памяти обращение к основной памяти возможно только с помощью двух специальных команд: **load** и **store**.

Команда **load** обеспечивает считывание значения из основной памяти, и занесение его в регистр процессора, или, что то же самое, в один из регистров общего назначения (в команде обычно указывается адрес ячейки памяти и номер регистра).

Пересылка информации в противоположном направлении производится командой **store**. Операнды во всех командах обработки информации находятся в регистрах общего назначения. Результат операции также заносится в регистр.

Общая идея и принцип конструирования процессора совпадает с тем, что рассмотрен в регистровой архитектуре: для регистров общего назначения отводится массив, для выбора регистров - два селектора; три шины для связи массива и АЛУ. Однако тут есть тонкий момент в устройстве шин, расположенных между массивом регистров общего назначения и АЛУ.





Две из трех шин, расположенных между массивом и АЛУ обеспечивают передачу в АЛУ операндов, хранящихся в двух регистрах общего назначения. Третья служит для занесения результата в выделенный для этого регистр. Эти же шины позволяют загрузить в регистры содержимое ячеек основной памяти и сохранить в ОП информацию, находящуюся в регистрах общего назначения.

Такая архитектура характерна для современных RISC машин. Для следования идеологии, все команды имеют длину 32 бита и трехадресный формат (то есть то, что будет сохранено в операндах, а что является операцией). Для еще более успешного понимания и конкретных примеров того, как устроены команды, почитайте главу 2 из книги Паттерсона и Хенесси, в которой рассказывается про MIPS архитектуру (она на базе RISC).

---

Реальными примерами архитектуры команд такого типа могут служить IBMRS/6000, SunSPARC, MIPS R4000, DECAlpha. Про SPARC и MIPS можно и вовсе прочитать в билетах номер 27 и 29.

---

Дополнительно можно рассмотреть еще одну классификацию команд, которая уже не имеет отношение к конкретной модели архитектуры, а устанавливает общие принципы работы вычислительной машины.

### **Классификация машинных команд по структуре и формату, и функциям.**

Если это еще не было понятно, то машинная команда представляет собой код, определяющий операцию вычислительной машины и данные, которые участвуют в операции.

В общем виде машинная команда представляет собой **Код Операции** и **Адресную часть**.

**Операционная часть** - содержит код, задающий вид операции (сложение, умножение, передача и т.д.). **Адресная часть** - содержит информацию об адресах операндов, результата операции и следующей команды.

**Структура команды** - определяется составом, назначением и расположением полей в команде. **Формат команды** – это структура команды с разметкой номеров разрядов, определяющих границы отдельных полей команды.

По функциям выделяют следующие виды команд:

- арифметические операции над числами с фиксированной или плавающей точкой;
- команды двоично-десятичной арифметики;
- логические (поразрядные) операции;
- пересылка операндов;
- операции ввода-вывода;
- передача управления;
- управление работой центрального процессора.

Рассмотри разные виды адресности. Пьянзин на паре выделил 5 типов: от 0 до 4. Рассмотрим все виды.

*1) Четырехадресная структура.*



Содержит наиболее полную информацию о выполняемой операции, так как содержит поле кода операции и четыре адреса для указания ячеек памяти двух операндов, ячейки результата операции, и ячейки, содержащей адрес следующей команды. Такой формат машинной команды еще называется принудительным - Таненбаум. Использовался в первых моделях вычислительных машин, имеющих небольшое число команд и очень незначительный объем основной памяти, ввиду того, что

длина такой команды зависит от разрядности адресов операндов и результата.

## **2) *Трехадресная структура.***

Не содержит информации об адрес следующей команды (только КОП, операнд 1, операнд 2, результат). Обычно используется в машинах, построенных по принципу: если адрес первого операнда  $A$ , размер поля  $B$ , тогда адрес второго операнда  $A+B$ . Такой порядок называется **естественным**.

## **3) *Двухадресная структура.***

Содержит только КОП, адрес операнда 1 и адрес операнда 2. В этом случае результат операции записывается либо в первый, либо второй операнд.

## **4) *Одноадресная структура.***

Содержит только КОП и адрес первого операнда. При этом один из операндов и результат операции размещаются в одном фиксированном регистре. Выделенный для этой цели внутренний регистр процессора получил название аккумулятора. Адрес другого операнда указывается в команде.

## **5) *Безадресная структура.***

Содержится только КОП. Фиксирует адреса обоих операндов и результата операции, например при работе со стековой памятью.

## **32. Сегментная модель памяти. Страничная модель памяти**

Два основных вида организации виртуальной памяти – сегментная и страничная организация.

При сегментной организации вся виртуальная память, используемая программой, разбивается на части, называемые сегментами. Это разбиение выполняется либо самим программистом (если он программирует на языке ассемблера), либо компилятором используемого языка программирования.

Размеры сегментов могут быть различными, но в пределах максимального размера, используемого в данной архитектуре. Разбиение обычно производится на логически осмысленные части, такие, как сегмент данных, сегмент кода, сегмент стека и т.п. Большая программа может содержать несколько сегментов одного типа, например, несколько сегментов кода или данных. Таким образом, при сегментной организации у программы нет единого линейного адресного пространства. Виртуальный адрес состоит из двух частей: селектора сегмента и смещения от начала сегмента.

Селектор сегмента представляет некоторое число, которое обычно является индексом в таблице сегментов данного процесса. Такая таблица содержит для каждого сегмента его размер, режим доступа (только чтение или возможна запись), флаг присутствия сегмента в памяти. Если сегмент находится в памяти, то в таблице хранится его базовый адрес (адрес физической памяти, соответствующий началу сегмента). Отсутствие сегмента означает, что его данные временно вытеснены на диск и хранятся в файле подкачки (swarfile). В кодах программы селектор сегмента может либо явно присутствовать как часть адреса, либо подразумеваться в зависимости от смысла конкретного адреса. Например, для адресов выполняемых команд используется селектор текущего сегмента команд, а для адресов операндов – селектор текущего сегмента данных. При каждом обращении к виртуальному адресу аппаратными средствами выполняется преобразование пары «сегмент : смещение» в физический адрес. Селектор сегмента используется для доступа к соответствующей записи таблицы сегментов. Если данный сегмент присутствует в памяти, то его базовый адрес, прочитанный в таблице, складывается со смещением из виртуального адреса.

Результат сложения представляет собой физический адрес, по которому и происходит обращение к памяти. Если сегмент отсутствует в памяти, то происходит прерывание.

Обработывая его, система должна подгрузить сегмент с диска на свободное место в памяти, записать его базовый адрес в таблицу сегментов и затем повторить команду, вызвавшую прерывание. Поскольку сегменты имеют различные размеры, то в ходе работы системы, сопровождающейся многократной загрузкой и выгрузкой сегментов, возникает эффект фрагментации памяти. Во всех случаях причиной фрагментации является многократное занятие и освобождение областей различного размера.

Для борьбы с фрагментацией можно время от времени производить дефрагментацию, т.е. перемещение всех сегментов, находящихся в памяти, на новые места, без «дырок» в памяти между сегментами. При этом, однако, требуется, чтобы

система откорректировала таблицы сегментов всех тех процессов, сегменты которых переместились в физической памяти. Кроме того, перемещение сегментов занимает ощутимое время, поэтому оно недопустимо для сегментов, содержащих, например, обработчики прерываний, которые должны срабатывать очень быстро. Чтобы избежать этих проблем, в некоторых системах сегменты могут находиться в одном из двух состояний: фиксированный сегмент не должен перемещаться в памяти; перемещаемый сегмент может перемещаться системой, однако программа не может обращаться к адресам в таком сегменте, поскольку его местоположение не определено. Чтобы работать с данными в перемещаемом сегменте, программа должна предварительно временно зафиксировать его вызовом специальной системной функции. При этом ОС определяет текущее местоположение сегмента и корректирует его базовый адрес в таблице сегментов процесса. Если же программа в течение некоторого времени не планирует обращаться к данному сегменту, то его следует расфиксировать, поскольку чем больше фиксированных сегментов в системе, тем менее эффективна будет дефрагментация. При переключении текущего процесса все, что должна сделать система в отношении памяти, заключается в замене таблицы сегментов. Для этого либо в специальный системный регистр записывается адрес таблицы сегментов текущего процесса, либо, если аппаратура допускает только одну таблицу сегментов, ее содержимое должно быть перезаписано так, чтобы соответствовать новому работающему процессу.

Страничная память. Эта форма организации виртуальной памяти во многом похожа на сегментную. Основные различия заключаются в том, что все страницы, в отличие от сегментов, имеют одинаковые размеры, а разбиение виртуального адресного пространства процесса на страницы выполняется системой автоматически. Типичный размер страницы – несколько килобайт. Для процессоров Pentium, например, страница равна 4 Кб. Все виртуальные адреса одного процесса относятся к единому линейному пространству. Проще сказать, виртуальный адрес выражается одним числом, от 0 до некоторого максимума.

Старшие разряды двоичного представления этого адреса определяют номер виртуальной страницы, а младшие разряды – смещение от начала страницы. Например, для страниц по 4 Кб смещение занимает 12 младших разрядов адреса. Физическая память также считается разбитой на части, размеры которых совпадают с размером виртуальной страницы. Эти части называются физическими страницами или страничными кадрами (page frames). Таблица страниц процесса по структуре похожа на таблицу сегментов. Для каждой виртуальной страницы она содержит режим доступа, флаг присутствия страницы в памяти, номер страничного кадра, флаг чистоты.

Если страница отсутствует в памяти, ее данные сохраняются в файле подкачки, который в этом случае чаще называют страничным файлом (page file). При переключении текущего процесса система просто изменяет адрес используемой таблицы страниц, тем самым полностью изменяя отображение виртуальных адресов на физические. Страничная организация памяти не может привести к фрагментации, поскольку все страницы одинаковы по размеру, а потому каждая высвобожденная физическая страница может быть затем использована для любой понадобившейся виртуальной страницы. Размер пространства виртуальных адресов каждого процесса может быть огромным, ибо он определяется только разрядностью адреса. Для

32-разрядных процессоров этот размер равен  $2^{32} = 4$  Гб. На самом деле, программа обычно использует лишь небольшую часть своего адресного пространства, не более нескольких десятков или, в крайнем случае, сотен мегабайт. Только эти используемые страницы и должны быть отображены на физическую память. Тем не менее, суммарный объем страниц, используемых всеми процессами в системе, обычно превосходит объем имеющейся физической памяти, поэтому использование страничного файла становится неизбежным. Управление замещением страниц в физической памяти в современных РС строится по принципу загрузки по требованию (demand paging). Это означает следующее. Когда программа только лишь планирует использование определенной области виртуальной памяти (например, для хранения массива переменных, описанного в программе), соответствующие виртуальные страницы помечаются в таблице страниц как существующие, но находящиеся в данный момент на диске.

В некоторых системах при этом за виртуальной страницей действительно закрепляются конкретные блоки в страничном файле, хотя из соображений экономии дисковой памяти это можно сделать позже, когда реально потребуются записать страницу на диск. Выделение страниц физической памяти не выполняется до тех пор, пока программа не обратится к одной из ячеек виртуальной страницы. Недостатком страничной организации является то, что при большом объеме виртуального адресного пространства сама таблица страниц должна быть очень большой. При размере страницы 4 Кб и адресном пространстве 4 Гб таблица должна содержать миллион записей! Однако вряд ли программа процесса постоянно использует весь огромный диапазон адресов. Как правило, на каждом интервале времени интенсивно используются только некоторые части таблицы страниц (это еще одно проявление локальности ссылок). Желательно иметь возможность вытеснять на диск временно неиспользуемые части таблицы страниц. Такая возможность в современных процессорах обеспечивается использованием более сложной, двухуровневой схемы страничной адресации.

В этой схеме все адресное пространство делится на разделы равной величины, каждый из которых описывается отдельной небольшой таблицей страниц. Имеется также каталог таблиц страниц, который описывает текущее состояние каждой таблицы точно так же, как сама таблица страниц описывает состояние страниц памяти. Те таблицы страниц, которые долго не используются, вытесняются на диск и соответствующим образом помечаются в каталоге. Виртуальный адрес делится не на две, а на три части. Старшие разряды адреса указывают позицию таблицы в каталоге, средние разряды – позицию страницы в таблице, младшие – смещение адреса от начала страницы.

## 33. Режимы работы процессора.

### А. Реальный режим.

После инициализации (системного сброса) микропроцессор (МП) находится в реальном режиме.

В реальном режиме МП работает как очень быстрый 8086 с возможностью использования 32-битных расширений.

Механизм адресации, размеры памяти и обработка прерываний МП 8086 полностью совпадают с аналогичными функциями других МП IA-32 в реальном режиме.

В отличие от 8086, остальные члены семейства IA-32 в определенных ситуациях генерируют исключения, например, при превышении предела сегмента, который для всех сегментов в реальном режиме равен 0FFFFh.

Имеется две фиксированные области в памяти, которые резервируются в режиме реальной адресации: область инициализации системы и область таблицы прерываний.

Ячейки от 00000h до 003FFh резервируются для векторов прерываний. Каждое из 256 возможных прерываний имеет зарезервированный 4-байтовый адрес перехода. Ячейки от FFFFFFF0h до FFFFFFFFh резервируются для инициализации системы.

### В.

## 35. Физическая организация DRAM.

Ключевые слова: Матрица ячеек, интерфейсная «обвязка», регенерация, синхронизация.

Для начала, можно рассмотреть **абстрактную иерархию памяти в операционных системах**: Пользовательская программа -> Библиотеки управления памятью -> Менеджер куч -> Менеджер виртуальной памяти -> CPU -> Кэш-контроллер -> контроллер памяти -> Оперативная память. (может пригодиться вообще)

Основная оперативная память большинства современных компьютеров реализуется на крайне медленных микросхемах динамической памяти - DRAM (ввиду цены).

Ядро микросхемы динамической памяти состоит из множества ячеек, каждая такая ячейка хранит всего 1 бит информации. На уровне физической реализации эти ячейки объединяются в прямоугольную матрицу. Горизонтальные линии (линейки, в некоторой литературе) матрицы называются **строками**. Вертикальные - **столбцами (или страницами)**. Линейки представлены в виде проводников, на пересечении которых находится устройство, состоящее из одного транзистора и одного конденсатора - так называемое “сердце ячейки”.

Конденсатор играет роль хранителя информации (1 бит). Если на обкладках конденсатора отсутствует заряд, это соответствует логическому **0**. Его наличие - **1**. Транзистор в данном случае выполняет роль ключа, то есть удерживает конденсатор от разряда. В состоянии, когда тока нет - транзистор закрыт. Однако, если на некоторую строку матрицы подать сигнал, то он открывается, соединяя обкладку конденсатора с соответствующим ей столбцом.

И что из этого всего следует? Вот сейчас, мы подали ток на строку и получили, что некоторая ячейка “активизировалась”, но эта ячейка, имея теперь заряд, активирует и последующие за ней. То есть теперь мы имеем дело с потоком электронов, устремившихся через открытые транзисторы с



обкладок конденсаторов наружу. Куда, наружу? Такого понятия нет. Есть другое. Чувствительный усилитель (**Sense AMP**) - устройство, подключенное к каждому из столбцов, реагируя на движение электронов, считывает весь столбец (хотя, еще раз, во многих источниках это страницы, так что не забывайте, что это синонимы плиз). Из этого следует важнейший вывод: **СТРАНИЦА - МИНИМАЛЬНАЯ ПОРЦИЯ ОБМЕНА С ЯДРОМ ДИНАМИЧЕСКОЙ ПАМЯТИ.**

**Чтение ячейки деструктивно**, поскольку открыв транзистор, мы считываем заряд с обкладок. Более того, из-за микроскопических размеров и, как следствие, малой ёмкости, записанная информация хранится очень мало, тысячные секунды. Причина - саморазряд конденсатора. То есть динамическая память - память разового действия.

Чтобы избежать потери информации, все ячейки нужно перезаписывать сразу после того, как они были прочитаны (если мы читаем ячейки). Этот процесс называется **refreshing**. Если же речь идёт о второй проблеме (саморазряд конденсатора) - *регенерация*. **Регенерация** - периодическое считывание ячеек с последующей перезаписью.

Регенератор современных микросхем динамической памяти, встроен в них. Перед регенерацией содержимое ячейки копируется в специальный **буфер**. Это предотвращает блокировку доступа к информации(\* - [предлагаю помещать всякие крутые ништяки](#)).

\*

---

В машине ХТ/АТ регенерация осуществлялась раз в 18 мс по таймерному прерыванию через специальный контроллер DMA (Direct Memory Access, прямой доступ к памяти). Попытка отложить эти прерывания искажали данные в оперативной памяти. Более того, это снижало производительность.

---

В процессе развития микросхем, регенерацией ячеек памяти управляли программисты, специальные микросхемы (DMA), контроллер памяти или же сама микросхема (все современные микросхемы занимаются этим сами).

Логически микросхема DRAM представлена в виде двух систем:

1. ядро памяти. Оно реализовано в виде двумерной матрицы.
2. интерфейсной схемы. Она обеспечивает взаимодействие ядра памяти с внешними устройствами (интерфейсная обвязка).

Внешне микросхема представляет собой прямоугольный керамический корпус, который с двух (реже с четырёх) сторон оснащен выводами.

Нас интересуют три вывода. Во-первых, это **линии адреса и линии данных**.

Линии адреса служат для выбора конкретной ячейки памяти. Линии данных - для записи ее содержимого.

#### **Про линии данных:**

Как понять, что именно сейчас от нас требуется - считать содержимое или записать в конкретную ячейку?

Для этого нужен еще один важный вывод микросхемы - WE (Write Enable), Низкий уровень, то есть активный, готовит микросхему к считыванию состояния линий данных и записи полученной информации в соответствующую ячейку.

Высокий уровень, наоборот - заставляет считывать содержимое ячейки и выдавать его в линию данных.

Из сказанного выше следует, что нельзя вести одновременную запись и считывание (всё-таки выход WE один). Это уменьшает габариты.

Меньший размер позволяет добиться более высокой частоты (хотя бы потому, что один сигнал не опережает другой в распространении).

#### **Про линии адреса:**

Ранее упоминались столбцы и строки матрицы ядра памяти. Так вот, они тоже объединены в одну линию - адресную.

Однако есть неоднозначность: что в данный момент находится на адресной линии: столбец или строка? Или она вообще пустует?

Из-за этого нужны еще два дополнительных вывода: RAS (строб адреса строки) и CAS (строб адреса колонки). На обоих выводах поддерживается высокий уровень сигнала, если все остальные сигналы сброшены. Если же производятся какие-то манипуляции, то всё происходит по одинаковому алгоритму.

### ***Процесс считывания содержимого ячеек.***

1. Поступила команда (контроллер получил физический адрес) прочесть содержимое ячейки.
2. Контроллер преобразует физический адрес в пару чисел - номер строки и столбца.
3. Номер строки посылается на адресные линии.
4. RAS переходит в низкий уровень - на линии есть информация, микросхема теперь об этом знает.
5. Микросхема считывает этот адрес(из линии) и подаёт на нужную строку электрический сигнал.
6. Транзисторы, подключенные к этой строке активизируются, выпуская электроны с обкладок конденсатора, всё это идёт на выход sense AMP.
7. чувствительный усилитель декодирует всю строку, преобразуя в набор 0 и 1. Полученная информация теперь хранится в буфере.
8. Наступает маленький момент ожидания, когда микросхема готовится вновь принимать информацию (это, вообще говоря, RAS to CAS delay).
9. Контроллер подает на адресные линии номер колонки (всё ещё RAS to CAS delay, потому что сигнал стабилизируется) и затем, устанавливает CAS вывод в низкое состояние.
10. Микросхема преобразует адрес колонки в смещение внутри буфера. Осталось прочесть и выдать на линии данных (это, вообще говоря, CAS delay).
11. Контроллер считывает состояние линии данных, сбрасывает RAS и CAS.

12. Микросхема тратит некоторое время перезарядку внутренних цепей, а также восстановительную запись строки из буфера (refreshing).
13. Либо все операции закончились, либо на адресные линии приходит новый адрес строки. Задержка между чтением последней ячейки в буфере и подачей номера новой строки - RAS precharge.

## 36. Типы DRAM, схемы пакетных циклов

**Ключевые слова:** FPM, EDO, BEDO, SDRAM, DDR, DDR2, DDR3, тайминги, латентность.

В предыдущем билете вводились такие понятия, как RAS to CAS delay, CAS delay и RAS precharge. До середины 1990-х годов всех вполне устраивали длительности этих задержек, но с ростом производительности процессоров, медленная оперативная память, сконструированная на базе **Conventional DRAM** (обычная DRAM, ее принцип работы в предыдущем билете) становилась объективно большой проблемой.

Первые машины с новой архитектурой памяти (FPM DRAM) - Intel 60, Intel 486DX4 100.

**FPM DRAM** - Fast Page Mode DRAM (Память быстрого страничного режима). Главное отличие, в сравнении с обычной DRAM - поддержка сокращенных адресов. Принцип, на самом деле очень прост, если прочитать алгоритм получения данных из ячейки (в предыдущем билете).

Если очередная запрашиваемая ячейка находится в той же самой строке, что и предыдущая, то ее адрес однозначно определяется одним лишь номером столбца. Тогда понятно, что передача номера строки уже не требуется.

*Далее приведена временная диаграмма, демонстрирующая работу основных типов памяти, рассмотренных в данном вопросе.*

Обычная DRAM устроена таким образом, что после считывания данных, то есть, когда строка и столбец определены и вадана нужна ячейка в линию данных, RAS сигнал сбрасывается, то есть переходит в высокий уровень. После этого начинается подготовка микросхемы к новому циклу обмена.

контроллер FPM удерживает сигнал RAS в низком состоянии при последовательном формировании трех сигналов CAS, избавляясь, тем самым, от постоянной пересылки номера строки.

Если ячейки читаются последовательно, то это колоссальный выигрыш! Ведь обрабатываемая строка (Помните, у нас нет прямого доступа к конкретной ячейке - только к целой строке) находится во внутреннем буфере микросхемы, и обращаться к самой матрице памяти вовсе нет нужды.

Конечно, хаотичные запросы сводят на нет все преимущество и тогда FPM работает в режиме обычной DRAM.

Если очередная запрашиваемая ячейка лежит внутри той строки, что подана RAS, то такая **строка** называется **открытой**. Если же нет, то приходится выдерживать все паузы, что и у обычной DRAM.

Еще несколько терминов. Если запрашиваемая ячейка находится в открытой строке - **попадание на страницу (page hit)**. Противоположная ситуация - **промах (page miss)**. Просто к сведению, FPM существенно сложнее в проектировании именно из-за возможной ситуации промаха, так как появляются дополнительные задержки. В этом случае приходится проектировать архитектуру памяти в соответствии с архитектурой самого вычислительного устройства.

## Тайминги

Раз мы говорим о существенном выигрыше во времени, то логично именно здесь ввести некоторые величины, характеризующие быстродействие. **Латентность** ( то же, что и **тайминг**) - временная задержка сигнала при работе оперативной памяти(любого вида).

Непостоянство времени доступа затрудняет измерение производительности микросхем памяти и их сравнение результатов друг с другом. Какое будет худшее время обращение к ячейке в FPM? Как и простой DRAM :  $RAS\ to\ CAS\ delay + CAS\ delay + RAS\ precharge$  в наносекундах. Лучшее же :  $CAS\ delay$ . Можно рассмотреть,

альтернативно, хаотичное, но не слишком интенсивно обращение к памяти (так, чтобы во время обращения цепь успевала перезаряжаться), тогда будет среднее RAS to CAS delay + CAS delay.

Величины всех этих задержек не очень связаны друг с другом. Поэтому, вообще говоря, оценка производительности требует всех трех показателей. На практике применяются несколько иные величины. **Время доступа** (синоним выражения **время полного доступа**) - то же самое, что и RAS to CAS delay + CAS delay - характеризует время доступа к произвольной ячейке. **Время рабочего цикла** - то же самое, что и CAS delay - характеризует время доступа к последующим ячейкам из уже открытой строки.

Существует много и других таймингов, однако выше перечислены основные. К слову, RAS precharge исключен из учета, хотя во многих источниках он учитывается (*из тех терминов, что Пьянзин просил выписать, этот тайминг все равно исключен*). Просто для информации можно добавить, что сегодня популярен **Row Active Time** - чистое время, в течении которого активна одна строка. Command Rate - определяет время при обмене между контроллером памяти и каким-либо другим модулем памяти (вычисляется, как средняя величина всех обращений).

*Не знаю, как сильно важно это для билета, но тема таймингов обычно идет с таким определением, как **Формула памяти**.*

---

Это чисто ознакомительная информация. В 1990-х значение RAS to CAS delay составляло около 30 нс, CAS delay - 40 нс, RAS precharge - 30 нс. Если частота системной шины 60 МГц (~ 17 нс на такт), то на открытие и доступ к первой ячейке уходило 6 тактов, а на доступ к остальным ячейкам этой страницы - 3 такта. Отсюда появилась широко известная запись 6-3-х-х.

Что дает эта формула? Как рассказал на парах Пьянзин, особой объективности это не добавляет и сравнивать 5-4-х-х с 6-3-х-х все так же бессмысленно. Надо отталкиваться от вопроса, для какой задачи это

лучше. Если идет хаотическое (интенсивное) обращение к ячейкам не из одной страницы, то эти числа вообще нарушатся. будет 5 + RAS precharge для первой, но и 6 + RAS precharge для второй архитектуры. Первая лучше? Нифига подобного. Если на машине работают с потоковыми алгоритмами, то второй вариант предпочтительнее, так как экономия второй цифры во второй архитектуре очевидно, если учесть, что обращения постоянны. Более того, может так получиться (об этом говорил и Пьянзин), что вторая схема будет лучше в обоих случаях - и последовательном, и хаотичном, в виду еще одной величины, которую опускают - RAS precharge. Вдруг, в первом случае он такой медленный, что смысла нет...

---

## EDO DRAM

Следующая историческая разновидность DRAM - **EDO DRAM** ( EDO ~ Extended Data Out), или память с усовершенствованным выходом.

А что, если на микросхему памяти добавить специальный триггер-защелку, который будет удерживать линии данных после снятия сигнала CAS ? Обратитесь к рисунку, вполне видно, что как только сигнал снят, можно ведь сразу же его сбросить и отправить на перезарядку, а линии данных будут удержаны. И пока наблюдается CAS delay, данные уже считаются или все еще будут считываться, но все равно можно приступить к следующему столбцу. То есть наблюдается **параллельность**: сигнал CAS сбрасывается в процессе чтения данных параллельно с перезарядкой внутренних цепей, благодаря чему номер следующего столбца можно подать **до** завершения считывания линий данных. *(кажется, я это объяснил уже раза 3 и все понятно)*

Сразу же виден прирост в производительности на чтении из открытой страницы. Например, при частоте 66 МГц формула лучших микросхем того времени выглядела так: 5-2-х-х.



**BEDO DRAM** (Burst EDO - пакетная EDO DRAM). Существенное изменение в сравнение с родительской EDO было в том, что в микросхему добавили **генератор номера столбца**, тем самым, инженеры ликвидировали CAS delay окончательно (Пьянзин сказал, что это ложь, ликвидировали только *почти*). После обращения к произвольной ячейке BEDO сама, без указаний контроллера увеличивает номер столбца на единицу, не требуя его явной передачи. Длина пакета тоже увеличилась, она стала равной 4. По причине ограниченной разрядности адресного счетчика, под разрядность генератора отвели только 2 бита. Почему именно так - лучше спросите у Пьянзина. Последующие модели с EDO DRAM, типа Intel 80486, вообще считывали пакеты длиной не менее 4, но при этом всегда по степеням 2.

Это все хорошо, но, как и всех предшественниц, у BEDO была проблема - она оставалась до сих пор **асинхронной**. Это накладывает ограничения на максимальную тактовую частоту. Ну и просто рассуждая логически. Вот есть контроллер DRAM, который *управляет* микросхемой. Но откуда гарантия, что время **рабочего цикла** микросхемы и момент начала тактового импульса контроллера совпадут? То есть, вообще говоря, рабочий цикл никогда не совпадает с началом тактового импульса. Для примера можно взять хоть тот факт, доказывающий это, что требуется несколько наносекунд на стабилизацию сигналов. и уже потому начало подачи сигнала контроллером будут не совпадать. Чтобы синхронизироваться, им нужно не менее 2 тактов (обычно 4).

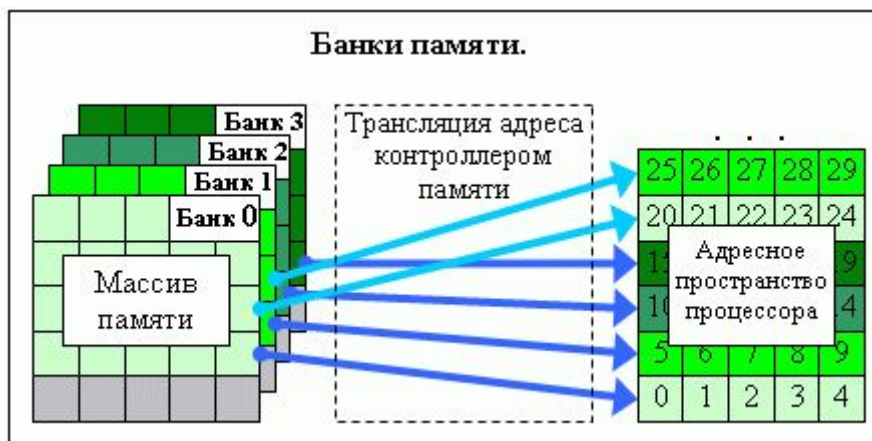
### **SDRAM (синхронная DRAM).**

Из названия следует, что контроллер памяти и микросхема работают синхронно, то же что и согласованно. То есть строки и столбцы подаются теперь таким образом, чтобы к приходу следующего тактового импульса от контроллера сигналы уже успели стабилизироваться и были готовы к считыванию. Если говорить еще об общих чертах устройства, то пакетный режим (помним же, что это то, сколько столбцов будет считано из открытой строки) стал более усовершенствованным. Теперь контроллер

может запросить сколько угодно последовательных ячеек памяти ( степень двойки), вплоть до целой строки целиком. Все это благодаря полноразрядному адресному счетчику.

Другая модификация заключается в том, что появилось новое обозначение, хотя это просто модификация старого определения. Мы помним, что существует такое определение, как матрица памяти. Чтобы обеспечить возможность работы с разными участками памяти одновременно, используется архитектура с несколькими такими матрицами, каждая из которых теперь называется **банком**. Количество банков изначально увеличилось до 2, позже 4 (в спецификации это четко отражено). Это позволяет реально распараллелить процесс получения нужной ячейки или ячеек: можно обратиться к ячейкам одного банка, в то время как внутренние цепи другого банка будут перезаряжаться (ну и понятно, что раз банков много, значит есть несколько буферов, где эти матрицы лежат).

Непонятно, что именно лежит в банках. Там могут лежать просто одни и те же копии какой-то матрицы (так было на ранних этапах), однако это не очень эффективно и в последствии технологию изменили.



Распространенный способ устройства банков таков: сначала идет строка первого банка, далее второго, после третьего и так до последнего. Позже, если это влезит в пакет, цикл начинается заново. В этом случае мы получаем оптимизацию при работе с хаотичными запросами, если же они упорядочены, то это существенный минус.

Еще одно нововведение, так это возможность одновременного открытия нескольких страниц памяти (то есть передача номера строки, но Пьянзин оперировал обоими терминами, так что надо знать оба). Они могут открываться в момент, когда информация только лишь считывается с другой. В силу этого на каждом новом тактовом цикле можно обращаться к новому столбцу строки.

Если все виды памяти, что рассмотрены выше выполняют перезарядку внутренних цепей при закрытии страницы, то SDRAM делает это автоматически (на самом деле за счет того, что контроллер то умеет синхронизироваться с микросхемой памяти), что позволяет держать страницы открытыми сколько угодно долго (еще раз, страница == строка).

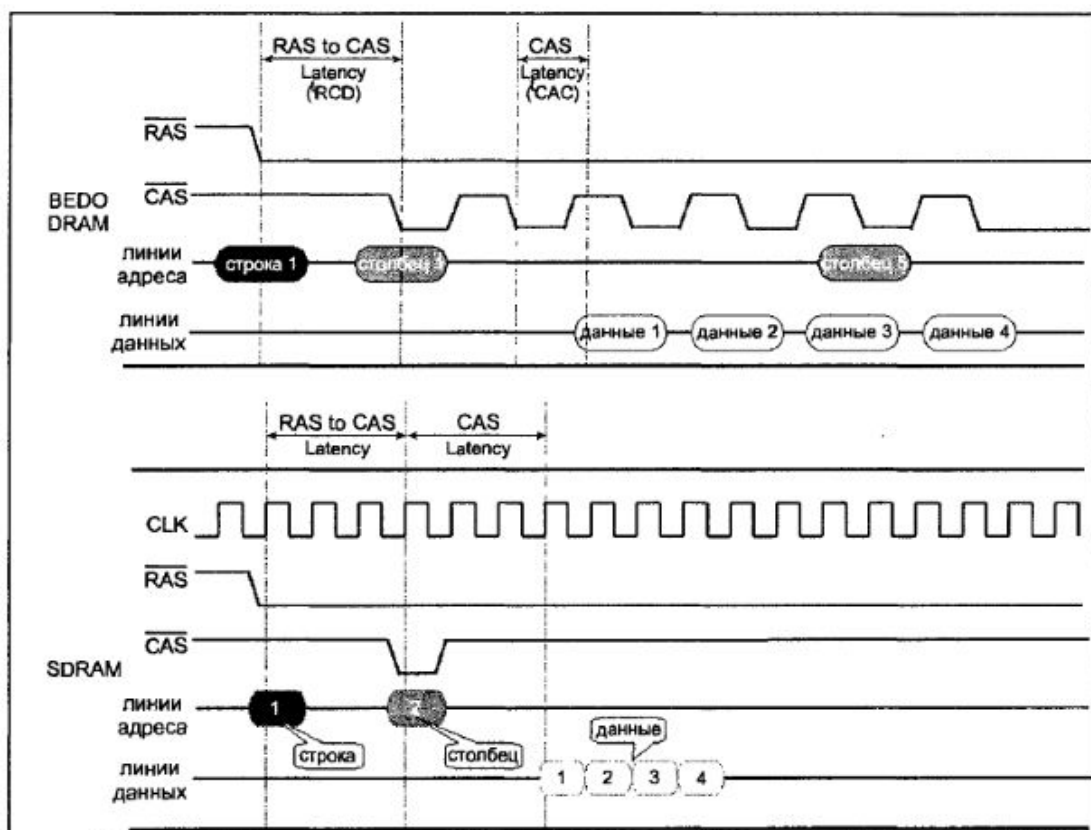
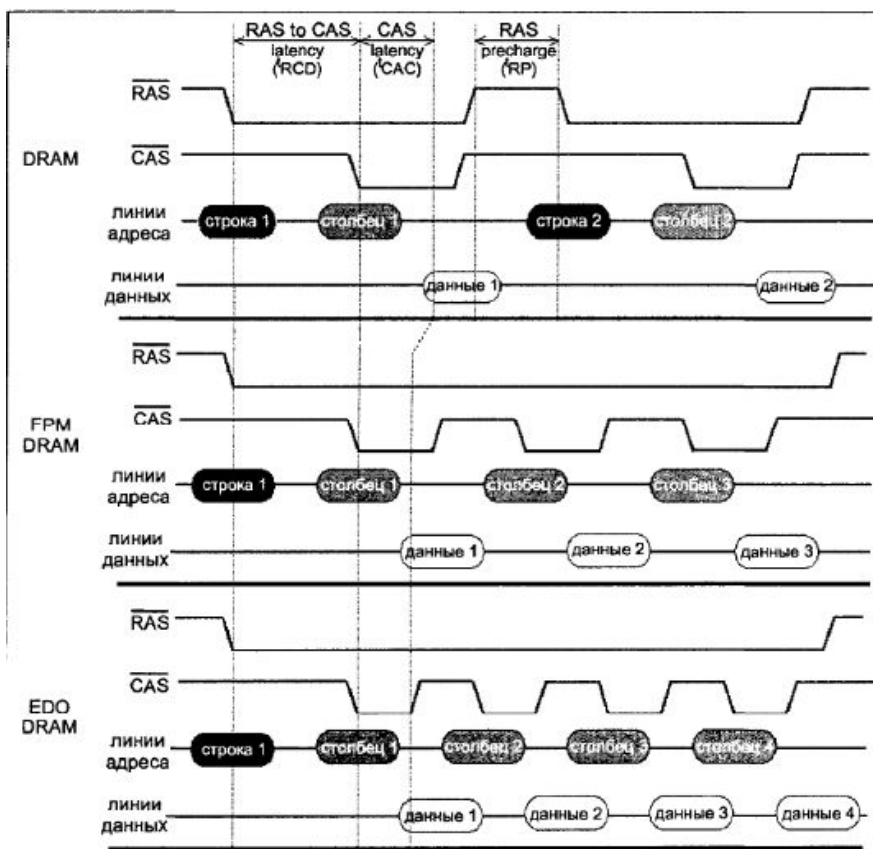
Чтобы понять принцип посмотрите на самое последнее изображение. Появилась новая величина, **CLK** - clock latency (это, между прочим, очень тонкий момент, так как именно CLK - это и есть эффективная частота!), по простому, это моменты, когда контроллер синхронизируется с микросхемой. Видите, как часто? С какого-то момента открывается строка некоторого банка, которая считывается (параллельно работает еще и другой банк, вероятно), считывается нужный столбец текущей страницы, где-то параллельно взялся номер такой же номер строки другого банка и считался его столбец, получив еще порцию данных. Но не принимайте слова в буквальном смысле, в пределах одного такта мы все равно можем считать лишь одну ячейку (ведь нужно время на перезарядку и обновление). Постоянно наблюдается синхронизация с контроллером, и если контроллер решит, что длина пакета удовлетворяет каким-то условиям, он заберет данные и, возможно, закроет страницу, но возможно и нет (на рисунке она остается открытой.)

В завершении, разрядность линии данных тоже выросла. изменилась с 32 бит до 64.

## DDR, DDR2, DDR3

Дальнейшее усовершенствование SDRAM, это появление DDR (Double Data Rate SDRAM) - то есть SDRAM с удвоенной скоростью передачи данных. На лекции это не разбиралось, и понятие фронт-сигнала и спад-сигнала также не вводились, и, кажется, Пьянзин разрешил этого и не делать, а просто знать, что ***фронт*** - переход сигнала из состояния 0 в состояние 1, а ***спад*** - наоборот. Удвоение скорости достигается именно за счет этого эффекта - передача данных осуществляется в обоих направлениях. Благодаря этому CLK (эффективная частота) для DDR стала 100 МГц да при этом еще и энергопотребление сократилось. Эта величина аналогична 200 МГц для обычной SDRAM. Последующие версии, то есть DDR2 и DDR3 являли собой только дополнение идеи DDR. Основное отличие DDR2 от DDR - вдвое большая частота работы шины, по которой данные передаются в буфер микросхемы, для DDR3 - соответственно в 3. Это означает, что происходит увеличение пропускной способности шины. Однако помните, что эффективная частота во всех модификациях остается той же самой - **100 МГц**.

В конце можно отметить, что уже в DDR была существенно пересмотрена архитектура работы с банками. Кроме того, что количество банков выросло до 4 (было 2), каждый банк обзавелся собственным контроллером (не путать с контроллером памяти!). Визуально это стало выглядеть, будто бы вместо одной микросхемы получилось 4, которые работают независимо друг от друга. Соответственно, максимальное количество ячеек, обрабатываемых за один такт возросло до 4.



## **Дополнительная информация.**

1. [Структура микропроцессоров IA-32, Режимы работы микропроцессоров IA-32](#)

## 2. [Архитектура ЦП 8086](#)

Семейство процессоров Intel с ключевыми данными:

Название	Год Выпус- ка	Макс. так- товая час- тота, ГГц	Трази- сто-ров ЦП, млн.	Размер регистров, бит	Ширина шины данных, бит	Адресное простран- ство	Проектная ширина, мкм
i8086	1978	0.008	0.029	16	16	1 Мб	3
i80286	1982	0.025	0.134	16	16	16 Мб	1.5
i80386	1985-92	0.016-0.033	0.275	32	32	4 Гб	1.5-1.0
i80486	1989-94	0.025-0.100	1.2	32	32	4 Гб	1.0-0.6
P5 (Pentium)	1993-96	0.060-0.233	3.1	32	64	4 Гб	0.8-0.35
P6 (Pentium Pro)	1995-97	0.150-0.200	5.5	32	64	64 Гб	0.6-0.35
Pentium II	1997-98	0.233-0.450	7.5	32	64	64 Гб	0.25-0.18
Celeron	1998-02	0.266-2.2	18.9	32	64	64	0.25-0.13
Pentium III	1999-02	0.450-1.200	28	32	64	64 Гб	0.18-0.13
Pentium 4	2000-02	1.400-3.000	55	32	2x64	64 Гб	0.18-0.13
Itanium 1	2001	0.733-0.800	220	64	64	16 Тб	0.18
Itanium 2	2002-07	0.900-1.000	220-1000	64	128	16 Тб	0.18
Itanium 9300	2010	1.4 – 1.8	2000	64	128	64 Тб	
Itanium 9500	2012	1.7–2.5	3100	64	128	512 Тб	

3. [Ясное и подробное введение в ассемблер\(а также связанные с ним определения\)](#)

4) [Разжёвано про CISC and RISC](#)