Оглавление

[Глава 1. Вычислители 3](#_Toc508616638)

[1.1. Электронная вычислительная машина. Классификация ЭВМ 3](#_Toc508616639)

[1.2. Архитектура вычислителя и его расширяющие компоненты 8](#_Toc508616640)

[1.3. Специализированные вычислители 12](#_Toc508616641)

[1.4. Понятие об иерархических и параллельных вычислителях 14](#_Toc508616642)

[1.5. Разрядность элементов вычислителя 18](#_Toc508616643)

[1.6. Программная среда специализированного вычислителя 26](#_Toc508616644)

[1.7. Использование компьютера общего назначения для разработки программного обеспечения для специализированных вычислителей. Кросс-средства 29](#_Toc508616645)

[Глава 2. Процессор (компьютер) 34](#_Toc508616646)

[2.1. Основные узлы процессоров 34](#_Toc508616647)

[2.2. Ядро процессора 43](#_Toc508616648)

[2.3. Представление о микросхемах обвязки процессоров 51](#_Toc508616649)

[2.4. Представление о конвейерном выполнении команд в процессорах 54](#_Toc508616650)

[2.5. Наборы команд и взаимодействие с операндами с позиций набора команд 60](#_Toc508616651)

[2.6. Классификация и особенности шин в отношении способа передачи, метода синхронизации, топологии, способа управления, адресации устройств 63](#_Toc508616652)

[Глава 3. Процессор (программирование) 74](#_Toc508616653)

[3.1. Понятие о параллельных и распределённых вычислениях 74](#_Toc508616654)

[3.2. Программно-аппаратные переходы и аспекты их применения. Жесткие процессоры и процессоры, использующие микрокод 79](#_Toc508616655)

[3.3. Специализированные средства разработки программного обеспечения 80](#_Toc508616656)

[3.4. Модульное, функциональное и структурное программирование применительно к специализированным вычислителям 86](#_Toc508616657)

[3.5. Директивы и выражения Ассемблера: основные классы команд по назначению 92](#_Toc508616658)

[3.6. Директивы и выражения Ассемблера для работы с памятью (куча и стек). 100](#_Toc508616659)

[3.7. Директивы и выражения Ассемблера для осуществления арифметических, побитовых и логических операций. 113](#_Toc508616660)

[3.8. Директивы и выражения Ассемблера для управления вычислениями (ветвление, циклы, переходы, прерывания). 126](#_Toc508616661)

[3.9. Алгоритм ассемблирования, внутренние таблицы, выходные файлы. 136](#_Toc508616662)

[3.10. Загрузчики и компоновщики. Форматы объектных модулей. Оверлейные загрузчики и оверлеи. 140](#_Toc508616663)

[Предметный указатель 144](#_Toc508616664)

# Глава 1. Вычислители

# 1.1. Электронная вычислительная машина. Классификация ЭВМ

*Определение 1.1.1*

**Электронная вычислительная машина (ЭВМ) или компьютер** — это устройство, выполняющее операции ввода данных, их сохранение и обработку по определенной программе, вывод полученных результатов в форме, пригодной для восприятия человеком. В дальнейшем будем считать ЭВМ=компьютер.

Компьютеры классифицируется по следующим критериям.

**Классификация компьютеров в отношении круга решаемых задач**:

(Критерием деления вычислительных машин на эти три класса является форма представления информации, с которой они работают.)

* Цифровые вычислительные машины (ЦВМ) – вычислительные машины дискретного действия, работают с информацией, представленной в дискретной, а точнее, в цифровой форме.
* Аналоговые вычислительные машины (АВМ) – вычислительные машины непрерывного действия, работают с информацией, представленной в непрерывной (аналоговой) форме, т.е. в виде непрерывного ряда значений какой-либо физической величины (чаще всего электрического напряжения). АВМ машины весьма просты и удобны в эксплуатации; программирование задач для решения на них, как правило, не трудоемкое; скорость решения задач изменяется по желанию оператора и может быть сделана сколь угодно большой (больше, чем у ЦВМ), но точность решения задач очень низкая (относительная погрешность 2 –5%). На АВМ наиболее эффективно решать математические задачи, содержащие дифференциальные уравнения, не требующие сложной логики. В современном мире почти не используются.
* Гибридные вычислительные машины (ГВМ) – вычислительные машины комбинированного действия, работают с информацией, представленной и в цифровой, и в аналоговой форме; они совмещают в себе достоинства АВМ и ЦВМ. ГВМ целесообразно использовать для решения задач управления сложными быстродействующими техническими комплексами.

**Классификация компьютеров в отношении условий эксплуатации:**

* офисные(универсальные) - предназначены для решения широкого класса задач при нормальных условиях эксплуатации.
* специальные(можно подразделить на мобильные, военные и т.д.) - для решения более узкого класса задач или даже одной задачи, требующей многократного решения, и функционируют в особых условиях эксплуатации. Также специальные компьютеры управляют технологическими установками, работают в операционных или машинах скорой помощи, на ракетах, самолетах и вертолетах, вблизи высоковольтных линий передач или в зоне действия радаров, радиопередатчиков, в неотапливаемых помещениях, под водой на глубине, в условиях пыли, грязи, вибраций, взрывоопасных газов и т. п.

Машинные ресурсы специальных компьютеров часто ограничены. Однако их узкая ориентация позволяет реализовать заданный класс задач наиболее эффективно.

**Классификация компьютеров в отношение архитектуры:**

*Определение 1.1.2*

**Архитектура компьютера** — логическая организация и структура аппаратных и программных ресурсов вычислительной системы. Архитектура заключает в себе требования к функциональности и принципы организации основных узлов компьютера.

*Определение 1.1.3*

**Арифметико-логическое устройство** (АЛУ) — блок процессора, который под управлением устройства управления (УУ) служит для выполнения арифметических и логических преобразований (начиная от элементарных) над данными, называемыми в этом случае операндами.

В настоящее время наибольшее распространение в компьютерах получили два типа архитектуры:

* принстонская (фон Неймана)
* [гарвардская](http://ru.wikipedia.org/wiki/%D0%93%D0%B0%D1%80%D0%B2%D0%B0%D1%80%D0%B4%D1%81%D0%BA%D0%B0%D1%8F_%D0%B0%D1%80%D1%85%D0%B8%D1%82%D0%B5%D0%BA%D1%82%D1%83%D1%80%D0%B0).

Обе выделяют два основных узла компьютера: центральный процессор и память компьютера.

Принстонская архитектура (фон Неймана) (Рис.1.1.1а) характеризуется использованием общей оперативной памяти для хранения программ, данных, а также для организации стека. Для обращения к этой памяти используется общая системная шина, по которой в процессор поступают и команды, и данные.

Классическая гарвардская архитектура (Рис.1.1.1б) использует разделенную кэш-память (команды хранятся в одной кэш-памяти, а данные — в другой). Разделенная кэш-память позволяет осуществлять параллельный доступ. К тому же, поскольку команды обычно не меняются во время выполнения программы, содержание кэша команд не приходится записывать обратно в основную память.

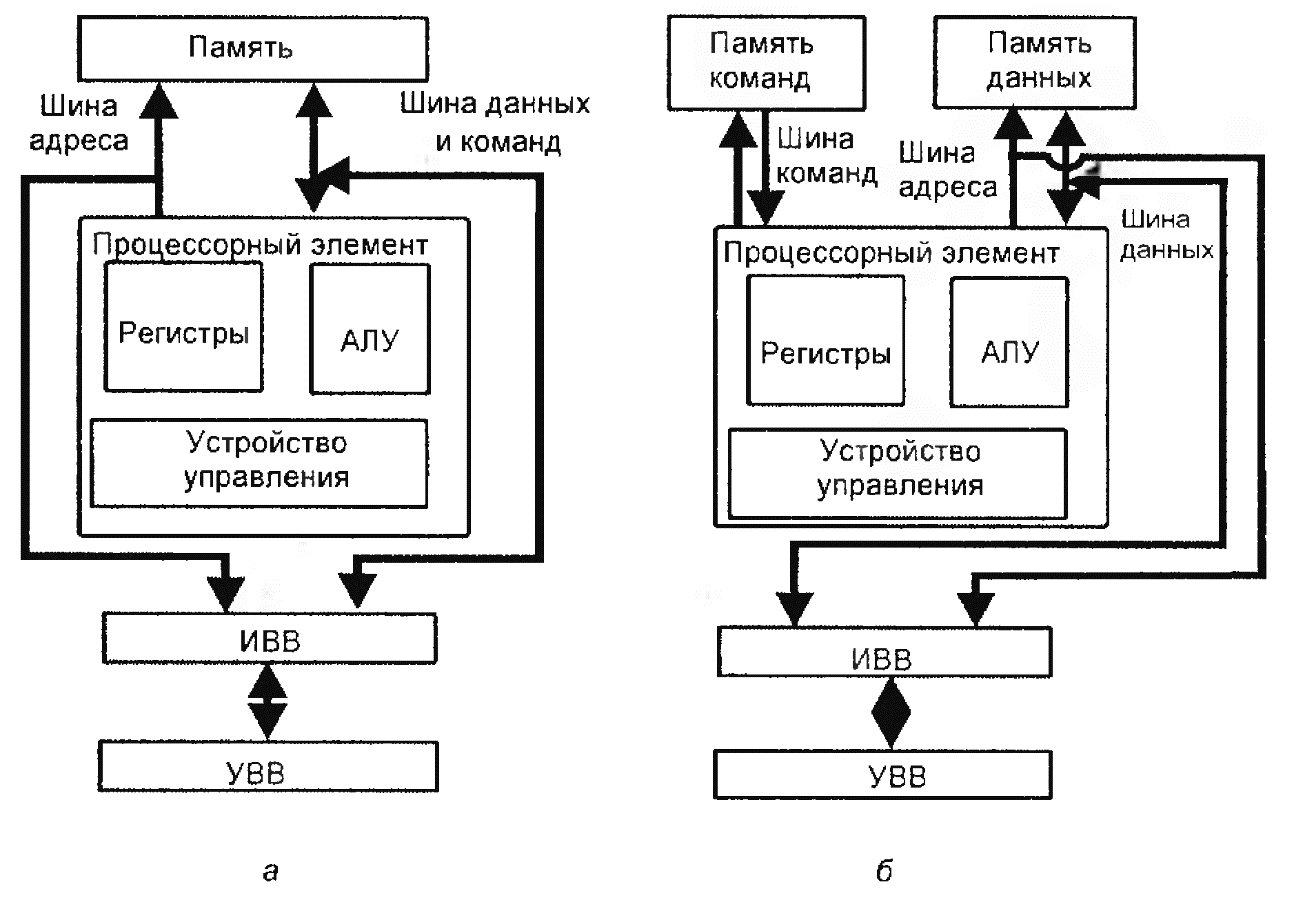


Рис.1.1.1

(**ИВВ** - интерфейсы ввода/вывода, **УВВ** - устройства ввода/вывода)

То есть различие заключается в структуре памяти: в принстонской (фон Неймана) архитектуре программы и данные хранятся в одном массиве памяти и передаются в процессор по одной шине, тогда как гарвардская архитектура предусматривает отдельные хранилища и потоки передачи для команд и данных.

**Классификации компьютеров в отношении производительности и области применения:**

По производительности и характеру использования компьютеры можно условно подразделить на:

* **микрокомпьютеры** — это компьютеры, в которых центральный процессор выполнен в виде микропроцессора.

Продвинутые модели микрокомпьютеров имеют несколько микропроцессоров. Производительность компьютера определяется не только характеристиками применяемого микропроцессора, но и ёмкостью оперативной памяти, типами периферийных устройств, качеством конструктивных решений и др.

*Определение 1.1.4*

**Микроконтроллеры** – микрокомпьютеры выполнение в виде одной микросхемы. Используются для автоматизации работы не сложных электронных устройств.

*Определение 1.1.5*

**Персональные компьютеры** – однопользовательские микрокомпьютеры, удовлетворяющие требованиям общедоступности и универсальности.

*Определение 1.1.6*

**Рабочие станции** – однопользовательские мощные микрокомпьютеры, специализированные на выполнении определенного вида работы (инженерные, графические, издательские и т.д.).

*Определение 1.1.7*

**Серверы** – многопользовательские мощные микрокомпьютеры в вычислительных сетях выделенные для обработки запросов от всех станций сети.

Можно выделить следующие группы в классе микрокомпьютеры:

* мэйнфреймы (универсальные компьютеры). Основные направления применения мэйнфреймов – это решение научно-технических задач, работа в вычислительных системах с пакетной обработкой информации, работа с большими базами данных, управление вычислительными сетями и ресурсами. В мэйнфреймах плохо соблюдается принцип открытых систем – а именно совместимость с другими системами.
* миникомпьютеры - надежные, недорогие и удобные в эксплуатации компьютеры, обладающие несколько низкими параметрами по сравнению с мэйнфреймами. Основными особенностями являются: широкий диапазон производительности в конкретных условиях применения, аппаратная реализация большинства системных функций ввода-вывода информации, простая реализация микропроцессорных и многомашинных систем, высокая скорость обработки прерываний, возможность работы с форматами данных различной длины. К достоинствам можно отнести: специфическую архитектуру с большой модульностью, лучшее, чем у мэйнфреймов, соотношение производительности и цены, повышенная точность вычисления. Миникомпьютеры ориентированы на использование в качестве управляющих вычислительных комплексов, для вычислений в многопользовательских вычислительных системах, в системах автоматического проектирования, для моделирования несложных объектов.
* суперкомпьютеры - компьютеры с огромной вычислительной мощностью, предназначающиеся для высокопроизводительных вычислений. В настоящее время суперкомпьютеры используются для работы с приложениями, требующими наиболее интенсивных вычислений (например, прогнозирование погодноклиматических условий, моделирование ядерных испытаний и т. п.). Архитектура суперкомпьютеров основана на идеях параллелизма (параллельная обработка данных) и конвейеризации вычислений (увеличения числа инструкций, выполняемых в единицу времени — эксплуатация [параллелизма на уровне инструкций](https://ru.wikipedia.org/wiki/%D0%9F%D0%B0%D1%80%D0%B0%D0%BB%D0%BB%D0%B5%D0%BB%D0%B8%D0%B7%D0%BC_%D0%BD%D0%B0_%D1%83%D1%80%D0%BE%D0%B2%D0%BD%D0%B5_%D0%BA%D0%BE%D0%BC%D0%B0%D0%BD%D0%B4)).

**Критерии выбора решений:**

(Полностью зависит от поставленной задачи)

Основные параметры для выбора решений:

* производительность
* стоимость
* надежность
* масса
* габариты.

# 1.2. Архитектура вычислителя и его расширяющие компоненты

*Определение 1.2.1*

Под **архитектурой вычислительного средства** – понимается совокупность общих принципов организации аппаратно-программных средств и их характеристик, определяющая функциональные возможности компьютера при решении соответствующих классов задач.

**Обязательные компоненты вычислителя:**

* процессоры,
* энергонезависимая память,
* системы управления и ввода-вывода информации

*Определение 1.2.2*

**Процессор** – функциональная часть компьютера, выполняющая основные операции по обработке данных и управлению работой других блоков.

Процессор является преобразователем информации, поступающей из памяти и внешних устройств. Наиболее важными частями процессора являются арифметико-логическое устройство (АЛУ) и устройство управления (УУ).

*Определение 1.2.3*

**Сопроцессор** — это специальное устройство для выполнения математических операций над вещественными (дробными) числами.

*Определение 1.2.4*

**Оперативная память** (оперативное запоминающее устройство (ОЗУ), RAM – Random Access Memory) - [энергозависимая](https://ru.wikipedia.org/wiki/%D0%AD%D0%BD%D0%B5%D1%80%D0%B3%D0%BE%D0%BD%D0%B5%D0%B7%D0%B0%D0%B2%D0%B8%D1%81%D0%B8%D0%BC%D0%B0%D1%8F_%D0%BF%D0%B0%D0%BC%D1%8F%D1%82%D1%8C) часть системы [компьютерной памяти](https://ru.wikipedia.org/wiki/%D0%9A%D0%BE%D0%BC%D0%BF%D1%8C%D1%8E%D1%82%D0%B5%D1%80%D0%BD%D0%B0%D1%8F_%D0%BF%D0%B0%D0%BC%D1%8F%D1%82%D1%8C), в которой во время работы компьютера хранится выполняемый машинный код, а также входные, выходные и промежуточные данные, обрабатываемые [процессором](https://ru.wikipedia.org/wiki/%D0%9F%D1%80%D0%BE%D1%86%D0%B5%D1%81%D1%81%D0%BE%D1%80).

Содержимое этого вида памяти не сохраняется при выключении ПК.

*Определение 1.2.5*

**Кэш-память** (cache – тайник) — это высокоскоростная память произвольного доступа, используемая процессором компьютера для временного хранения информации.

Время доступа к информации, хранящейся в кэш-памяти, меньше, чем время доступа к этой же информации, хранящейся в других видах памяти ПК.

*Определение 1.2.6*

**Энергонезависимая память** — это компьютерная память, которая может хранить информацию при отсутствии питания (флеш-память, жесткий диск и т.д.).

*Определение 1.2.7*

**Порты** – это устройства для подключения к системной шине различных внешних устройств.

Порты ввода-вывода соединяют два устройства. Однако часто целесообразно подключить к одному порту передачи данных несколько устройств, причем необязательно однотипных.

*Определение 1.2.8*

Каждая передаваемая по такому порту порция данных обязана сопровождаться адресом, который указывает, какому из подключенных устройств она предназначена. Такие многоточечные порты называются - **шинами**.

Компьютерная шина служит для передачи данных между отдельными функциональными блоками системы.

Шина состоит из трех частей:

1. шина адреса, на которой устанавливается адрес требуемой ячейки памяти или устройства, с которым будет происходить обмен информацией;
2. шина данных, по которой собственно и будет передана необходимая информация;
3. шина управления, регулирующая этот процесс (например, один из сигналов на этой шине позволяет компьютеру различать между собой адреса памяти и устройств ввода/вывода).

*Определение 1.2.9*

**Периферийные устройства** - аппаратура, которая позволяет вводить [информацию](https://ru.wikipedia.org/wiki/%D0%98%D0%BD%D1%84%D0%BE%D1%80%D0%BC%D0%B0%D1%86%D0%B8%D1%8F) в [компьютер](https://ru.wikipedia.org/wiki/%D0%9A%D0%BE%D0%BC%D0%BF%D1%8C%D1%8E%D1%82%D0%B5%D1%80) или выводить её из него[.](https://ru.wikipedia.org/wiki/%D0%9F%D0%B5%D1%80%D0%B8%D1%84%D0%B5%D1%80%D0%B8%D0%B9%D0%BD%D0%BE%D0%B5_%D1%83%D1%81%D1%82%D1%80%D0%BE%D0%B9%D1%81%D1%82%D0%B2%D0%BE#cite_note-1)

*Определение 1.2.10*

Периферийные устройства присоединяются к ПК через так называемые **устройства сопряжения** или **адаптеры**.

Взаимодействие периферийных устройств с адаптером происходит через порты ввода/вывода.

По способу передачи информации порты ввода-вывода делятся на:

Последовательные – информация передается последовательно бит за битом; для передачи информации используется один провод. Подключаются внешние устройства, находящиеся на большом расстоянии от ПК.

Параллельные – несколько битов информации передается одновременно; для передачи информации используется несколько проводов. Подключаются устройства, находящиеся рядом с ПК.

**Системы электропитания и энергосбережения**

*Определение 1.2.11*

**Гибернация** — энергосберегающий режим операционной системы компьютера, позволяющий сохранять содержимое оперативной памяти на энергонезависимое устройство хранения данных (жесткий диск) перед выключением питания.

В обычном режиме все компоненты компьютера потребляют столько энергии, сколько им нужно. В режиме сна выключаются периферийные устройства, а на процессор, оперативную память, беспроводные модули и прочие компоненты в таком режиме подается минимальное напряжение. В режиме гибернации компьютер обесточивается полностью (остается только питание на кварцевом генераторе для поддержания даты/времени). При этом на жесткий диск записывается точная копия оперативной памяти ПК в момент выключения и при включении ПК этот образ восстанавливается с жесткого диска обратно в оперативную память. Иными словами, компьютер запоминает состояние системы перед уходом в режим гибернации, а затем восстанавливает все вкладки и настройки какие были. Разница между режимами сна и гибернации в скорости возобновления работы с компьютером с прежнего места, из режима сна это происходит значительно быстрее, чем из режима гибернации (а из гибернации быстрее, чем обычная загрузка Windows). Однако режим гибернации эффективнее экономит энергию в случае долгого бездействие компьютера (чем дольше компьютер находится в режиме гибернации, тем эффективнее экономится энергия, в сравнении с режимом сна).

Основная идея этой системы (электропитания и энергосбережения — это одна, неразделимая система) состоит в том, что у каждого элемента компьютера есть компонент, который следит за загруженностью этого элемента. Если в течении заданного времени элемент бездействует, то включается режим энергосбережения, а что именно этот режим делает с элементом компьютера зависит от конкретного элемента. Например, в случае процессора понижается тактовая частота или питание от процессора отключается вовсе, при этом данные из регистров процессора записываются в энергонезависимую память (чтобы их можно было восстановить). В случае энергозависимой памяти, система энергосбережения сначала записывает все данные из неё в энергонезависимую память, а затем отключает питание. В случае с жестким диском обычно уменьшается скорость вращения самого диска, полная его остановка не желательна, так как требуется значительное (по меркам компьютера) время, для его запуска (раскрутки). В современных компьютерах (в частности под управлением ОС Windows) присутствуют разные режимы энергопотребления. Обычный (нормальный) режим, режим сна и режим гибернации.

# 1.3. Специализированные вычислители

*Определение 1.3.1*

**Специализированные вычислители —** компьютеры, заточенные по архитектуре и набору команд под узкий класс задач или под конкретную задачу.

При проектировании, для соответствия устройства поставленным требованиям (техническим, экономическим - общая стоимость, складывающаяся из стоимости всех составляющих компьютера), приходится жертвовать характеристиками, такими как: габаритность компьютера, производительность, функциональность и надежность ради выгоды в других параметрах, таких как: надежность компьютера, стоимость устройства, энергопотребление и тепловыделение.

Пример: GPU в видеокарте или специальные компьютеры повышенной надежности для вооруженных сил.

Одним из наиболее простых способов классифицировать различные типы вычислительных устройств является определение их способностей. Все вычислители могут, таким образом, быть отнесены к одному из трёх типов:

1. специализированные устройства, умеющие выполнять только одну функцию;
2. устройства специального назначения, которые могут выполнять ограниченный диапазон функций;
3. устройства общего назначения, используемые сегодня. Название «Компьютер» применяется, как правило, именно к машинам общего назначения.

*Определение 1.3.2*

**Процессор общего назначения** - такие процессоры могут достаточно эффективно решать широкий класс задач управления, вычислительных и прочих. Именно процессоры этого класса используются в качестве центрального процессора в настольных рабочих станциях.

*Определение 1.3.3*

**Специализированный процессор** — это процессор, у которого особенности архитектуры, набора структурных блоков, системы команд или конструктивно - технологического исполнения позволяют значительно повысить эффективность решения достаточно узкого круга специальных задач по сравнению с иными применениями.

*Определение 1.3.4*

**Цифровой сигнальный процессор** - это специализированный микропроцессор, архитектура которого оптимизирована для операций, необходимых для цифровой обработки сигналов в режиме реального времени. ЦСП измеряет, фильтрует и/или сжимает постоянный аналоговый сигнал.

Математически эти задачи сводятся к поэлементному перемножению элементов многомерных векторов действительных чисел, последующему суммированию произведений. Сигнальные процессоры оптимизированы по быстродействию для выполнения именно таких операций. ЦСП используют специальную архитектуру памяти, позволяющую получать данные и инструкции одновременно (Гарвардская архитектура)*.*

# 1.4. Понятие об иерархических и параллельных вычислителях

*Определение 1.4.1*

**Уровни образуются группами функций ОС** - файловая система, управление процессами и устройствами и т.п. Каждый уровень может взаимодействовать только со своим непосредственным соседом - выше или нижележащим уровнем. Прикладные программы или модули самой операционной системы передают запросы вверх или вниз по этим уровням.

В организации вычислительной системы можно выделить 9 уровней иерархии:

1. Языки высокого уровня
2. Языки ассемблера
3. Уровень ОС
4. Уровень машинных команд
5. Микроархитектурный уровень
6. Уровень системотехники
7. Уровень цифровой схемотехники
8. Уровень аналоговой схемотехники
9. Физический уровень

Иерархические вычислители бывают разные.

Примеры:

1. Дерево, которое сперва разветвляется, а потом, когда на нижнем уровне возник ответ, собирает всю информацию обратно и отправляет к вам через обратный путь.
2. Разделяй и властвуй — это когда сложность решения двух половинок сложной задачи и объединение этих решений ниже, чем решение одной большой сложной задачи.

*Определение 1.4.2*

Современные разработчики пошли по этому пути, поскольку сейчас широко распространены **конвейерные архитектуры**, а при конвейерной организации обращения и к командам, и к данным (операндам) должны осуществляться одновременно.

*Определение 1.4.3*

Современные процессоры имеют **гибридную архитектуру**, например, обладают раздельной кэш-памятью для инструкций и данных, что позволяет им за один рабочий такт получать одновременно и команду, и данные для ее выполнения. То есть процессорное ядро, формально, является гарвардским, но программно оно фон-неймановское, что упрощает написание программ. Обычно в данных процессорах одна шина используется и для передачи команд, и для передачи данных, что упрощает конструкцию системы.

**Понятие избыточности и её использование для создания отказоустойчивых систем**

Существует два типа избыточности: пространственная и временная.

*Определение 1.4.4*

**Избыточность пространства** реализуется путем введения дополнительных компонентов, функций или данных, которые не нужны при безотказном функционировании. Дополнительные (избыточные) компоненты могут быть аппаратными, программными и информационными.

*Определение 1.4.5*

**Временная избыточность** реализуется путем повторных вычислений или отправки данных, после чего результат сравнивается с сохранённой копией предыдущего.

Избыточность является основным способом повышения отказоустойчивости.

Избыточность на уровне данных очень полезна для защиты целостности данных и для обеспечения средств исправления. Информационная избыточность позволяет составлять обоснованные тесты обнаружения ошибок для систем, использующих независимые данные. Она также позволяет восстановить данные, испорченные из-за программных ошибок или аппаратных сбоев.

Наиболее эффективный метод избыточности — аппаратная избыточность, которая достигается путем резервирования.

*Определение 1.4.6*

**Резервирование** — метод повышения характеристик надёжности технических устройств или поддержания их на требуемом уровне посредством введения аппаратной избыточности за счет включения запасных (резервных) элементов и связей, дополнительных по сравнению с минимально необходимым для выполнения заданных функций в данных условиях работы. (Например, дублирование блока питания, резервирование жёстких дисков для построения «зеркального» RAID 1.)

*Определение 1.4.7*

**Отказоустойчивость** — свойство технической системы сохранять свою работоспособность после отказа одного или нескольких составных компонентов. Отказоустойчивость определяется количеством любых последовательных единичных отказов компонентов, после которого сохраняется работоспособность системы в целом. Базовый уровень отказоустойчивости подразумевает защиту от отказа одного любого элемента — исключение единой точки отказа.

**Вычислители с множественными потоками команд и одиночным потоком данных (MISD), истинно параллельные процессоры (MIMD), систолические вычислители**

*Определение 1.4.8*

**MISD** — множественными потоками команд и одиночным потоком данных. Здесь несколько команд оперируют одним набором данных. Специфическая архитектура, она существует (вопреки скептикам), но это крайне редко бывает нужно.

*Определение 1.4.9*

**MIMD** — множественный поток команд, множественный поток данных — содержит много процессоров, которые (как правило, асинхронно) выполняют разные команды над разными данными. Т.е. несколько независимых процессоров работают как часть большой системы. Таких машин крайне мало. В эту категорию попадают большинство параллельных процессоров. К MIMD-машинам относятся и мультипроцессоры (машины с общей памятью), и мультикомпьютеры (машины с обменом сообщениями).

**Гибкие архитектуры и дилемма «отказоустойчивость-эффективность»**

Рассмотрим несколько примеров гибких архитектур.

Например, машины сгруппированы в так называемый блейд-сервер, т.е. в самостоятельный блок, на котором стоит процессор, память и все, что нужно для общения с шиной. Никаких внешних контроллеров, кроме контроллера шины на нем нет. Велика вероятность, что за час что-то выйдет из строя. Но пользователь, по возможности, этого заметить не должен. Есть встроенные аппаратные средства диагностики, которые автоматически отключают сгоревшую плату от шины и вообще снимают с нее питание, чтобы техник мог ее заменить/починить. Тем временем тут же от аппаратных средств идет в ОС-у сигнал о выходе из строя одного из вычислителей. ОС анализирует какие задания этот вычислитель получил, но не успел посчитать. Эти задания до их исполнения хранятся в этой иерархии серверов, которые распределяют задачи. Далее планировщик раскидывает эти задачи, чтобы примерно выровнять нагрузку на все компьютеры. Таким образом пользователь особо ничего не заподозрит, даже при том, что задержка будет более чем двойная.

Также примером гибких архитектур является архитектура систем сотовых связей. Система сотовой связи — это сложная и гибкая техническая система, допускающая большое разнообразие, как по вариантам конфигураций, так и по набору выполняемых функций. Примером сложности и гибкости системы является то, что она может обеспечивать передачу, как речи, так и других видов информации, в частности текстовых сообщений и компьютерных данных. В части передачи речи, в свою очередь, может быть реализована обычная двусторонняя телефонная связь, многосторонняя телефонная связь (так называемая конференцсвязь - с участием в разговоре более двух абонентов одновременно), голосовая почта. При организации обычного двустороннего телефонного разговора, начинающегося с вызова, возможны режимы автодозвона, ожидания вызова, переадресации вызова.

Также рассмотрим **дилемму «отказоустойчивость-эффективность»**.

Там, где ответственность за ошибку ниже, там используется динамическое балансирование: когда имеется возможность при выходе какой-то взаимозаменяемой части ресурса заменить другими, естественно с общим уменьшением производительности. Т.е. есть статическое в этом смысле распараллеливание, есть динамическая балансировка нагрузки.

Там, где система жесткая, т.е. вероятность ошибки должна стремиться к нулю, используется статическое распараллеливание. А также при таком решении задачи центральный компьютер может посчитывать вероятность того, что найденное решение правильное.

# 1.5. Разрядность элементов вычислителя

*Определение 1.5.1*

**Разрядность процессора** — это число битов, обрабатываемых процессором одновременно. Процессор может быть 8-, 16-, 32- и 64-, 128-, 256-, 512-разрядным (разрядность начиная с 128- по 512— это возможные разрядности шины памяти видеокарт). Вместе с быстродействием разрядность характеризует объем информации, перерабатываемый процессором компьютера за единицу времени. Также на разрядность вычислителя влияют разрядности шин. На данный момент пытаются делать так, чтобы шина была быстрее, чем процессор.

Чем больше разрядность, тем медленнее, но точнее будет процессор считать. Эффективность зависит от решаемой задачи. Из-за того, что в комплектующих компьютера зачастую различная разрядность, эффективность падает, но все зависит от конкретных задач.

**Специализированные алгоритмы и вычисления с повышенной точностью.**

Существует задачи: вычисления криптографических ключей, задачи астрономии и др., которые требуют очень большой точности. Для целых чисел точность в млрд. Для чисел с плавающей точкой — некоторые вычисления очень неустойчивы и малейшее вариация исходных данных или промежуточных сильно сказывается на результате. Необходимо изобразить много разрядов целых чисел и очень большую мантиссу в случае с плавающей точкой. Есть много библиотек, таких как Bignum, которые выстраивают “эту радость” паровозиком. И есть специальные разработанные алгоритмы, которые позволяют с такими кортежами, интерпретируя их как число либо целое, либо с плавающей точкой, выполнять очень долго, но зато очень точно операции. Рунге-Ромберг, Ньютон, Лакс-Вендорф, дихотомия - частные случаи, но все же реализующие невысокую точность по сравнению с необходимой.

**Представление данных с фиксированной точкой и правила вычислений и преобразования.**

q - основание системы счисления

r - количество разрядов под дробную часть

n- количество разрядов под целую часть

**,** где - коэффициенты

**,**

При представлении числа в форме с фиксированной точкой указываются знак числа () и модуль числа () в q-ичном коде. Иногда такую форму представления чисел называют естественной формой. Место точки (запятой) постоянно для всех чисел и в процессе решения задач не меняется. Знак положительного числа кодируется цифрой «0», а знак отрицательного числа — цифрой «1».

*Определение 1.5.2*

Код числа в форме с фиксированной точкой, состоящий из кода знака и q-ичного кода его модуля, называется **прямым кодом**.

*Определение 1.5.3*

Разряд прямого кода числа, в котором располагается код знака, называется знаковым **разрядом кода**.

Разряды прямого кода числа, в которых располагается q-ичный код модуля числа, называются цифровыми разрядами кода. При записи прямого кода знаковый разряд располагается левее старшего цифрового разряда и обычно отделяется от цифровых разрядов точкой.

В общем случае разрядная сетка компьютера для размещения чисел в форме с фиксированной точкой показана на рис. 1.5.1.

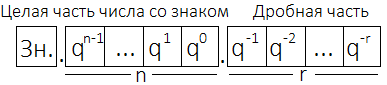


Рис. 1.5.1

На рисунке показано n разрядов для представления целой части числа и r разрядов — для дробной части числа. Использование формы с фиксированной точкой для представления смешанных (с целой и дробной частью) чисел в компьютерах практически не встречается. Как правило, используются компьютеры либо с дробной арифметикой (n=0), либо с целочисленной арифметикой (r=0).

Форма представления чисел с фиксированной точкой упрощает аппаратную реализацию компьютера, уменьшает время выполнения машинных операций, однако при решении задач на машине необходимо постоянно следить за тем, чтобы все исходные данные, промежуточные и окончательные результаты находились в допустимом диапазоне представления. Если этого не соблюдать, то возможно переполнение разрядной сетки, и результат вычислений будет неверным. От этих недостатков в значительной степени свободны компьютера, использующие форму представления чисел с плавающей точкой, или нормальную форму.

Пример числа с фиксированной точкой. Ячейка с записью целого числа на рис 1.5.2.

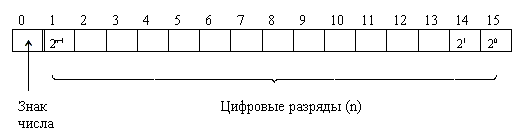


Рис 1.5.2

К достоинствам использования чисел с фиксированной точкой относятся простота выполнения арифметических операций и высокая точность изображения чисел. К недостаткам - небольшой диапазон представления чисел.

Пример числа с плавающей точкой:

2,62510 = 10,1012 = 0,101012\*2210 (нормальная форма)

Если число х положительное, то его прямой код представлен как х2.

Если число х отрицательное, то его прямой код представлен как 1 - х2.

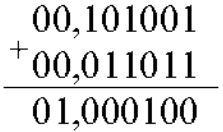
Правила вычислений:

Сложение

Операции сложения и умножения обычно выполняются над двоич­ными, числами, являющимися правильными дробями. Последователь­ность выполнения операции сложения следующая:

* исходные числа записываются в принятом для данной машины коде;
* производится поразрядное сложение кодов чисел, включая и зна­ковые разряды;
* производится анализ на переполнение разрядной сетки. В случае переполнения вырабатывается сигнал на прерывание программы.

Сложим числа: x=0,101001, y=0,011011.



Сочетание 01 в знаковых разрядах свидетельствует о переполнении.

Умножение

Умножение производится в столбик, как обычное умножение: путем сложения со сдвигами и последующим сложением этих чисел. Знак результирующего числа определяется путем сложения знаков изначальных чисел.

Пример: x=0,1101, y=1,1011. (Т.е. x - положительное число, у - отрицательное). Знак их произведения: 0 + 1 = 1 ⇒ число отрицательное.

|  |  |
| --- | --- |
| \* | \_\_\_\_1101 |
| \_\_\_\_1011 |
| + | \_\_\_\_1101 |
| \_\_\_1101\_ |
| \_\_0000\_\_ |
| \_1101\_\_\_ |
| = | 10001111 |

Ответ: ху = 1,1000111.

*Определение 1.5.4*

**Advanced Vector Extensions (AVX)** — расширение системы команд x86 для микропроцессоров Intel и AMD, предложенное Intel в марте 2008.

AVX предоставляет различные улучшения, новые инструкции и новую схему кодирования машинных кодов.

Улучшения расширения:

* Ширина векторных регистров [SIMD](https://ru.wikipedia.org/wiki/SIMD) увеличивается со 128 ([XMM](https://ru.wikipedia.org/wiki/SSE#%D0%A0%D0%B5%D0%B3%D0%B8%D1%81%D1%82%D1%80%D1%8B)) до 256 бит (регистры YMM0 — YMM15). Существующие 128-битные [SSE](https://ru.wikipedia.org/wiki/SSE)-инструкции будут использовать младшую половину новых YMM регистров, не изменяя старшую часть. Для работы с YMM-регистрами добавлены новые 256-битные AVX-инструкции. В будущем возможно расширение векторных регистров [SIMD](https://ru.wikipedia.org/wiki/SIMD) до 512 или 1024 бит. Например, процессоры с архитектурой [Xeon Phi](https://ru.wikipedia.org/wiki/Xeon_Phi) уже в 2012 году имели векторные регистры ([ZMM](https://ru.wikipedia.org/w/index.php?title=ZMM&action=edit&redlink=1)) шириной в 512 бит, и используют для работы с ними SIMD-команды с MVEX- и VEX-префиксами, но при этом они не поддерживают AVX.
* Неразрушающие операции. Набор AVX-инструкций использует трёхоперандный синтаксис. Например, вместо a=а+b можно использовать c=a+b, при этом регистр a остаётся неизменённым. В случаях, когда значение a используется дальше в вычислениях, это повышает производительность, так как избавляет от необходимости сохранять перед вычислением и восстанавливать после вычисления регистр, содержавший a, из другого регистра или памяти.
* Для большинства новых инструкций отсутствуют требования к выравниванию операндов в памяти. Однако рекомендуется следить за выравниванием на размер операнда, во избежание значительного снижения производительности.
* Набор инструкций AVX содержит в себе аналоги 128-битных [SSE](https://ru.wikipedia.org/wiki/SSE) инструкций для вещественных чисел. При этом, в отличие от оригиналов, сохранение 128-битного результата будет обнулять старшую половину YMM регистра. 128-битные AVX-инструкции сохраняют прочие преимущества AVX, такие, как новая схема кодирования, трехоперандный синтаксис и не выровненный доступ к памяти.
* Intel рекомендует отказаться от старых [SSE](https://ru.wikipedia.org/wiki/SSE) инструкций в пользу новых 128-битных AVX-инструкций, даже если достаточно двух операндов.

Таблица 1.5.1

|  |  |
| --- | --- |
| **Инструкция** | **Описание** |
| VBROADCASTSS, VBROADCASTSD, VBROADCASTF128 | Копирует 32-х-, 64-х- или 128-битный операнд из памяти во все элементы векторного регистра XMM или YMM. |
| VINSERTF128 | Замещает младшую или старшую половину 256-битного регистра YMM значением 128-битного операнда. Другая часть регистра-получателя не изменяется. |
| VEXTRACTF128 | Извлекает младшую или старшую половину 256-битного регистра YMM и копирует в 128-битный операнд-назначение. |
| VMASKMOVPS, VMASKMOVPD | Условно считывает любое количество элементов из векторного операнда из памяти в регистр-получатель, оставляя остальные элементы несчитанными и обнуляя соответствующие им элементы регистра-получателя. Также может условно записывать любое количество элементов из векторного регистра в векторный операнд в памяти, оставляя остальные элементы операнда памяти неизменёнными. |
| VPERMILPS, VPERMILPD | Переставляет 32-х или 64-х битные элементы вектора согласно операнду-селектору (из памяти или из регистра). |
| VPERM2F128 | Переставляет 4 128-битных элемента двух 256-битных регистров в 256-битный операнд-назначение с использованием непосредственной константы (imm) в качестве селектора. |
| VZEROALL | Обнуляет все YMM-регистры и помечает их как неиспользуемые. Используется при переключении между 128-битным режимом и 256-битным. |
| VZEROUPPER | Обнуляет старшие половины всех регистров YMM. Используется при переключении между 128-битным режимом и 256-битным. |

[**Понятие о масштабируемой разрядности**](#44sinio)**:**

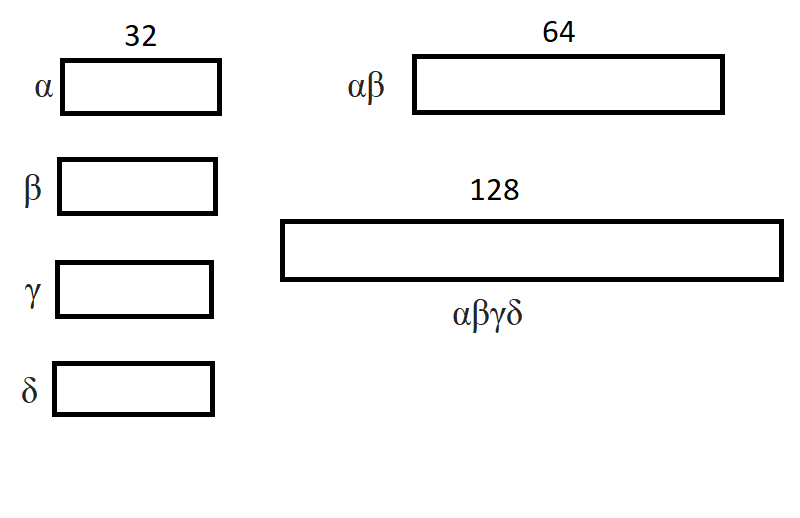
Есть регистры α, β, γ, δ каждый — 32 бита, если обычные типы long и unsigned, то мы этими регистрами оперируем независимо, но оказывается возможным оперировать этими регистрами — парами регистров, когда эти регистры выстроены в картежи(выставлены друг за другом). Они полноценно работают с 64 разрядом, а есть регистры, которые работают по четверкам — α, β, γ, δ (соотв. разрядность 128, Рис.1.5.3). В итоге можно сказать, что масштабируемая разрядность это — использование как будто раздельных регистров вместе — над парами или четверками осуществляется одна и та же операция, понимая, что справа — младшие разряды, а слева — старшие.

Рис. 1.5.3

# 1.6. Программная среда специализированного вычислителя

*Определение 1.6.1*

**BIOS (частный случай)** (Basic Input Output System) - базовая система ввода-вывода; набор микропрограмм, реализующих [API](https://ru.wikipedia.org/wiki/%D0%98%D0%BD%D1%82%D0%B5%D1%80%D1%84%D0%B5%D0%B9%D1%81_%D0%BF%D1%80%D0%BE%D0%B3%D1%80%D0%B0%D0%BC%D0%BC%D0%B8%D1%80%D0%BE%D0%B2%D0%B0%D0%BD%D0%B8%D1%8F_%D0%BF%D1%80%D0%B8%D0%BB%D0%BE%D0%B6%D0%B5%D0%BD%D0%B8%D0%B9) для работы с аппаратурой компьютера и подключёнными к нему устройствами. В состав этой системы входят различные программы ввода-вывода, которые обеспечивают взаимодействие между операционной системой, прикладными программами с одной стороны и устройствами, входящими в состав компьютера (внутренними и внешними) с другой.

BIOS бывает для материнских плат и для периферийных устройств.

В настоящее время BIOS представляет собой сложную систему, состоящую из большого количества утилит, предназначенных для автоматического распознавания, установленного на компьютер оборудования, его настройки и проверки функционирования. Вызов программ BIOS, как правило, осуществляется через программные или аппаратные прерывания. При включении питания компьютера BIOS тестирует (POST - Power-On-Self-Test) компоненты системы - процессор, память, приводы дисков (как жестких, так и флоппи-дисководов), клавиатуру т.д.

BIOS реализован в виде микросхемы, установленной на материнской плате компьютера. Заметим, что название ROM BIOS в настоящее время не совсем справедливо, ибо "ROM" предполагает использование постоянных запоминающих устройств (Read Only Memory), а для хранения кодов BIOS в настоящее время применяют в основном перепрограммируемые запоминающие устройства. Наиболее перспективной для хранения системы BIOS является флэш-память. Она позволяет модифицировать функции для поддержки новых устройств, подключаемых к компьютеру.

При каждом включении питания компьютера типа IBM PC (или совместимого с ним) и до начала загрузки операционной системы процессор компьютера выполняет процедуру BIOS под названием "Самотест по включению питания" - POST (Power On Self Test). Основной целью процедуры POST является проверка базовых функций и подсистем компьютера (таких как память, процессор, материнская плата, видеоконтроллер, клавиатура, гибкий и жесткий диски и т. д.) перед загрузкой операционной системы. Это в некоторой степени застраховывает пользователя от попытки работать на неисправной системе, что могло бы привести, например, к разрушению пользовательских данных на HDD.

*Определение 1.6.2*

**Firmware** – специализированное программное обеспечение внутри техники с микроконтроллером. Почти все электронные устройства кроме простейших содержат Firmware: компьютерные мыши, клавиатуры, жесткие диски, маршрутизаторы, современные модели автомобилей, телевизор, стиральная машина и т.д.

Окружение времени выполнения (run-time environment) программных модулей - RTE. Существуют обычные ОС, если их упрощать и убирать многие функции, то получается ОСРВ (RTOS - Real Time Operating System). При убирании почти всех функций получается RTE. RTE - небольшое количество функций, которые уже не являются операционной системой и часто стоят на различных специфичных ЭВС. Могут включать в себя до пары десятков функций. ОСРВ, в отличие от RTE, может иметь какую-то вариативность в действиях, другими словами универсальность.

*Определение 1.6.3*

**Встраиваемые операционные системы** — операционные системы, предназначенные для управления специализированными устройствами и вследствие этого способные работать в условиях ограниченных ресурсов (малые объемы памяти, недостаток вычислительных мощностей и т.п.) и в необслуживаемом режиме. Характерными особенностями встраиваемых ОС являются модульная структура, компактность, производительность, масштабируемость и повышенная отказоустойчивость.

*Определение 1.6.4*

**Операционные системы реального времени (ОСРВ (RTOS))** — операционные системы, способные обеспечить предсказуемое время обработки непредсказуемо возникающих внешних событий. Разделяют ОС «жесткого» и «мягкого» реального времени: для первых временные характеристики гарантированы, и выход за их пределы расценивается как отказ, для вторых временные ограничения, как правило, соблюдаются, и выход за их пределы считается снижением производительности. Большинство современных ОСРВ являются встраиваемыми. ОСРВ и RTE отличаются от обычных ОС областью применения, возможностями (функциональностью), скоростью работы и отказоустойчивостью. ОСРВ, например, могут не иметь возможности воспроизводить звук (и кучи других функций), но могут работать быстрее и надёжней в условиях, в которых ОС общего назначения затормозит. Аналогичный переход от ОСРВ к RTE. OS → RTOS → RTE.

Таблица 1.6.1

|  |  |
| --- | --- |
| ОС (OS) | Обычная операционная система.  Пример использования: на пк, ноутбуках, некоторых планшетах (Windows, Ubuntu, Android). |
| ОСРВ (RTOS - real time OS) | Обкусанная операционная система, из которой выкинуты все лишние функции, но в то же время сохраняется некоторая универсальность (свобода действий, возможность решения широкого круга задач)  Пример использования: мосты, шлюзы, маршрутизаторы (RTOS Linux). |
| ОВВ (RTE - run-time environment) | Это уже нельзя назвать операционной системой, потому что она обкусана до такой степени, что может решать только строго определнный узкий круг задач. Нет динамического выделения памяти, есть только стек, нет кеша. Нет перефирических устройств, кроме портов: с одного мы всегда берем, на другой отдаем. Там нет никаких функций отладки.  Пример использования: GPS-навигация, сотовые станции. |

# 1.7. Использование компьютера общего назначения для разработки программного обеспечения для специализированных вычислителей. Кросс-средства

*Определение 1.7.1*

**Компьютер общего назначения** — компьютер, способный решить любую задачу, которая может быть выражена в виде программы и выполнена в рамках разумных ограничений, накладываемых ёмкостью системы хранения компьютера, допустимым размером программы, скоростью её выполнения и надёжностью оборудования.

В отличие от специализированных вычислительных устройств компьютер общего назначения способен выполнять множество зачастую не связанных между собой функций. Так суперкомпьютеры общего назначения способны обслуживать программные приложения, разработанные для самых разных и далеко отстоящих друг от друга направлений научных исследований, таких как лингвистика и астрономия, науки о Земле и проектирование техники. Знакомый всем персональный компьютер тоже является примером компьютера общего назначения, сегодня он используется не только как вычислительное устройство, но и как интеллектуальный офисный инструмент, цифровая аудио-видео-студия или центр развлечений.

Более узко определение «компьютер общего назначения» может быть формализовано в требовании, чтобы конкретный компьютер был способен подражать поведению универсальной машины Тьюринга.

*Определение 1.7.2*

**Универсальной машиной Тьюринга** называют [машину Тьюринга](https://ru.wikipedia.org/wiki/%D0%9C%D0%B0%D1%88%D0%B8%D0%BD%D0%B0_%D0%A2%D1%8C%D1%8E%D1%80%D0%B8%D0%BD%D0%B3%D0%B0), которая может заменить собой любую машину Тьюринга(абстрактный исполнитель, рис 1.7.1). Получив на вход программу и входные данные, она вычисляет ответ, который вычислила бы по входным данным машина Тьюринга, чья программа была дана на вход.

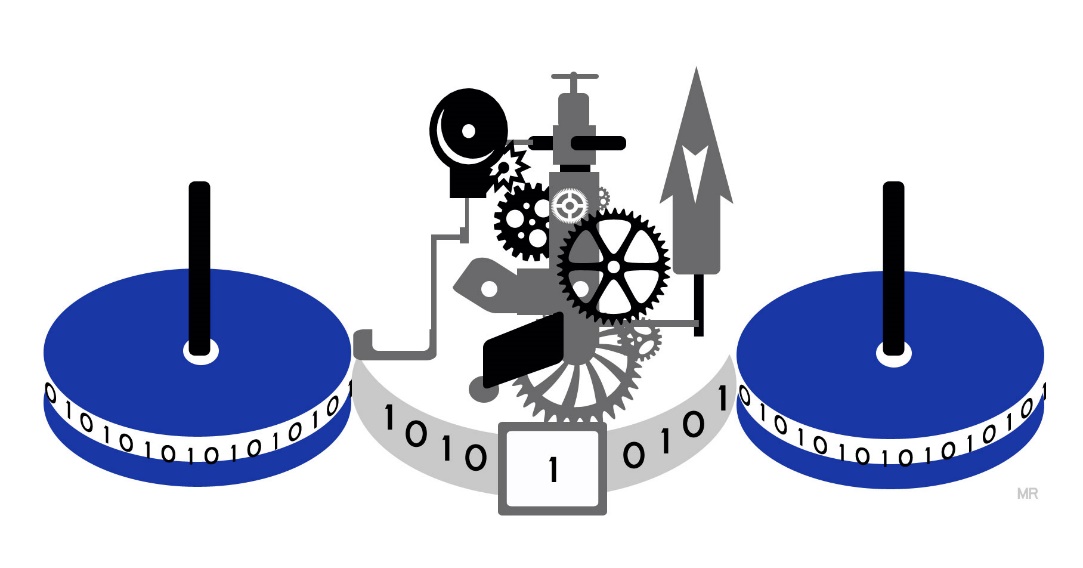


Рис. 1.7.1

*Определение 1.7.3*

**Специализированная вычислительная машина** (специализированный вычислитель) — вычислительная машина, предназначенная для решения одной задачи или узкого круга задач. Специализация такой машины повышает эффективность средств вычислительной техники, поскольку структурная и аппаратная интерпретация программ способствует повышению точности и быстродействия устройств, упрощает математическое обеспечение, снижает аппаратные затраты.

*Определение 1.7.4*

**Прикладное программное обеспечение** (программное обеспечение, использующееся в специализированных вычислителях) – это комплекс программных средств и документации к ним, предназначенных для решения сравнительно узких классов задач в конкретных предметных областях, рассчитанных на определенного потребителя: научно-технических, экономических, инженерных, конструкторских и других специальных задач в различных сферах человеческой деятельности. Оно включает в себя текстовые, графические, музыкальные редакторы, игры, электронные таблицы, математические пакеты и т.д., которые служат для создания документов, рисунков, для обработки табличных данных и массивов информации, для различных расчетов и игр.

Примеры важнейших прикладных программ: Word, Excel,Works, Лексикон, Paint Brush, AutoCad. Прикладные программы делятся на программы общего назначения и специального назначения:

**Программы общего назначения:**

* текстовые и графические редакторы
* системы управления базами данных
* табличные процессоры
* коммуникационные (сетевые) программы
* компьютерные игры

**Программы специального назначения:**

* бухгалтерские пакеты
* системы автоматизированного проектирования
* экспертные системы
* программы для проведения сложных математических расчетов
* многие другие программы для профессиональной деятельности

*Определение 1.7.5*

**Кросс-средства разработки (К.-с. р.)**

Средства, позволяющие разрабатывать программное обеспечение для некоторой вычислительной системы с использованием инструментальной системы с другой архитектурой.

Широкое применение **К.-с. р.** находят при разработке встроенных систем на основе микроконтроллеров, построение инструментальной системы на которых затруднено в связи с особенностями их архитектуры. При этом **К.-с. р.** работают на инструментальном персональном компьютере, а результирующий код загружается в целевую систему при помощи программатора.

В качестве примера **К.-с. р.** Можно привести широкораспространенную интегрированную среду разработки Atmel AVR Studio 5.0 фирмы Atmel. Это средство предназначено для работы на IBM PC-совместимых компьютерах в среде операционной системы MS Windows и позволяет разрабатывать и отлаживать программы для всех микроконтроллеров семейства AVR на ассемблере и языке программирования C.

Рассмотрим пример, который показывает использование компьютера общего назначения для разработки программного обеспечения для специализированных вычислителей. Например, нам нужно разработать холодильник и нужно разработать программный код для специализированного вычислителя, который будет управлять холодильником. Ну ведь его же надо как-то разработать, ни клавиатуры, ни мышки подключить не удастся, поэтому при помощи кросс-средств (переходника) подключаем плату со специализированным процессором со своим источником питания к большой машине. Можно туда залить код, посмотреть, как он мигает диодами, можно с нажатия клавиш клавиатуры имитировать открытие дверей, отключение дверей и др. Т.е. через кросс-средства можно имитировать какой программе, написанной на С на большом компьютере, вводя с клавиатуры разные события. Они через шланг забиваются туда, и контролер холодильника думает, что произошло прерывание. Так и происходит отладка, а мы считываем с него информацию, его реакцию на наши имитации, зажег ли он нужные лампочки и др.

*Определение 1.7.6*

**Аппаратное обеспечение**, **аппаратные средства** (англ. *hardware*) — электронные и механические части вычислительного устройства, входящие в состав системы или сети, исключая программное обеспечение и данные (информацию, которую вычислительная система хранит и обрабатывает). Аппаратное обеспечение включает: компьютеры и логические устройства, внешние устройства и диагностическую аппаратуру, энергетическое оборудование, батареи и аккумуляторы.

*Определение 1.7.7*

**Аппаратное обеспечение** вычислительных систем — обобщённое название оборудования, на котором работают компьютеры и сети компьютеров.

К аппаратному обеспечению обычно относят:

* центральный процессор (процессоры)
* оперативную память
* системную логику
* периферийные устройства
* сетевое оборудование

Рассмотрим пример, который показывает использование компьютера общего назначения для разработки аппаратной части специализированных вычислителей. Программируя какую-то плату, например, FPGA, чтобы он выполнял какую-то задачу (считал поля Галуа и др.), есть большая вероятность человеческой ошибки, поэтому нужно делать отладку. Для этого на заданный порт FPGA посылаем контрольные операнды (например, в поле Галуа 16 нужно то-то умножить на то-то). Зная заранее правильный ответ, сравниваем его с ответом выданном выходным портом FPGA платы. Допустим есть некая разница, анализируем, что не так запрограммировано (там есть свой отладчик), перепрограммируем само железо, так как оно много раз перепрограммируемо. Есть языки, на которых можно описывать логически, как устроено железо. Смысл состоит в том, что при помощи перепрограммируемой логики, отлаживаем так, как будто это программа. И когда уже все отлажено, код правильный, отсылаем на разработку, после которой создаются уже непрограммируемые схемы на основе этой.

# Глава 2. Процессор (компьютер)

# 2.1. Основные узлы процессоров

*Определение 2.1.1*

**Регистры** - специальные ячейки памяти, расположенные физически внутри процессора, доступ к которым осуществляется не по адресам, как к основной памяти, а по именам.

Они позволяют управлять выполнением программы, хранить временные результаты и т.д.

Регистры уровня архитектуры команд делятся на два типа:

1. Регистры общего назначения - для запоминания данных и/или операндов при исполнении команд.
2. Специальные регистры - выполнение специальных функций при работе процессора.

Доступ к значениям, хранящимся в регистрах, как правило, в несколько раз быстрее, чем доступ к ячейкам оперативной памяти, но объём оперативной памяти намного превосходит суммарный объём регистров.

Регистры EAX, EBX, ECX, EDX – это регистры общего назначения. Они имеют определённое назначение (так уж сложилось исторически), однако в них можно хранить любую информацию.

Регистры EBP, ESP, ESI, EDI – это также регистры общего назначения. Они имеют уже более конкретное назначение. В них также можно хранить пользовательские данные, но делать это нужно уже более осторожно, чтобы не получить «неожиданный» результат.

Следует отметить, что регистры могут быть неравнозначны и при использовании определенных инструкций могут иметь специальное значение:

* EAX - аккумулятор, операнд-источник или приемник результата (некоторые инструкции могут быть короче на один байт при использовании EAX);
* EBX - указатель на данные в сегменте DS;
* ECX - счетчик для цепочечных (например, MOVS) и циклических (с префиксом REP и т.п.) инструкций;
* EDX - адрес порта ввода-вывода для инструкций IN/INS, OUT/OUTS;
* ESI - указатель на операнд-источник в сегменте DS для цепочечных инструкций;
* EDI - указатель на операнд-приемник в сегменте ES для цепочечных инструкций;
* EBP - указатель на данные в сегменте SS.

*Определение 2.1.2*

**Счётчик команд** – специальный регистр, в котором хранится адрес команды, которая должна быть выполнена после выполнения текущей команды.

Регистр флагов носит название **EFLAGS**. Это 32-разрядный регистр. Однако старшие 16 разрядов используются при работе в защищённом режиме. К младшим 16 разрядам этого регистра можно обращаться как к отдельному регистру с именем FLAGS.

*Определение 2.1.3*

**Флаг** – это один или несколько битов памяти, которые могут принимать двоичные значения (или комбинации значений) и характеризуют состояние какого-либо объекта. Обычно флаг может принимать одно из двух логических значений.

**Свойства регистров:**

* Регистры нужны для вычислений и связи ЦП с внешним миром.
* Большая часть кода программ состоит из команд копирования значений из оперативной памяти в регистры и обратно.
* Во всех операциях программы без регистров не обойтись.
* И если даже данные не хранятся в регистрах, то в них обязательно будут указатели на эти данные (адреса данных в памяти), других способов работы с данными у процессора нет, только через собственные регистры. А если речь идёт исключительно о сторонних устройствах, то мы тоже используем регистры, только это уже не регистры процессора, а, например, регистры PCI, PCIe, SATA и так далее.
* К доступным регистрам пользователь может обращаться с помощью команд машинного языка. К этим регистрам, как правило, имеют доступ все программы — как приложения, так и системные. Обычно среди доступных регистров есть регистры данных, адресные регистры и регистры кода условия.

*Определение 2.1.4*

**Регистры данных –** регистры, которые служат для хранения промежуточных вычислений.

*Определение 2.1.5*

**Адресные регистры** –применяются для хранения адреса (или его части) ячейки в оперативной памяти.

*Определение 2.1.6*

**Индексный регистр** – регистр, который используется в обычном режиме адресации, когда адрес получается в результате сложения содержимого индексного и базового регистра.

*Определение 2.1.7*

**Сегментные регистры** –регистры, указывающие на сегменты.

При сегментной адресации память разделяется на блоки (сегменты), состоящие из различного количества машинных слов. Адрес ячейки памяти складывается из адреса сегмента и смещения относительно начала сегмента.

При этом режиме адресации базовый адрес сегмента (его начало) хранится в одном из регистров. Таких регистров может быть несколько; например, один — для операционной системы (т.е. использующийся при выполнении процессором кода операционной системы), другие — для исполняющихся в данный момент приложений.

*Определение 2.1.8*

**Регистр стека –** регистр, указывающий на стек.

При стековой адресации выделяется специальный регистр, в котором размещен указатель на вершину стека. Этот режим адресации позволяет использовать некоторые команды, в которых отсутствует поле адреса, например, push и pop.

В состав всех процессоров входит также регистр (или набор регистров), известный под названием **регистра слова состояния программы** (program status word — PSW). В нем, как правило, содержатся коды условий и другая информация о состоянии, например, бит разрешения/запрещения прерываний или бит режима системный/пользовательский.

*Определение 2.1.9*

**Коды условий (известные также как флаги)** – это последовательность битов, устанавливаемых или сбрасываемых процессором в зависимости от результата выполненных операций.

Задача **центрального процессора –** выполнять программы, находящиеся в основной памяти. Для этого он вызывает команды из памяти, определяет их тип, выполняет одну за другой.

Большинство компьютеров имеет как **внутренние**, так и **внешние** шины.

*Определение 2.1.10*

**Внутренняя шина** **–** шина, которая подключает все внутренние компоненты компьютера к материнской плате (и, следовательно, к процессору и памяти). Такой тип шин также называют локальной шиной, поскольку она служит для подключения локальных устройств.

*Определение 2.1.11*

**Внешняя шина** **-** шина, которая подключает внешнюю периферию к материнской плате.

*Определение 2.1.12*

**Шина адреса** **-** [компьютерная шина](https://ru.wikipedia.org/wiki/%D0%A8%D0%B8%D0%BD%D0%B0_(%D0%BA%D0%BE%D0%BC%D0%BF%D1%8C%D1%8E%D1%82%D0%B5%D1%80%D1%8B)), используемая центральным [процессором](https://ru.wikipedia.org/wiki/%D0%9F%D1%80%D0%BE%D1%86%D0%B5%D1%81%D1%81%D0%BE%D1%80) или устройствами, способными инициировать сеансы [DMA](https://ru.wikipedia.org/wiki/DMA)(Прямой доступ к памяти **-** [англ.](https://ru.wikipedia.org/wiki/%D0%90%D0%BD%D0%B3%D0%BB%D0%B8%D0%B9%D1%81%D0%BA%D0%B8%D0%B9_%D1%8F%D0%B7%D1%8B%D0%BA) direct memory access ), для указания физического адреса слова [ОЗУ](https://ru.wikipedia.org/wiki/%D0%9E%D0%97%D0%A3) (или начала блока слов), к которому устройство может обратиться для проведения операции чтения или записи.

**Как работает шина:**

Некоторые устройства, соединенные с шиной, являются активными и могут инициировать передачу информации по шине, другие - пассивные и ждут запросов. Активное устройство называется задающим, пассивное – подчиненным.

Пример: когда центральный процессор требует от контроллера диска считать или записать блок информации, центральный процессор действует как задающее устройство, а контроллер диска как подчиненное. Контроллер диска может действовать как задающее устройство, когда он командует памяти принять слова, которые считал с диска.

**Внешние формирователи**: Большинство задающих устройств связаны с шиной через микросхему, которая называется **драйвером** шины и по существу является цифровым усилителем. Большинство же подчиненных устройств связаны с шиной **приемником** шины.

Для устройств, которые могут быть и задающим и подчиненным устройством используется **приемопередатчик** шины (трансивер).

Регистры вместе с АЛУ формируют тракт данных, по которому поступают данные. Базовая операция тракта данных выполняется следующим образом: выбирается один или два регистра, АЛУ производит над ними какую-либо операцию, после чего результат вновь помещается в какой-либо регистр.

АЛУ может выполнять следующие операции:

* арифметические операции сложения, вычитания, умножения и деления
* логические операции И, ИЛИ, исключающее ИЛИ
* операции циклического сдвига, сброса, инвертирования

В простейшем случае АЛУ состоит из двух регистров: сумматора с соответствующими логическими схемами и элемента управления выполняемым процессом. Устройство работает в соответствии с сообщаемыми ему именами (кодами) операций, которые при пересылке данных нужно выполнить над переменными, помещаемыми в регистры.

Арифметико-логическое устройство функционально можно разделить на две части:

а) микропрограммное устройство (устройство управления), задающее последовательность микрокоманд (команд);

б) операционное устройство (АЛУ), в котором реализуется заданная последовательность микрокоманд (команд).

Схема взаимодействия АЛУ с другими узлами микропроцессора на рис 2.1.1:

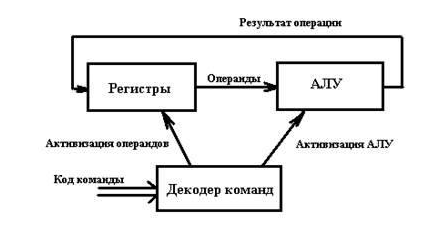


рис 2.1.1

Важной особенностью АЛУ является его способность оперировать не только байтами, но и битами. Отдельные программно-доступные биты могут быть установлены, сброшены, инвертированы, переданы, проверены и использованы в логических операциях. Эта способность достаточно важна, поскольку для управления объектами часто применяются алгоритмы, содержащие операции над входными и выходными булевыми переменными, реализация которых средствами обычных микропроцессоров сопряжена с определенными трудностями.

АЛУ обычно работают только с положительными целыми числами. Однако при выполнении вычитания получаются отрицательные числа, если вычитаемое больше уменьшаемого. Для представления отрицательных чисел ис­пользуется дополнительный код («дополнение до двух»).

**Выполнение команд**

Центральный процессор (ЦП) выполняет каждую команду за несколько шагов. Он делает следующее:

1. Вызывает следующую команду из памяти и переносит ее в регистр команд.
2. Меняет положение счетчика команд, который после этого указывает на следующую команду.
3. Определяет тип вызванной команды.
4. Если команда использует слово из памяти, определяет, где находится это слово.
5. Переносит слово, если это необходимо, в регистр центрального процессора.
6. Выполняет команду.
7. Переходит к шагу 1, чтобы начать выполнять следующие команды.

Программа, имитирующая работу центрального процессора, называется **интерпретатором**.

**Устройство управления**

Чтобы обеспечить автоматические вычисления по программе, процессор должен уметь выполнять еще ряд дополнительных действий:

* извлекать из памяти очередную команду;
* расшифровывать ее и преобразовывать в последовательность стандартных элементарных действий;
* заносить в АЛУ исходные данные;
* сохранять полученный в АЛУ результат;
* обеспечивать синхронную работу всех узлов машины.

Для выполнения этих функций служит **устройство управления (УУ)**.

УУ содержит несколько важных регистров для хранения информации, необходимой в ходе выполнения текущей команды.

Регистр команды – служит для размещения текущей команды, которая находится в нем в течение текущего цикла процессора.

Кроме этого, имеются регистры, содержащие адрес команды, счетчик адреса команды, адреса операндов, операнды и результаты выполнения команды.

**Планировщик очередей**

Очередь — буфер, работающий по принципу FIFO.

buffer, буфер — общее название структуры, разделяющей поток данных (в т. ч. между стадиями конвейера). Если буфер содержит более одного слова, то оформлен в виде очереди или полно ассоциативной памяти и в таком виде позволяет сглаживать неравномерность поступления потока данных относительно его приёма.

*Определение 2.1.13*

**Ассоциативная память (АП)** или **ассоциативное запоминающее устройство (АЗУ)** является особым видом машинной памяти, используемой в приложениях очень быстрого поиска. Известна также как память, адресуемая по содержимому, ассоциативное запоминающее устройство, контентно-адресуемая память или ассоциативный массив, хотя последний термин чаще используется в программировании для обозначения структуры данных.

*Определение 2.1.14*

**FIFO** (first-in, first-out: первым пришёл, первым вышел) — принцип работы буфера, при котором чтение слов происходят в порядке их записи.

*Определение 2.1.15*

**Очередь** — это структура, разделяющая поток данных по принципу того, что чтение слов происходит в порядке их записи.

*Определение 2.1.16*

**Математический сопроцессор** — сопроцессор для расширения командного множества центрального процессора и обеспечивающий его функциональностью модуля операций с плавающей запятой, для процессоров, не имеющих интегрированного модуля.

Модуль операций с плавающей запятой (или с плавающей точкой; англ. floating point unit) — часть процессора для выполнения широкого спектра математических операций над вещественными числами.

Простым «целочисленным» процессорам для работы с вещественными числами и математическими операциями требуются соответствующие процедуры поддержки и время для их выполнения. Модуль операций с плавающей запятой поддерживает работу с ними на уровне примитивов — загрузка, выгрузка вещественного числа (в/из специализированных регистров) или математическая операция над ними выполняется одной командой, за счёт этого достигается значительное ускорение таких операций.

**Блок предсказания переходов**

Эффективность алгоритмов предсказания переходов критична для архитектур, где используется высокий уровень параллелизма инструкций. Переходы разрывают параллелизм, поскольку необходимо ждать результат предыдущей инструкции, прежде чем продолжать выполнение потока инструкций. Предсказание переходов определяет возможность того, какая будет взята инструкция, и если предсказание верно, то вычисляется дальнейший адрес выполнения. Для этого требуется Branch Target Buffer (BTB), который сохраняет результаты ветвлений по мере продолжения выполнения кода. К массиву прилагается алгоритм определения результата следующего ветвления.

Декодеры преобразуют команды в макрокоманды, которые являются совокупностью команд, которая часто встречается в программах, под управлением которых в процессоре выполняются элементарные операции. Каждая простая инструкция преобразуется в 1–2 микрокоманды, которые являются совокупностью кодов микроопераций, управляющих схемами в течение одного машинного такта, а для сложной инструкции из памяти микрокода выбирается последовательность микрокоманд, которая содержит более двух микрокоманд. Используя технологию macrofusion, четыре декодера могут обработать одновременно пять команд, преобразуя их в четыре микрокоманды. Данная технология заключается в том, что ряд пар связанных между собой последовательных инструкций, таких как, например, сравнение со следующим за ним условным переходом, представляются декодером одной микрокомандой. Таким образом, технология macrofusion позволяет закодировать одной микрооперацией две команды. В то время, как технология microfusion позволяет закодировать одной микрооперацией две операции, чтобы снизить нагрузку на конвейер для некоторых сложных команд.

# 2.2. Ядро процессора

*Определение 2.2.1*

**Ядро процессора** – это его основная часть, содержащая все функциональные блоки и осуществляющая выполнение всех логических и арифметических операций.

*Определение 2.2.2*

**Постоянное запоминающее устройство (ПЗУ)** — энергонезависимая память, используется для хранения массива неизменяемых данных

Ядро процессора состоит из нескольких функциональных блоков:

* блока выборки инструкций;
* блоков декодирования инструкций;
* блоков выборки данных;
* управляющего блока;
* блоков выполнения инструкций;
* блоков сохранения результатов;
* блока работы с прерываниями;
* ПЗУ, содержащего микрокод;
* набора регистров;
* счетчика команд.

**Блок выборки инструкций** осуществляет считывание инструкций по адресу, указанному в счетчике команд. Обычно, за такт он считывает несколько инструкций. Количество считываемых инструкций обусловлено количеством блоков декодирования, так как необходимо на каждом такте работы максимально загрузить блоки декодирования. Для того чтобы блок выборки инструкций работал оптимально, в ядре процессора имеется предсказатель переходов.

**Предсказатель переходов** пытается определить, какая последовательность команд будет выполняться после совершения перехода. Это необходимо, чтобы после условного перехода максимально нагрузить конвейер ядра процессора.

**Блоки декодирования**, как понятно из названия, – это блоки, которые занимаются декодированием инструкций, т.е. определяют, что надо сделать процессору, и какие дополнительные данные нужны для выполнения инструкции. Задача эта для большинства современных коммерческих процессоров, построенных на базе концепции CISC, – очень сложная. Дело в том, что длина инструкций и количество операндов – нефиксированные, и это сильно усложняет жизнь разработчикам процессоров и делает процесс декодирования нетривиальной задачей.

**Блоки выборки данных** осуществляют выборку данных из КЭШ-памяти или ОЗУ, необходимых для выполнения текущих инструкций. Обычно, каждое процессорное ядро содержит несколько блоков выборки данных. Например, в процессорах Intel Core используется по два блока выборки данных для каждого ядра.

**Управляющий блок** на основании декодированных инструкций управляет работой блоков выполнения инструкций, распределяет нагрузку между ними, обеспечивает своевременное и верное выполнение инструкций. Это один из наиболее важных блоков ядра процессора.

**Блоки выполнения инструкций** включают в себя несколько разнотипных блоков:

ALU – арифметическое логическое устройство;

FPU – устройство по выполнению операций с плавающей точкой;

**Блок сохранения результатов** обеспечивает запись результата выполнения инструкции в ОЗУ по адресу, указанному в обрабатываемой инструкции.

**Блок работы с прерываниями.** Работа с прерываниями – одна из важнейших задач процессора, позволяющая ему своевременно реагировать на события, прерывать ход работы программы и выполнять требуемые от него действия. Благодаря наличию прерываний, процессор способен к псевдопараллельной работе, т.е. к, так называемой, многозадачности.

Обработка прерываний происходит следующим образом. Процессор перед началом каждого цикла работы проверяет наличие запроса на прерывание. Если есть прерывание для обработки, процессор сохраняет в стек адрес инструкции, которую он должен был выполнить, и данные, полученные после выполнения последней инструкции, и переходит к выполнению функции обработки прерывания.

После окончания выполнения функции обработки прерывания, из стека считываются сохраненные в него данные, и процессор возобновляет выполнение восстановленной задачи.

*Определение 2.2.3*

**Счетчик команд** – регистр, содержащий адрес команды, которую процессор начнет выполнять на следующем такте работы.

В упрощенном виде этапы цикла работы ядра процессора можно представить следующим образом:

1. Блок выборки инструкций проверяет наличие прерываний. Если прерывание есть, то данные регистров и счетчика команд заносятся в стек, а в счетчик команд заносится адрес команды обработчика прерываний. По окончанию работы функции обработки прерываний, данные из стека будут восстановлены;
2. Блок выборки инструкций из счетчика команд считывает адрес команды, предназначенной для выполнения. По этому адресу из КЭШ-памяти или ОЗУ считывается команда. Полученные данные передаются в блок декодирования;
3. Блок декодирования команд расшифровывает команду, при необходимости используя для интерпретации команды записанный в ПЗУ микрокод. Если это команда перехода, то в счетчик команд записывается адрес перехода и управление передается в блок выборки инструкций (пункт 1), иначе счетчик команд увеличивается на размер команды (для процессора с длинной команды 32 бита – на 4) и передает управление в блок выборки данных;
4. Блок выборки данных считывает из КЭШ-памяти или ОЗУ требуемые для выполнения команды данные и передает управление планировщику;
5. Управляющий блок определяет, какому блоку выполнения инструкций обработать текущую задачу, и передает управление этому блоку;
6. Блоки выполнения инструкций выполняют требуемые командой действия и передают управление блоку сохранения результатов;
7. При необходимости сохранения результатов в ОЗУ, блок сохранения результатов выполняет требуемые для этого действия и передает управление блоку выборки инструкций (пункт 1).

Описанный выше цикл называется процессом (именно поэтому процессор называется процессором). Последовательность выполняемых команд называется программой.

Скорость перехода от одного этапа цикла к другому определяется тактовой частотой процессора, а время работы каждого этапа цикла и время, затрачиваемое на полное выполнение одной инструкции, определяется устройством ядра процессора.

*Определение 2.2.4*

**Многоядерный процессор** — центральный процессор, содержащий два и более вычислительных ядра на одном процессорном кристалле или в одном корпусе.

В многоядерных процессорах тактовая частота, как правило, намеренно снижена. Это позволяет уменьшить энергопотребление процессора без потери производительности. В некоторых процессорах тактовая частота каждого ядра может меняться в зависимости от его индивидуальной нагрузки. Ядро является полноценным микропроцессором, использующим все достижения микропроцессорной техники: конвейеры, внеочередное исполнение кода, многоуровневый кэш, поддержка векторных команд.

Каждое ядро также может использовать технологию SMT для поочередного исполнения нескольких потоков, создавая иллюзию нескольких «логических процессоров» на основе каждого ядра.

*Определение 2.2.5*

**Многопоточность** (англ. Multithreading — SMT) — свойство платформы (например, операционной системы, виртуальной машины и т.д.) или приложения, состоящее в том, что процесс, порождённый в операционной системе, может состоять из нескольких потоков, выполняющихся «параллельно», то есть без предписанного порядка во времени. При выполнении некоторых задач такое разделение может достичь более эффективного использования ресурсов вычислительной машины.

*Определение 2.1.6*

**Одновременная многопоточность** (англ. Simultaneous Multithreading — SMT) — одна из двух главных форм многопоточности, которая может быть реализована в процессорах аппаратно. Второй формой является временная многопоточность. Технология одновременной многопоточности позволяет исполнять инструкции из нескольких независимых потоков выполнения на множестве функциональных модулей суперскалярного микропроцессора в одном цикле.

*Определение 2.2.7*

**Временная многопоточность** (англ. Temporal multithreading). Различие между этими двумя формами состоит в максимальном количестве потоков, которые исполняются на каждой стадии вычислительного конвейера в определенный тактовый цикл процессора. При временной многопоточности в каждый данный момент исполняется только один поток, а при одновременной многопоточности — несколько. Некоторые специалисты используют термин super-threading в качестве синонима временной многопоточности.

На обычном процессоре управление потоками осуществляется операционной системой. Поток исполняется до тех пор, пока не произойдет аппаратное прерывание, системный вызов или пока не истечёт отведённое для него операционной системой время. После этого процессор переключается на код операционной системы, который сохраняет состояние потока (его контекст) и переключается на состояние следующего в очереди потока, которому тоже выделяется время на исполнение. При такой многопоточности достаточно большое количество тактов процессора тратится на код операционной системы, переключающий контексты. Если поддержку потоков реализовать аппаратно, то процессор сам сможет переключаться между потоками, а в идеальном случае — выполнять несколько потоков одновременно за каждый такт.

Одновременная многопоточность объединяет параллельное исполнение инструкций суперскалярной архитектуры с аппаратной многопоточностью. Применение одновременной многопоточности, благодаря динамическому распределению функциональных модулей процессора между потоками, увеличивает использование процессора при наличии задержек памяти и ограниченной возможности параллельного исполнения инструкций внутри потока.

Многоядерные микропроцессоры по своей организации наиболее близки к микропроцессорам с одновременной многопоточностью — имеют несколько наборов регистров, несколько функциональных модулей и суперскалярность каждого из ядер. Главное отличие между ними заключается в распределении ресурсов — в многоядерном процессоре каждый поток получает фиксированное количество функциональных модулей процессора, тогда как в процессоре с одновременной многопоточностью распределение модулей изменяется в каждом цикле. Вследствие этого процессоры с одновременной многопоточностью показывают большую производительность при максимальной загрузке потоками, а при снижении количества потоков производительность падает медленней, по сравнению с многоядерным процессором.

**Кэш-память**: во всех существующих на сегодня многоядерных процессорах кэш-памятью 1-го уровня обладает каждое ядро в отдельности, а кэш-память 2-го уровня существует в нескольких вариантах:

* разделяемая — расположена на одном кристалле с ядрами и доступна каждому из них в полном объёме.
* индивидуальная — отдельные кэши равного объёма, интегрированные в каждое из ядер.

Многоядерные процессоры также имеют гомогенную или гетерогенную архитектуру:

* гомогенная архитектура — все ядра процессора одинаковы и выполняют одни и те же задачи
* гетерогенная архитектура — ядра процессора выполняют разные задачи.

Подавляющее большинство современных процессоров имеют два и более ядра. Топовые модели могут содержать и 8, и даже 12 ядер. Мы практически получаем несколько процессоров, способных независимо решать каждый свои задачи, при этом, естественно, возрастает производительность. Однако прирост производительности далеко не всегда оправдывает ожидания.

Во-первых, далеко не все программы поддерживают распределение вычислений на несколько ядер. Естественно, можно программы разделять между ядрами, чтобы на каждом ядре работал свой набор независимых программ. Например, на одном ядре работает операционная система с набором служебных программ, на другом пользовательские программы и так далее.

Но это дает выигрыш в производительности до тех пор, пока не появляется программа, требующая ресурсов больше, чем может дать одно ядро. Хорошо, если она поддерживает распределение нагрузки между несколькими ядрами

Во-вторых, усложняется работа с памятью, так как ядер – много, и всем им требуется доступ к ОЗУ. Требуется сложный механизм, определяющий очередность доступа ядер процессора к памяти и к другим ресурсам компьютера.

В-третьих, возрастает энергопотребление, а, следовательно, увеличивается тепловыделение и требуется мощная система охлаждения.

**Синхронная и асинхронная работа ядер**

Синхронная загрузка означает, что совершенно одинаковые операции осуществляются независимо над разными кусочками данных. Эту обработку можно осуществлять последовательно на одном ядре. Бывает выгодно дать разные кусочки всем ядрам для обработки. Все ядра, т.к. у каждого, например, свое собственное независимое АЛУ, если начали обработку разных кусочков по одинаковой операции в один момент времени, то синхронно закончат так же в один момент времени.

Если надо выполнить много разных операций, то синхронной работы не выйдет, т.к. выполняя разные операции с разными данными, ядра заканчивают в разный момент, т.к. для каждой команды потратится разное время.

Синхронный режим — это когда процессор и память работают на одной частоте шины.

Асинхронный режим — когда шины процессора и памяти находятся в соотношении 2:3, 4:5 и т.д.

В многоядерных процессорах появились два новых элемента: синхронизатор и арбитр.

*Определение 2.2.8*

**Арбитр** - специальное устройство, позволяющее обеспечить доступ с низкой латентностью к интерфейсу FSB.

*Определение 2.2.9*

**Front Side Bus** (FSB, системная шина) — шина, обеспечивающая соединение между x86/x86-64-совместимым центральным процессором и внутренними устройствами.

На каждом ядре существует специальный служебный элемент, посредством которого арбитр связан с самим ядром - синхронизатор. Арбитр работает на фиксированной частоте, кратной частоте FSB.

Асинхронный интерфейс между арбитром и каждым ядром позволяет изменять частоту работы ядра и кэша по необходимости. Разумеется, наличие арбитра обуславливает появление соответствующей латентности при общении ядра и системного интерфейса. При прочих равных условиях приоритет отдается запросам на чтение, как наиболее важным.

*Определение 2.2.10*

**Application-specific instruction-set processor** **(ASIP)** (рус. «проблемно-ориентированный процессор», «заказной микропроцессор», «заказная микросхема») — компонент (как правило, процессорное ядро), используемый в проектировании [систем на кристалле](https://ru.wikipedia.org/wiki/%D0%A1%D0%B8%D1%81%D1%82%D0%B5%D0%BC%D0%B0_%D0%BD%D0%B0_%D0%BA%D1%80%D0%B8%D1%81%D1%82%D0%B0%D0%BB%D0%BB%D0%B5).

*Определение 2.2.11*

**Система на кристалле** или **однокристальная система** ([англ.](https://ru.wikipedia.org/wiki/%D0%90%D0%BD%D0%B3%D0%BB%D0%B8%D0%B9%D1%81%D0%BA%D0%B8%D0%B9_%D1%8F%D0%B7%D1%8B%D0%BA) System-on-a-Chip) — в [микроэлектронике](https://ru.wikipedia.org/wiki/%D0%9C%D0%B8%D0%BA%D1%80%D0%BE%D1%8D%D0%BB%D0%B5%D0%BA%D1%82%D1%80%D0%BE%D0%BD%D0%B8%D0%BA%D0%B0) — электронная схема, выполняющая функции целого устройства (например, [компьютера](https://ru.wikipedia.org/wiki/%D0%9A%D0%BE%D0%BC%D0%BF%D1%8C%D1%8E%D1%82%D0%B5%D1%80)) и размещённая на одной [интегральной схеме](https://ru.wikipedia.org/wiki/%D0%98%D0%BD%D1%82%D0%B5%D0%B3%D1%80%D0%B0%D0%BB%D1%8C%D0%BD%D0%B0%D1%8F_%D1%81%D1%85%D0%B5%D0%BC%D0%B0).

[Система команд](https://ru.wikipedia.org/wiki/%D0%A1%D0%B8%D1%81%D1%82%D0%B5%D0%BC%D0%B0_%D0%BA%D0%BE%D0%BC%D0%B0%D0%BD%D0%B4) ASIP специально спроектирована для выполнения специфичных программ. Подобная специализация ядра позволяет достичь компромисса между универсальностью [процессора](https://ru.wikipedia.org/wiki/%D0%9F%D1%80%D0%BE%D1%86%D0%B5%D1%81%D1%81%D0%BE%D1%80) общего назначения (CPU) и производительностью [ASIC](https://ru.wikipedia.org/wiki/ASIC).

*Определение 2.2.12*

**«Жесткий» процессор** — процессор, в котором набор инструкций задается жёстко, каждая машинная инструкция (сложение, сдвиг, копирование) воплощена непосредственно в схеме. Чем больше инструкций, тем сложнее схема.

В процессорах, использующих микрокод, на схеме реализованы только основные команды, остальные инструкции реализованы в микропрограммах. Это позволяет не тратится на создание сложных схем, реализуя большое множество инструкций процессора. Подход, при котором используются всё более сложные микрокодные инструкции, называют CISC.

# 2.3. Представление о микросхемах обвязки процессоров

*Определение 2.3.1*

**Чипсет** - набор микросхем обычно применяемых, при создании системных плат, для "обвязки" процессора и спроектированных для совместной работы с целью выполнения набора каких-либо функций.

Так, в компьютерах чипсет, размещаемый на материнской плате, выполняет роль связующего компонента, обеспечивающего совместное функционирование подсистем памяти, центрального процессора, ввода-вывода и других. Чипсеты встречаются и в других устройствах, например, в сотовых телефонах.

*Определение 2.3.2*

**Обвязка** - вспомогательные части системы, обеспечивающие работу основного элемента, и как правило расположенные вокруг него.

Микросхемы обвязки процессоров - вспомогательные микросхемы, расположенные вокруг процессора.

В современных процессорах внутри корпуса уже расположены несколько шин, видео карта, и несколько контроллеров шин. Постепенно так получается, что обвязки для современных процессоров требуется все меньше и меньше. Но это не значит, что её нет, процессору по-прежнему нужны шины, нужны внешние устройства, но их постепенно удается уместить в корпус процессора. При этом надо понимать, что с точки зрения архитектуры, хотя все это и находится в одном корпусе, все же это отдельные части, которые просто находятся под одной крышкой.

*Определение 2.3.3*

**Аппаратный порт** — специализированный разъём в компьютере, предназначенный для подключения оборудования определённого типа.

Обычно портами называют разъёмы, предназначенные для работы периферийного оборудования, существенно разделённого от архитектуры компьютера.

**Северный и южный мосты компьютера** (а правильнее будет сказать, материнской платы) — это два основных функциональных контроллера, которые отвечают за работу всех компонентов системной платы и называются **чипсетом** (англ. chipset).

*Определение 2.3.4*

**Северный мост** (от англ. Northbridge) — это системный контроллер, являющийся одним из элементов чипсета материнской платы, отвечающий за работу с оперативной памятью (RAM), видеоадаптером и процессором (CPU).

Северный мост отвечает за частоту системной шины, тип оперативной памяти и ее максимально возможный объем. Одной из основных функций северного моста является обеспечение взаимодействия системной платы и процессора, а также определение скорости работы. Частью северного моста во многих современных материнских платах является встроенный видеоадаптер. Таким образом, функциональная особенность северного моста являет собой еще и управление шиной видеоадаптера и ее быстродействием. Также северный мост обеспечивает связь всех вышеперечисленных устройств с южным мостом.

*Определение 2.3.5*

**Южный мост** (Southbridge) — это функциональный контроллер, известен как контроллер ввода-вывода или ICH (In/Out Controller Hub). Отвечает за так называемые "медленные" операции, к которым относится отработка взаимодействия между интерфейсами IDE, SATA, USB, LAN, Embeded Audio и северным мостом системы, который, в свою очередь, напрямую связан с процессором и другими важными компонентами, такими как оперативная память или видеоподсистема. Также южный мост отвечает за обработку данных на шинах PCI, PCIe.

Таймеры - все современные таймеры спроектированы на базе первой микросхемы h. Но при всем при этом в функциональности и расположении выводов никаких различий у всех этих версий нет.

*Определение 2.3.6*

NE555 — аналоговая интегральная схема (входные и выходные сигналы являются аналоговыми сигналами), универсальный таймер — устройство для формирования (генерации) одиночных и повторяющихся импульсов со стабильными временными характеристиками.

Впервые выпущен в 1971 году компанией Signetics под обозначением NE555. Функциональные аналоги оригинального NE555 выпускаются во множестве биполярных и КМОП-вариантов. Сдвоенная версия 555 выпускается под обозначением 556, счетверенная — под обозначением 558.

*Определение 2.3.7*

**Контроллер** (англ. controller — регулятор) — это электронное устройство, предназначенное для подключения к магистрали компьютера разных по принципу действия, интерфейсу и конструктивному исполнению периферийных устройств. Т.е. это микросхема, которая управляет каким-либо подключенным к процессору устройством (например, жестким диском). В задачи контроллера входит получение от программного обеспечения различных команд и передача их устройству.

*Определение 2.3.8*

**Контроллер прерываний** (англ. Programmable Interrupt Controller, PIC) — микросхема или встроенный блок процессора, отвечающий за возможность последовательной обработки запросов на прерывание от разных устройств.

*Определение 2.3.9*

**Цифро-аналоговый преобразователь (ЦАП)** — устройство для преобразования цифрового (обычно двоичного) кода в аналоговый сигнал (ток, напряжение или заряд). Цифро-аналоговые преобразователи являются интерфейсом между дискретным цифровым миром и аналоговыми сигналами.

*Определение 2.3.10*

**Аналого-цифровой преобразователь (АЦП)** — устройство, преобразующее входной [аналоговый сигнал](https://ru.wikipedia.org/wiki/%D0%90%D0%BD%D0%B0%D0%BB%D0%BE%D0%B3%D0%BE%D0%B2%D1%8B%D0%B9_%D1%81%D0%B8%D0%B3%D0%BD%D0%B0%D0%BB) в дискретный код.

# 2.4. Представление о конвейерном выполнении команд в процессорах

Выполнение каждой команды складывается из ряда последовательных этапов (шагов, стадий), суть которых не меняется от команды к команде. С целью увеличения быстродействия процессора и максимального использования всех его возможностей в современных микропроцессорах 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, а новая поступает в него.

На рисунке 2.4.1 изображен конвейер из 5 блоков, которые называются ступенями. Первая ступень (блок С1) вызывает команду из памяти и помещает ее в буфер, где она хранится до тех пор, пока не потребуется. Вторая ступень (блок С2) декодирует эту команду, определяя ее тип и тип ее операндов. Третья ступень (блок С3) определяет местонахождение операндов и вызывает их из регистров или из памяти. Четвертая ступень (блок С4) выполняет команду, обычно проводя операнды через тракт данных. И наконец, блок С5 записывает результат обратно в нужный регистр.

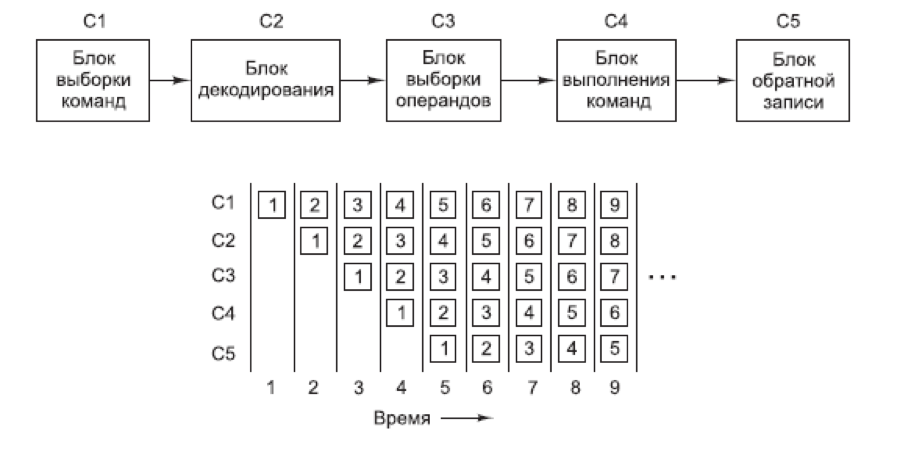


Рис. 2.4.1 Пятиступенчатый конвейер. Состояние каждой ступени в зависимости от кол-ва пройденных циклов.

1 цикл: блок С1 обрабатывает команду 1, вызывая ее из памяти.

2 цикл: блок С2 декодирует команду 1, в то время как блок С1 вызывает из памяти команду 2.

3 цикл: блок С3 вызывает операнды для команды 1, блок С2 декодирует команду 2, а блок С1 вызывает команду 3.

4 цикл: блок С4 выполняет команду 1, С3 вызывает операнды для команды 2, С2 декодирует

команду 3, а С1 вызывает команду 4.

5 цикл: блок С5 записывает результат выполнения команды 1 обратно в регистр, тогда как другие ступени конвейера обрабатывают следующие команды.

**Наборы команд и взаимодействие с операндами с позиций длины и операндов**

*Определение 2.4.1*

**RISC** (англ. restricted (reduced) instruction set computer — «компьютер с сокращённым набором команд») — архитектура процессора, в котором быстродействие увеличивается за счёт упрощения команд, чтобы их декодирование было более простым, а время выполнения — меньшим.

Недостатки RISC прямо связаны с некоторыми преимуществами этой архитектуры. Принципиальный недостаток - сокращенное число команд: на выполнение ряда функций приходится тратить несколько команд вместо одной в CISC. Это удлиняет код программы, увеличивает загрузку памяти и трафик команд между памятью и ЦП. Исследования показали, что RISC-программа в среднем на 30% длиннее CISC-программы, реализующей те же функции.

Хотя большое число регистров дает существенные преимущества, само по себе оно усложняет схему декодирования номера регистра, тем самым увеличивается время доступа к регистрам.

Устройство управления с аппаратной логикой, реализованное в большинстве RISC-систем, менее гибко, более склонно к ошибкам, затрудняет поиск и исправление ошибок, уступает при выполнении сложных команд.

**Процессор с расширенным набором команд**

*Определение 2.4.2*

**VLIW** (англ. very long instruction word — «очень длинная машинная команда») – архитектура процессоров, характеризующаяся возможностью объединения нескольких простых команд в так называемую связку. Входящие в нее команды должны быть независимы друг от друга и выполняться параллельно. Таким образом, из нескольких независимых машинных команд транслятор формирует одно «очень длинное командное слово».

Идея VLIW базируется на том, что задача эффективного планирования параллельного выполнения команд возлагается на «разумный» компилятор. Такой компилятор вначале анализирует исходную программу. Цель анализа: обнаружить все команды, которые могут быть выполнены одновременно, причем так, чтобы между командами не возникали конфликты. В ходе анализа компилятор может даже частично имитировать выполнение рассматриваемой программы. На следующем этапе компилятор пытается объединить такие команды в пакеты (связки), каждый из которых рассматривается как одна сверхдлинная команда. Объединение нескольких простых команд в одну сверхдлинную производится по следующим правилам:

* количество простых команд, объединяемых в одну команду сверхбольшой длины, равно числу имеющихся в процессоре функциональных (исполнительных) блоков (ФБ);
* 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 256 до 1024 битов. Такая метакоманда содержит несколько полей (по числу образующих ее простых команд), каждое из которых описывает операцию для конкретного функционального блока.

В качестве простых команд, образующих сверхдлинную, обычно используются команды RISC-типа.

То, что в выполняемой сверхдлинной команде исключена возможность конфликтов, позволяет предельно упростить аппаратуру VLIW-процессора и, как следствие, добиться более высокого быстродействия. Подавляющее большинство цифровых сигнальных процессоров и мультимедийных процессоров с производительностью более 1 млрд операций/с базируется на VLIW-архитектуре.

Какие же есть недостатки у VLIW. Код для VLIW обладает невысокой плотностью. Из-за большого количества пустых инструкций для простаивающих устройств программы для VLIW-процессоров могут быть гораздо длиннее, чем аналогичные программы для традиционных архитектур.

Из-за сложных внутренних зависимостей кода, программирование на уровне машинных кодов для VLIW-архитектур практически невозможно вручную. Приходится полагаться на оптимизацию компилятора, который сам может содержать ошибки.

**Проблемами VLIW-архитектуры являются**:

1) усложнение регистрового файла и, прежде всего, связей этого файла с вычислительными устройствами;

2) трудности создания компиляторов, способных найти в программе независимые команды, связать такие команды в длинные строки и обеспечить их параллельное выполнение.

**Наборы команд и взаимодействие с операндами с позиций количества независимых потоков данных**

*Определение 2.4.3*

**SISD** (Single Instruction stream over a Single Data stream) - вычислительная система с одиночным потоком команд и одиночным потоком данных.

**Архитектура SISD** — это традиционный компьютер фон-Неймановской архитектуры с одним процессором, который выполняет последовательно одну инструкцию за другой, работая с одним потоком данных. В данном классе не используется параллелизм ни данных, ни инструкций, и, следовательно, SISD-машина не является параллельной. К этому классу также принято относить конвейерные, суперскалярные и VLIW-процессоры.

*Определение 2.4.4*

**SIMD** (Single Instruction Stream & Multiple Data Stream) или ОКМД (Одиночный поток Команд и Множественный поток Данных) - архитектура, в которой есть возможность выполнять одну арифметическую операцию сразу над многими данными - элементами вектора.

SIMD-компьютеры состоят из одного командного процессора (управляющего модуля), называемого контроллером, и нескольких модулей обработки данных, называемых процессорными элементами. Управляющий модуль принимает, анализирует и выполняет команды. Если в команде встречаются данные, контроллер рассылает на все процессорные элементы команду, и эта команда выполняется на нескольких или на всех процессорных элементах. Каждый процессорный элемент имеет свою собственную память для хранения данных. Одним из преимуществ данной архитектуры считается то, что в этом случае более эффективно реализована логика вычислений. До половины логических инструкций обычного процессора связано с управлением выполнением машинных команд, а остальная их часть относится к работе с внутренней памятью процессора и выполнению арифметических операций.

*Определение 2.4.5*

**EPIC** (Explicitly Parallel Instruction Computing) - архитектура процессора с явным параллелизмом команд. EPIC позволяет микропроцессору выполнять инструкции параллельно, опираясь на работу компилятора, а не выявляя возможность параллельной работы инструкций при помощи специальных схем. Это могло упростить масштабирование вычислительной мощности процессора без увеличения тактовой частоты.

В архитектуре EPIC реализован новый подход, являющийся усовершенствованным вариантом технологии VLIW.

**Особенностями архитектуры** EPIC являются:

* большое количество регистров;
* масштабируемость архитектуры до большого количества функциональных блоков;
* явный параллелизм в машинном коде. Поиск зависимостей между командами осуществляет не процессор, а компилятор;
* предикация - команды из разных ветвей условного предложения снабжаются полями предикатов (полями условий) и запускаются параллельно;
* предварительная загрузка - данные из медленной основной памяти загружаются заранее.

# 2.5. Наборы команд и взаимодействие с операндами с позиций набора команд

*Определение 2.5.1*

**Архитектура набора команд** — часть архитектуры компьютера, определяющая программируемую часть ядра микропроцессора. На этом уровне определяются реализованные в микропроцессоре конкретного типа: архитектура памяти, взаимодействие с внешними устройствами ввода/вывода, режимы адресации, регистры, машинные команды, различные типы внутренних данных (например, с плавающей запятой, целочисленные типы и т.д.), обработчики прерываний и исключительных состояний.

*Определение 2.5.2*

**Микроархитектура** описывает модель, топологию и реализацию архитектуры набора команд на микросхеме микропроцессора. На этом уровне определяется:

* конструкция и взаимосвязь основных блоков ЦП,
* структура ядер, исполнительных устройств, АЛУ, а также их взаимодействия,
* блоков предсказания переходов,
* организация конвейеров,
* организация кэш-памяти,
* взаимодействие с внешними устройствами.

В рамках одного семейства микропроцессоров, микроархитектура со временем расширяется путем добавления новых усовершенствований и оптимизации существующих команд с целью повышения производительности, энергосбережения и функциональных возможностей микропроцессора.

*Определение 2.5.3*

**Процессор со стандартным набором команд** - процессор, в котором нельзя перегружать команды, они изначально заданы (“зашиты в процессор”).

Базовыми командами являются, как правило, следующие:

* арифметические, например, «сложения» и «вычитания»;
* битовые, например, «логическое и», «логическое или» и «логическое не»;
* присваивание данных, например, «переместить», «загрузить», «выгрузить»;
* ввода-вывода, для обмена данными с внешними устройствами;
* управляющие инструкции, например, безусловный, условный или косвенный переход, вызов подпрограммы, возврат из подпрограммы.

*Определение 2.5.4*

**Заказной микропроцессор** (Application-specific instruction-set processor, ASIP) — компонент (как правило, процессорное ядро), используемый в проектировании систем на кристалле. Необходим для криптографии или для работы с кодами, которые восстанавливают ошибки. Надо работать с элементами кода непосредственно. Создают сопроцессор. Ему “дают” его собственную команду, с которой он работает. [Сист](http://www.quickiwiki.com/ru/%D0%A1%D0%B8%D1%81%D1%82%D0%B5%D0%BC%D0%B0_%D0%BA%D0%BE%D0%BC%D0%B0%D0%BD%D0%B4)ема команд ASIP специально спроектирована для выполнения специфичных программ. Подобная специализация ядра позволяет достичь компромисса между универсальностью процессора общего назначения (CPU) и производительностью ASIC.

Некоторые ASIP обладают конфигурируемым набором инструкций (системой команд). Обычно такие ядра разделены на две части: статическую (static) логику, определяющую минимальную архитектуру системы команд, и конфигурируемую (configurable) логику, которую можно использовать для создания новых инструкций.

*Определение 2.5.5*

**Программируемая пользователем вентильная матрица** (ППВМ, англ. *Field-Programmable Gate Array*, FPGA)-полупроводниковое устройство, которое может быть сконфигурировано производителем или разработчиком после изготовления).

*Определение 2.5.6*

**Процессор с перегружаемым набором команд**. Когда необходимо, чтобы процессор имел возможность произвольного расширения, для изготовления разнообразных команд, к процессору подключается специальный ППВМ (FPGA). Матрица логических элементов, на которой можно сделать любую комбинационную логику. То есть на любую А она отвечает В. В зависит только от А, то есть работает единомоментно *(не как конечный автомат*). Можно сделать любой конечный автомат. Работает быстрее процессора.

FPGA могут быть модифицированы практически в любой момент в процессе их использования. Они состоят из конфигурируемых логических блоков, подобных переключателям с множеством входов и одним выходом (логические вентили или gates). В цифровых схемах такие переключатели реализуют базовые двоичные операции AND, NAND, OR, NOR и XOR. В большинстве современных микропроцессоров функции логических блоков фиксированы и не могут модифицироваться. Принципиальное отличие FPGA состоит в том, что и функции блоков, и конфигурация соединений между ними могут меняться с помощью специальных сигналов, посылаемых схеме.

*Определение 2.5.7*

**Микрокод** — программа, реализующая набор инструкций процессора. Так же, как одна инструкция языка высокого уровня преобразуется в серию машинных инструкций, в процессоре, использующем микрокод, каждая машинная инструкция реализуется в виде серии микроинструкций — микропрограммы, микрокода.

На большинстве компьютеров, использующих микрокод, он присутствует не в основной памяти, а в специальной быстродействующей памяти (англ. control store). Эта память может допускать только чтение либо чтение-запись; в последнем случае микрокод может быть загружен из постоянной памяти в процессе запуска процессора.

Возможность изменения микрокода позволяет исправлять найденные ошибки и добавлять реализацию новых инструкций. Микрокод также позволяет настроить микроархитектуру компьютера на эмуляцию другой (как правило, более сложной) архитектуры.

# 2.6. Классификация и особенности шин в отношении способа передачи, метода синхронизации, топологии, способа управления, адресации устройств

*Определение 2.6.1*

**Цифровая последовательная передача** — это последовательная отправка [битов](https://ru.wikipedia.org/wiki/%D0%91%D0%B8%D1%82) по одному проводу, частоте или оптическому пути. Так как это требует меньшей [обработки сигнала](https://ru.wikipedia.org/wiki/%D0%9E%D0%B1%D1%80%D0%B0%D0%B1%D0%BE%D1%82%D0%BA%D0%B0_%D1%81%D0%B8%D0%B3%D0%BD%D0%B0%D0%BB%D0%BE%D0%B2) и меньше вероятность ошибки, чем при параллельной передаче, то скорость передачи данных по каждому отдельному пути может быть быстрее. Этот механизм может использоваться на более дальних расстояниях, потому что легко может быть передана контрольная цифра или [бит чётности](https://ru.wikipedia.org/wiki/%D0%91%D0%B8%D1%82_%D1%87%D1%91%D1%82%D0%BD%D0%BE%D1%81%D1%82%D0%B8).

*Определение 2.6.2*

**Параллельной передачей** называется одновременная передача соответствующих элементов сигнала по двум или большему числу путей. Используя множество электрических проводов можно передавать несколько бит одновременно, что позволяет достичь более высоких скоростей передачи, чем при последовательной передаче. Этот метод применяется внутри компьютера, например, во внутренних [шинах данных](https://ru.wikipedia.org/wiki/%D0%A8%D0%B8%D0%BD%D0%B0_%D0%B4%D0%B0%D0%BD%D0%BD%D1%8B%D1%85), а иногда и во внешних устройствах, таких, как [принтеры](https://ru.wikipedia.org/wiki/%D0%9F%D1%80%D0%B8%D0%BD%D1%82%D0%B5%D1%80). Основной проблемой при этом является «перекос», потому что провода при параллельной передаче имеют немного разные свойства (не специально), поэтому некоторые биты могут прибыть раньше других, что может повредить сообщение. [Бит чётности](https://ru.wikipedia.org/wiki/%D0%91%D0%B8%D1%82_%D1%87%D1%91%D1%82%D0%BD%D0%BE%D1%81%D1%82%D0%B8) может способствовать сокращению ошибок. Тем не менее электрический провод при параллельной передаче данных менее надёжен на больших расстояниях, поскольку передача нарушается с гораздо более высокой вероятностью.

Основным отличием параллельных шин от последовательных является сам способ передачи данных. **Параллельные шины** можно рассматривать как совокупность сигнальных линий (можно сказать что просто проводников), объединённых по их назначению (данные, адреса, управление), которые имеют определённые электрические характеристики и протоколы передачи информации. Группы этих сигнальных линий также называются шинами:

* Линии для обмена данными (шина данных);
* Линии для адресации данных (шина адреса);
* Линии для управления данными (шина управления);

В **последовательных** шинах используется одна сигнальная линия (возможно использование двух отдельных каналов для разделения потоков приёма-передачи). Соответственно, информационные биты здесь передаются последовательно. Данные для передачи через последовательную шину облекаются в пакеты (пакет – единица информации, передаваемая как целое между двумя устройствами), в которые, помимо собственно полезных данных, включается некоторое количество служебной информации.

**Комбинированная шина:**

Бывает, иногда выгодно огромный кусок данных разбить так, чтобы какие-то его части передавалась параллельно, а какие-то - последовательно.

Например, нужно передать 4 элемента данных.

Какие могут быть варианты передачи:

1. Передать за 1 такт времени по параллельной шине.
2. Передать за 4 такта времени по последовательной шине.
3. Комбинированно: элементы данных объединяют по два и используют 2 такта для передачи.

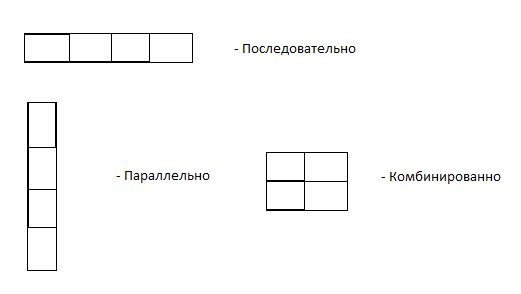


Рис. 2.6.1

**Классификация и особенности шин в отношении метода синхронизации:**

(Синхронизируются 2 объекта: передатчик и приёмник. Синхронизация - передача данных)

*Определение 2.6.3*

Для последовательной передачи данных достаточно одной линии, по которой могут последовательно передаваться биты данных. Приемник должен уметь распознавать, где начинается и где заканчивается сигнал, который отвечает каждому биту данных. Иначе говоря, передатчик и приемник должны уметь синхронизироваться. Если качество синхронизации низкое (за время передачи одного бита несогласованность достигает нескольких процентов), используется **асинхронный** режим передачи данных: выполняется согласование синхрогенераторов (генерирует импульс, называемый импульсом синхронизации) в начале передачи каждого байта.

Как правило, передача байта начинается из специального старт-бита, потом идут биты данные, а за ними, возможно, бит четности (Рис. 2.6.2). После всех битов данных передается стоп-бит. Старт-бит и стоп-бит всегда имеют определенное значение: старт-бит кодируется логическим нулем, а стоп-бит - логической единицей. Между передачей стоп-бита одного байта и старт-бита следующего байта может проходить произвольное время.

Асинхронный режим сильно зависит от погрешностей синхрогенераторов, что задает моменты приема битов. Чем выше скорость передачи, тем более эта погрешность. В результате этих и некоторых других ограничений скорость передачи в асинхронном режиме ограничена сотнями килобит в секунду (стандартные скорости: 50, 75, 110, 150, 300, 600, 1200, 2400, 4800, 9600, 19200, 38400, 57600, 115200 бит/с).

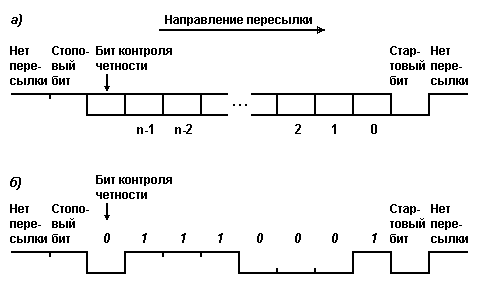


Рис. 2.6.2

*Определение 2.6.4*

Если синхронизация очень качественна (например, используется дополнительная линия, по которой передается синхросигнал), то можно передавать поток данных без дополнительной синхронизации отдельных байтов. Такой режим называется **синхронным**. Передача битов данных опережается и заканчивается выдачей в канал символа синхронизации. При отсутствии данных передатчик должен постоянно передавать в канал символы синхронизации.

Синхронная последовательная передача начинается с пересылки в приемник одного или двух символов синхронизации (не путать с импульсами синхронизации). Получив такой символ (символы), приемник начинает прием данных и их преобразование в параллельный формат. Естественно, что при такой организации синхронной последовательной передачи она целесообразна лишь для пересылки массивов слов, а не отдельных символов. Это обстоятельство, а также необходимость использования для обмена сравнительно дорогих (четырехпроводных или кабельных) линий связи помешало широкому распространению синхронной последовательности передачи данных.

*Определение 2.6.5*

**Плезиосинхронная** передача требует внутренней синхронизации узлов от источников с номинально совпадающими частотами. Срок “плезиохронна” значит “почти синхронная”, поскольку частоты источников точно не совпадают, и впоследствии накапливается разногласие, которое компенсируется вставкой фиктивных данных.

*Определение 2.6.6*

**Пакетный** режим передачи данных позволяет ускорить процессы чтения и записи данных в память. Суть заключается в том, что при необходимости чтения из памяти (или записи в память) одного машинного слова, процессор считывает вместе с ним ещё несколько подряд расположенных слов. Длина каждого машинного слова, т.е. размер отдельного блока данных, передаваемого между микропроцессором и памятью, равна разрядности внешней шины данных микросхемы памяти. Ширина шины данных самых первых микросхем памяти составляла всего 1 бит, в настоящее время чаще встречаются 4-, 8- и 16- (реже - 32-) битные микросхемы памяти.

*Определение 2.6.7*

**DRAM** ([англ.](https://ru.wikipedia.org/wiki/%D0%90%D0%BD%D0%B3%D0%BB%D0%B8%D0%B9%D1%81%D0%BA%D0%B8%D0%B9_%D1%8F%D0%B7%D1%8B%D0%BA) dynamic random access memory — динамическая память с произвольным доступом) — тип [компьютерной памяти](https://ru.wikipedia.org/wiki/%D0%9A%D0%BE%D0%BC%D0%BF%D1%8C%D1%8E%D1%82%D0%B5%D1%80%D0%BD%D0%B0%D1%8F_%D0%BF%D0%B0%D0%BC%D1%8F%D1%82%D1%8C), отличающийся использованием [полупроводниковых материалов](https://ru.wikipedia.org/wiki/%D0%9F%D0%BE%D0%BB%D1%83%D0%BF%D1%80%D0%BE%D0%B2%D0%BE%D0%B4%D0%BD%D0%B8%D0%BA%D0%BE%D0%B2%D1%8B%D0%B5_%D0%BC%D0%B0%D1%82%D0%B5%D1%80%D0%B8%D0%B0%D0%BB%D1%8B), [энергозависимостью](https://ru.wikipedia.org/wiki/%D0%AD%D0%BD%D0%B5%D1%80%D0%B3%D0%BE%D0%B7%D0%B0%D0%B2%D0%B8%D1%81%D0%B8%D0%BC%D0%B0%D1%8F_%D0%BF%D0%B0%D0%BC%D1%8F%D1%82%D1%8C) и возможностью доступа к данным, хранящимся в произвольных [ячейках памяти](https://ru.wikipedia.org/wiki/%D0%AF%D1%87%D0%B5%D0%B9%D0%BA%D0%B0_%D0%BF%D0%B0%D0%BC%D1%8F%D1%82%D0%B8) (см. [запоминающее устройство с произвольным доступом](https://ru.wikipedia.org/wiki/%D0%97%D0%B0%D0%BF%D0%BE%D0%BC%D0%B8%D0%BD%D0%B0%D1%8E%D1%89%D0%B5%D0%B5_%D1%83%D1%81%D1%82%D1%80%D0%BE%D0%B9%D1%81%D1%82%D0%B2%D0%BE_%D1%81_%D0%BF%D1%80%D0%BE%D0%B8%D0%B7%D0%B2%D0%BE%D0%BB%D1%8C%D0%BD%D1%8B%D0%BC_%D0%B4%D0%BE%D1%81%D1%82%D1%83%D0%BF%D0%BE%D0%BC)). [Модули памяти](https://ru.wikipedia.org/wiki/%D0%9C%D0%BE%D0%B4%D1%83%D0%BB%D1%8C_%D0%BF%D0%B0%D0%BC%D1%8F%D1%82%D0%B8) с памятью такого типа широко используются в современных [компьютерах](https://ru.wikipedia.org/wiki/%D0%9A%D0%BE%D0%BC%D0%BF%D1%8C%D1%8E%D1%82%D0%B5%D1%80) в качестве [оперативных запоминающих устройств](https://ru.wikipedia.org/wiki/%D0%9E%D0%BF%D0%B5%D1%80%D0%B0%D1%82%D0%B8%D0%B2%D0%BD%D0%B0%D1%8F_%D0%BF%D0%B0%D0%BC%D1%8F%D1%82%D1%8C) (ОЗУ), также используются в качестве [устройств постоянного хранения информации](https://ru.wikipedia.org/wiki/%D0%AD%D0%BB%D0%B5%D0%BA%D1%82%D1%80%D0%BE%D0%BD%D0%BD%D1%8B%D0%B9_%D0%B4%D0%B8%D1%81%D0%BA) в системах, требовательных к задержкам.

При пакетной передаче данных нет необходимости указывать полный адрес (номер строки и столбца) каждой ячейки памяти. Вместо этого подаётся только адрес начальной ячейки пакета, а следующие ячейки считываются подряд из текущей строки столько раз, сколько слов в пакете. Преимущество такой схемы в экономии времени на передачу адреса: для чтения нескольких слов данных требуется указать всего один адрес.

В большинстве архитектур SDRAM подсистема памяти высылает критическое слово в первую очередь, а следом идут остальные данные. Именно так и работает память SDRAM в пакетном режиме (burst mode). Память высылает на шину четыре блока по 8 байт (или один пакет). Там же присутствует задержка в три тактовых импульса между отсылкой запроса и поступлением на шину первого из четырех слов (если задержка чтения в нашей системе составляет 3 такта шины памяти). Поэтому первое критическое слово имеет задержку в 3 такта, и каждый последующий 8-байтный блок — задержку в один такт. Это соответствует задержкам DRAM пакетного режима типа 3-1-1-1. Общее время, которое требуется на передачу всех данных по шине, составляет 3+1+1+1=6 тактов шины памяти. Кстати, если бы пакетного режима не существовало, то на передачу того же количества данных ушло бы 3+3+3+3=12 тактов шины памяти.

**Классификация и особенности шин в отношении топологии**

Топология определяет способ взаимодействия компьютеров в сети. Различным видам топологий соответствуют различные методы взаимодействия, и эти методы оказывают большое влияние на сеть. Все сети строятся на основе трех базовых топологий: *шина (bus); звезда (star); кольцо (ring)*. Если компьютеры подключены вдоль одного кабеля [сегмента], топология называется шиной. В том случае, когда компьютеры подключены к сегментам кабеля, исходящим из одной точки, или концентратора, топология называется звездой. Если кабель, к которому подключены компьютеры, замкнут в кольцо, такая топология носит название кольца. Хотя сами по себе базовые топологии несложны, в реальности часто встречаются довольно сложные комбинации, объединяющие свойства нескольких топологий.

*Определение 2.6.8*

Топологию **"шина"** часто называют "линейной шиной" (linear bus). Данная топология относится к наиболее простым и широко распространенным топологиям. В ней используется один кабель, именуемый магистралью или сегментом, вдоль которого параллельно подключены все компьютеры сети. В сети с топологией "шина" компьютеры адресуют данные конкретному компьютеру, передавая их по кабелю в виде электрических сигналов. Данные в виде электрических сигналов передаются всем компьютерам сети; однако информацию принимает только тот, адрес которого соответствует адресу получателя, зашифрованному в этих сигналах. Причем в каждый момент времени только один компьютер может вести передачу. Так как данные в сеть передаются лишь одним компьютером, ее производительность зависит от количества компьютеров, подключенных к шине. Чем больше компьютеров, ожидающих передачи данных, тем медленнее сеть. (Но на быстродействие сети влияет множество факторов, в том числе: характеристики аппаратного обеспечения компьютеров в сети; частота, с которой компьютеры передают данные; тип работающих сетевых приложений; тип сетевого кабеля; расстояние между компьютерами в сети.) Шина - пассивная топология. Это значит, что компьютеры только "слушают" передаваемые по сети данные, но не перемещают их от отправителя к получателю. Поэтому, если один из компьютеров выйдет из строя, это не скажется на работе остальных. В активных топологиях компьютеры регенерируют сигналы и передают их по сети. Данные, или электрические сигналы, распространяются по всей сети - от одного конца кабеля к другому. Если не предпринимать никаких специальных действий, сигнал, достигая конца кабеля, будет отражаться и не позволит другим компьютерам осуществлять передачу. Поэтому, после того как данные достигнут адресата, электрические сигналы необходимо погасить. Чтобы предотвратить отражение электрических сигналов, на каждом конце кабеля устанавливают терминаторы (terminators), поглощающие эти сигналы. Все концы сетевого кабеля должны быть к чему-нибудь подключены, например, к компьютеру для увеличения длины кабеля.

*Определение 2.6.9*

При топологии **"звезда"** все компьютеры с помощью сегментов кабеля подключаются к центральному компоненту, именуемому концентратором (hub). Сигналы от передающего компьютера поступают через концентратор ко всем остальным. Эта топология возникла на заре вычислительной техники, когда компьютеры были подключены к центральному, главному, компьютеру. В сетях с топологией "звезда" подключение кабеля и управление конфигурацией сети централизованы. Но есть и недостаток: так как все компьютеры подключены к центральной точке, для больших сетей значительно увеличивается расход кабеля. К тому же, если центральный компонент выйдет из строя, нарушится работа всей сети. А если выйдет из строя только один компьютер (или кабель, соединяющий его с концентратором), то лишь этот компьютер не сможет передавать или принимать данные по сети. На остальные компьютеры в сети это не повлияет.

*Определение 2.6.10*

При топологии **"кольцо"** компьютеры подключаются к кабелю, замкнутому в кольцо. Поэтому у кабеля просто не может быть свободного конца, к которому надо подключать терминатор. Сигналы передаются по кольцу в одном направлении и проходят через каждый компьютер. В отличие от пассивной топологии "шина", здесь каждый компьютер выступает в роли репитера, усиливая сигналы и передавая их следующему компьютеру. Поэтому, если выйдет из строя один компьютер, прекращает функционировать вся сеть. Один из принципов передачи данных в кольцевой сети носит название передачи маркера. Суть его такова. Маркер последовательно, от одного компьютера к другому, передается до тех пор, пока его не получит тот, который "хочет" передать данные. Передающий компьютер изменяет маркер, помещает электронный адрес в данные и посылает их по кольцу. Данные проходят через каждый компьютер, пока не окажутся у того, чей адрес совпадает с адресом получателя, указанным в данных. После этого принимающий компьютер посылает передающему компьютеру сообщение, где подтверждает факт приема данных. Получив подтверждение, передающий компьютер создает новый маркер и возвращает его в сеть. На первый взгляд кажется, что передача маркера отнимает много времени, однако на самом деле маркер передвигается практически со скоростью света. В кольце диаметром 200м маркер может циркулировать с частотой 10 тысяч оборотов в секунду. Также “кольцо” может состоять из двух кабелей (сигналы передаются по часовой стрелке и против. Это гарантирует, что передача данных не остановится при обрыве одного из кабелей (или иной проблеме с одним из кабелей).

*Определение 2.6.11*

**Комбинированные топологии *-*** топологии, которые комбинируют компоновку сети по принципу шины, звезды и кольца. *Звезда-шина (star-bus)* — это комбинация топологий "шина" и "звезда". Чаще всего это выглядит так: несколько сетей с топологией "звезда" объединяются при помощи магистральной линейной шины. В этом случае выход из строя одного компьютера не оказывает никакого влияния на сеть - остальные компьютеры по-прежнему взаимодействуют друг с другом. А выход из строя концентратора повлечет за собой остановку подключенных к нему компьютеров и концентраторов.

*Определение 2.6.12*

**Звезда-кольцо** *(star-ring)* кажется несколько похожей на звезду-шину. И в той, и в другой топологии компьютеры подключены к концентратору, который фактически и формирует кольцо или шину. Отличие в том, что концентраторы в звезде-шине соединены магистральной линейной шиной, а в звезде-кольце на основе главного концентратора они образуют звезду.

Таблица 2.6.1

|  |  |  |
| --- | --- | --- |
| Топология | Преимущества | Недостатки |
| Шина | Экономный расход кабеля. Сравнительно недорогая и сложная в использовании среда передачи. Простота, надежность. Легко расширяется | При значительных объемах графика уменьшается пропускная способность среды. Трудно локализовать проблемы. Выход из строя кабеля останавливает работу многих пользователей |
| Кольцо | Все компьютеры имеют равный доступ. Количество пользователей не оказывает сколько-нибудь значительного влияния на производительность | Выход из строя одного компьютера может вывести из строя всю сеть. Трудно локализовать проблемы. Изменение конфигурации сети требует остановки работы всей сети |
| Звезда | Легко модифицировать сеть, добавляя новые компьютеры. Централизованный контроль и управление. Выход из строя одного компьютера не влияет на работоспособность сети. | Выход из строя центрального узла выводит из строя всю сеть. |

**Классификация и особенности шин в отношении адресации устройств (индивидуальная, групповая, широковещательная).**

Разделение шин в отношении адресации устройств напрямую зависит от назначения шины. Индивидуальная адресация применяется для простой связи двух устройств типа “точка-точка”. Однако гораздо чаще требуется связать большое число устройств (устройства соединены шиной и у каждого устройства есть адрес), для этого и нужны групповая и широковещательная адресации.

Шировещательная адресация, подразумевает, что есть сетевая станция, которая одновременно общается со всеми устройствами, входящими в ту же самую область (домен) широковещания или подсеть. В таком случае все устройства в этой области или подсети могут “слушать” сетевую станцию и получать одни и те же данные, которые сетевая станция передает только один раз для всех устройств. Но и у такого режима есть недостатки. Например, с помощью видеосервера можно в реальном времени осуществлять трансляцию телепередач, скажем новостей CNN, в сеть масштаба университетского городка, и при этом они будут доступны любому пользователю. Однако в этом случае широковещательный трафик должен пересекать границы подсетей и под него должна быть отведена значительная доля драгоценной пропускной способности. Кроме того, для поддержки широковещания требуется, чтобы все машины и межсетевое оборудование сети (маршрутизаторы и коммутаторы) обрабатывали пакеты широковещательного трафика, даже если получение широковещательных сообщений требуется только для небольшой части хостов.

Адекватное решение проблемы предлагает групповая адресация. При таком методе связи информация с одной станции может передаваться на несколько станций-приемников одновременно, но в отличие от одноадресного и широковещательного режимов, компьютер-передатчик выбирает определенную группу машин для получения его информации. Это становится возможным благодаря групповому адресу, который можно представить себе в виде отдельного телевизионного канала. Машины группы просто "настраиваются" на конкретный групповой адрес для приема через него потока представляющих интерес данных. При групповой адресации потоки информации общего доступа передаются по сети только один раз и исключительно тем пользователям, которые хотят их получать. Если, допустим, имеется 40 подсетей, а трансляция CNN принимается только на машины двух из них, то полоса пропускания 38 оставшихся подсетей не используется.

# Глава 3. Процессор (программирование)

# 3.1. Понятие о параллельных и распределённых вычислениях

*Определение 3.1.1*

**Параллельные вычисления** — способ организации компьютерных вычислений, при котором программы разрабатываются как набор взаимодействующих вычислительных процессов, работающих параллельно (одновременно).

*Определение 3.1.2*

**Распределённые вычисления** — способ решения трудоёмких вычислительных задач с использованием нескольких компьютеров, чаще всего объединённых в параллельную вычислительную систему.

*Определение 3.1.3*

**Распределенная система** — [система](https://ru.wikipedia.org/wiki/%D0%A1%D0%B8%D1%81%D1%82%D0%B5%D0%BC%D0%B0), для которой отношения [местоположений](https://ru.wikipedia.org/wiki/%D0%9C%D0%B5%D1%82%D1%80%D0%B8%D1%87%D0%B5%D1%81%D0%BA%D0%BE%D0%B5_%D0%BF%D1%80%D0%BE%D1%81%D1%82%D1%80%D0%B0%D0%BD%D1%81%D1%82%D0%B2%D0%BE) элементов (или групп элементов) играют существенную роль с точки зрения функционирования системы, а, следовательно, и с точки зрения анализа и синтеза системы.

Для распределённых систем характерно распределение функций, ресурсов между множеством элементов (узлов) и отсутствие единого управляющего центра, поэтому выход из строя одного из узлов не приводит к полной остановке всей системы. Типичной распределённой системой является [Интернет](https://ru.wikipedia.org/wiki/%D0%98%D0%BD%D1%82%D0%B5%D1%80%D0%BD%D0%B5%D1%82).

Аппаратные и программные средства, реализующие распределённую модель вычислений.

*Определение 3.1.4*

**Программным средством**, реализующим модель распределенных вычислений, является распределенная операционная система.

Распределенная ОС, динамически и автоматически распределяя работы по различным машинам системы для обработки, заставляет набор сетевых машин работать как виртуальный [унипроцессор](https://en.wikipedia.org/wiki/Uniprocessor_system). Пользователь распределенной ОС, вообще говоря, не имеет сведений о том, на какой машине выполняется его работа. Появление сетей, предназначенных для взаимной связи различных компьютеров, привело к разработке средств, а затем и операционных систем, позволяющих осуществлять управление, так называемой, мультимашинной архитектурой, то есть совокупности полносоставных компьютеров (процессоры, память, вводы-выводы и т.д..), связанных в сеть.

*Определение 3.1.5*

**Аппаратным средством** модели распределенных вычислений может служить **Грид-вычисления** (англ. grid — решётка, сеть) — это форма распределённых вычислений, в которой «виртуальный суперкомпьютер» представлен в виде кластеров, соединённых с помощью сети, слабосвязанных гетерогенных компьютеров, работающих вместе для выполнения огромного количества заданий (операций, работ). Эта технология применяется для решения научных, математических задач, требующих значительных вычислительных ресурсов. Грид-вычисления используются также в коммерческой инфраструктуре для решения таких трудоёмких задач, как экономическое прогнозирование, сейсмоанализ, разработка и изучение свойств новых лекарств.

Грид с точки зрения сетевой организации представляет собой согласованную, открытую и стандартизованную среду, которая обеспечивает гибкое, безопасное, скоординированное разделение вычислительных ресурсов и ресурсов хранения информации, которые являются частью этой среды, в рамках одной виртуальной организации.

**Многопортовая память, зеркалирование памяти, шины с поддержкой режимов групповой адресации и широковещания**

*Определение 3.1.6*

**Многопортовая память** - это статическое ОЗУ (статические запоминающие устройства) с двумя или более независимыми интерфейсами, обеспечивающими доступ к пространству памяти через разделенные шины адреса, данных и управления. Структура двухпортового статического ОЗУ (Рис. 3.1.1) содержит единый массив памяти (COMMON CENTRAL MEMORY) и два независимых порта (PORT\_L и PORT\_R) для обращения к этому массиву.

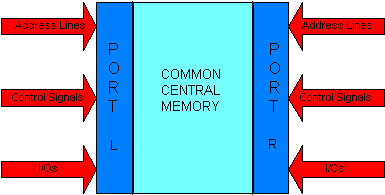


Рис. 3.1.1 Структура двухпортового статического ОЗУ

*Определение 3.1.7*

**Зеркалированние памяти** — это процесс разделения памяти на 2 канала, где первый канал отражается на второй, на втором канале создается избыточная копия памяти. Данный метод позволяет сделать ввод/вывод регистров и памяти в нескольких адресных пространствах, так как один и тот же байт доступен по нескольким адресам.

*Определение 3.1.8*

**Broadcast-шина**— термин для описания способа обмена данными, где некоторые объем информации передается из одной точки во все другие точки. Существует лишь один отправитель, информация передается всем соединенным получателям.

*Определение 3.1.9*

**Multicast-шина**— термин для описания способа обмена данными, где некоторые объем информации передается из одной или нескольких точек в набор других точек. В данном случае может существовать один или несколько отправителей, а информация распространяется по набору получателей (которых может и не быть).

*Определение 3.1.10*

**Volatile** — это спецификатор, применяемый при объявлении переменной. Он сообщает компилятору, что значение переменной может изменяться в любой момент – без какого-либо действия со стороны кода, который компилятор обнаруживает поблизости, и которая не должна быть оптимизирована.

*Определение 3.1.11*

**Семафор** — объект, ограничивающий количество потоков, которые могут войти в заданный участок кода. Определение введено Эдсгером Дейкстрой в 1962 или 1963 году. Семафоры используются для синхронизации и защиты передачи данных через разделяемую память, а также для синхронизации работы процессов и потоков.

*Определение 3.1.12*

**Критическая секция** — участок исполняемого кода программы, в котором производится доступ к общему ресурсу (данным или устройству), который не должен быть одновременно использован более чем одним потоком исполнения. При нахождении в **критической секции** двух (или более) процессов возникает состояние «гонки» («состязания»). Для избежание данной ситуации необходимо выполнение четырех условий:

1. Два процесса не должны одновременно находиться в критических областях.
2. В программе не должно быть предположений о скорости или количестве процессоров.
3. Процесс, находящийся вне критической области, не может блокировать другие процессы.
4. Невозможна ситуация, в которой процесс вечно ждет попадания в критическую область.

*Определение 3.1.13*

**Прерывание** — сигнал от программного или аппаратного обеспечения, сообщающий процессору о наступлении какого-либо события, требующего немедленного внимания. Прерывание извещает процессор о наступлении высокоприоритетного события, требующего прерывания текущего кода, выполняемого процессором. Процессор отвечает приостановкой своей текущей активности, сохраняя свое состояние, и выполняя функцию, называемую обработчиком прерывания (или программой обработки прерывания), который реагирует на событие и обслуживает его, после чего возвращает управление в прерванный код.

В зависимости от источника возникновения сигнала прерывания делятся на:

* асинхронные, или внешние (аппаратные) — события, которые исходят от внешних аппаратных устройств (например, периферийных устройств) и могут произойти в любой произвольный момент: сигнал от таймера, сетевой карты или дискового накопителя, нажатие клавиш клавиатуры, движение мыши. Факт возникновения в системе такого прерывания трактуется как запрос на прерывание - устройства сообщают, что они требуют внимания со стороны ОС;
* синхронные, или внутренние — события в самом процессоре как результат нарушения каких-то условий при исполнении машинного кода: деление на ноль или переполнение стека, обращение к недопустимым адресам памяти или недопустимый код операции;
* программные (частный случай внутреннего прерывания) — инициируются исполнением специальной инструкции в коде программы. Программные прерывания, как правило, используются для обращения к функциям встроенного программного обеспечения, драйверов и операционной системы.

*Определение 3.1.14*

Термин «**ловушка**» иногда используется как синоним термина «прерывание» или «внутреннее прерывание». Как правило, словоупотребление устанавливается в документации производителя конкретной архитектуры процессора. В вычислении и операционных системах, **ловушка**, также известная как **исключение** или **ошибка**, как правило является типом синхронного перерыва, как правило, вызванного исключительным условием.

*Определение 3.1.15*

В информатике, **блокировка** — механизм синхронизации, позволяющий обеспечить исключительный доступ к разделяемому ресурсу между несколькими потоками.

[Блокировки](https://ru.wikipedia.org/wiki/%D0%91%D0%BB%D0%BE%D0%BA%D0%B8%D1%80%D0%BE%D0%B2%D0%BA%D0%B0_(%D0%BF%D1%80%D0%BE%D0%B3%D1%80%D0%B0%D0%BC%D0%BC%D0%B8%D1%80%D0%BE%D0%B2%D0%B0%D0%BD%D0%B8%D0%B5)) — это один из способов обеспечить политику [управления распараллеливанием](https://ru.wikipedia.org/w/index.php?title=%D0%A3%D0%BF%D1%80%D0%B0%D0%B2%D0%BB%D0%B5%D0%BD%D0%B8%D0%B5_%D1%80%D0%B0%D1%81%D0%BF%D0%B0%D1%80%D0%B0%D0%BB%D0%BB%D0%B5%D0%BB%D0%B8%D0%B2%D0%B0%D0%BD%D0%B8%D0%B5%D0%BC&action=edit&redlink=1).

# 3.2. Программно-аппаратные переходы и аспекты их применения. Жесткие процессоры и процессоры, использующие микрокод

*Определение 3.2.1*

**Внутренняя симуляция** — с помощью программы симулируется (моделируется) работа специализированного вычислителя на компьютерах общего назначения. Производится его программирование.

*Определение 3.2.2*

**Внешняя симуляция** — Тестирование работы устройства непосредственно на конкретном экземпляре, изменение его аппаратных характеристик. Такой подход позволяет проводить локальную отладку.

*Определение 3.2.3*

**Сопряжение** — соединение специализированного устройства с компьютерами общего назначения для его программной настройки, отладки (удаленной).

# 3.3. Специализированные средства разработки программного обеспечения

*Определение 3.3.1*

**Ассемблеры** — компьютерные программы, осуществляющие преобразование программы в форме исходного текста на языке ассемблера в машинные команды в виде объектного кода.

*Определение 3.3.2*

**Трансляторы** — программы или технические средства, выполняющие трансляцию программы.

*Определение 3.3.3*

**Компоновщики (редакторы связей)** — программы, которые производят компоновку — принимают на вход один или несколько объектных модулей и собирают по ним исполнимый модуль.

*Определение 3.3.4*

**Препроцессоры исходных текстов** — это компьютерные программы, принимающие данные на входе и выдающие данные, предназначенные для входа другой программы, например, такой, как компилятор

*Определение 3.3.5*

**Отладчик** является модулем среды разработки или отдельным приложением, предназначенным для поиска ошибок в программе.

*Определение 3.3.6*

**Текстовые редакторы** — компьютерные программы, предназначенные для создания и изменения текстовых файлов, а также их просмотра на экране, вывода на печать, поиска фрагментов текста и т. п.

*Определение 3.3.7*

**Специализированные редакторы исходных текстов** — текстовые редакторы для создания и редактирования исходного кода программ. Специализированный редактор исходных текстов может быть отдельным приложением, или быть встроен в интегрированную среду разработки (IDE).

*Определение 3.3.8*

**Библиотеки подпрограмм** — сборники подпрограмм или объектов, используемых для разработки программного обеспечения.

Перечисленные инструменты могут входить в состав интегрированных сред разработки (IDE — Integrated Development Environment)

*Определение 3.3.9*

**Профилирование** — сбор характеристик работы [программы](https://ru.wikipedia.org/wiki/%D0%9A%D0%BE%D0%BC%D0%BF%D1%8C%D1%8E%D1%82%D0%B5%D1%80%D0%BD%D0%B0%D1%8F_%D0%BF%D1%80%D0%BE%D0%B3%D1%80%D0%B0%D0%BC%D0%BC%D0%B0), таких как время выполнения отдельных фрагментов (обычно подпрограмм), число верно [предсказанных условных переходов](https://ru.wikipedia.org/wiki/%D0%9F%D1%80%D0%B5%D0%B4%D1%81%D0%BA%D0%B0%D0%B7%D0%B0%D1%82%D0%B5%D0%BB%D1%8C_%D0%BF%D0%B5%D1%80%D0%B5%D1%85%D0%BE%D0%B4%D0%BE%D0%B2), число [кэш](https://ru.wikipedia.org/wiki/%D0%9A%D1%8D%D1%88)-промахов и т. д. Инструмент, используемый для анализа работы, называют профилировщиком или профайлером (profiler). Обычно выполняется совместно с [оптимизацией](https://ru.wikipedia.org/wiki/%D0%9E%D0%BF%D1%82%D0%B8%D0%BC%D0%B8%D0%B7%D0%B0%D1%86%D0%B8%D1%8F_(%D0%B8%D0%BD%D1%84%D0%BE%D1%80%D0%BC%D0%B0%D1%82%D0%B8%D0%BA%D0%B0)) программы.

С помощью отладки и профилирования можно найти в своей программе самую тормозящую функцию, переписать её на C, вместо C++, а затем тем же способом найти её самую тормозящую часть и переписать её на языке ассемблера.

*Определение 3.3.10*

**Точка останова** (англ. breakpoint) — это преднамеренное прерывание выполнения программы, при котором выполняется вызов отладчика (одновременно с этим, программа сама может использовать точки останова для своих нужд). После перехода к отладчику, программист может исследовать состояние программы (логи – файлы регистрации, состояние памяти, регистров процессора, стека и т. п.), с тем чтобы определить, правильно ли ведёт себя программа. После остановки в отладчике, программа может быть завершена либо продолжена с того же места где произошла остановка.

На практике, точка останова определяется как одно или несколько условий, при которых происходит прерывание программы. Наиболее часто используется условие останова при переходе управления к указанной инструкции программы (instruction breakpoint). Другое условие останова — операция чтения, записи или изменения указанной ячейки, или диапазона ячеек памяти (data breakpoint или watchpoint).

Многие процессоры имеют аппаратную поддержку точек останова (обычно, только для instruction breakpoint и watchpoint). При отсутствии такой аппаратной поддержки отладчики используют программные точки останова.

Для реализации точек останова в Visual C++ используется специальная инструкция процессора (int 3 на процессорах Intel). Выполнение этой инструкции приводит к исключению EXCEPTION\_BREAKPOINT. Установить точку останова можно в любом месте программы. Для этого отладчик записывает по соответствующему адресу инструкцию int 3 (1 байт с кодом 0xCC). Очевидно, что это можно сделать и вручную, вставив в программу инструкцию \_\_asm int 3.

Метка указывает ассемблеру, что надо создать переменную с этим именем, содержащую адрес команды, идущей после метки. Метка в современном ассемблере мало отличается от метки в С. Название метки начинается с буквы (ни в коем случае не с цифры!). Метки используются при прыжках (jmp) или с командой call.

Пример использования метки для того, чтобы перепрыгнуть процедуру, которую мы хотим реализовать в теле другой процедуры:

|  |  |
| --- | --- |
| .code  start proc  jmp ml  myproc proc near  ret  myproc endp  ml:  ...  start endp  end start | Начало кода программы  В этой (главной) процедуре мы хотим сделать еще одну процедуру  Перепрыгиваем процедуру, которую мы делаем (безусловный переход)  Наша процедура  Она выполняет всего одну команду - ret  Конец ее описания  Это метка, сюда прыгаем  ...  Теперь мы можем вызывать нашу процедуру  Конец главного процесса  Указание на то, что программа начинается с метки start |

Первым символом в метке должна быть буква или спецсимвол. Цифра не может быть первым символом метки, а символы $ и ? иногда имеют специальные значения и обычно не рекомендуются к использованию. Большие и маленькие буквы по умолчанию не различаются, но различие можно включить, задав ту или иную опцию в командной строке ассемблера. Максимальная длина метки - 31 символ. Примеры меток: COUNT, PAGE25, $E10. Рекомендуется использовать описательные и смысловые метки. Имена регистров, например, AX, DI или AL являются зарезервированными и используются только для указания соответствующих регистров.

Если метка располагается перед командой процессора, сразу после нее всегда ставится символ «:» (двоеточие), который указывает ассемблеру, что надо создать переменную с этим именем, содержащую адрес текущей команды:

some\_loop:

lodsw ; cчитать слово из строки,

cmp ax,7 ; если это 7 - выйти из цикла

loopne some\_loop

Когда метка стоит перед директивой ассемблера, она обычно оказывается одним из операндов этой директивы и двоеточие не ставится:

codesg segment

lodsw ; cчитать слово из строки,

cmp ax,7 ; если это 7 - выйти из цикла

codesg ends

Рассмотрим директивы, работающие напрямую с метками и их значениями: LABEL, EQU и =.

*Директива LABEL*

метка label тип

Директива LABEL определяет метку и задает ее тип. Тип может быть одним из: BYTE (байт), WORD (слово), DWORD (двойное слово), FWORD (6 байт), QWORD (учетверенное слово), TBYTE (10 байт), NEAR (ближняя метка), FAR (дальняя метка). Метка получает значение, равное адресу следующей команды или следующих данных, и тип, указанный явно. В зависимости от типа команда

mov метка,0

запишет в память байт (слово, двойное слово и т.д.), заполненный нулями, а команда

call метка

выполнит ближний или дальний вызов подпрограммы.

С помощью директивы LABEL удобно организовывать доступ к одним и тем же данным, как к байтам, так и к словам, определив перед данными две метки с разными типами.

*Директива EQU*

Директива EQU присваивает метке значение, которое определяется как результат целочисленного выражения в правой части. Результатом этого выражения может быть целое число, адрес или любая строка символов:

метка equ выражение

truth equ 1

message1 equ 'Try again$'

var2 equ 4[si]

cmp ax,truth ; cmp ax,1

db message1 ; db 'Try again$'

mov ax,var2 ; mov ax, 4[si]

Директива EQU чаще всего используется с целью введения параметров, общих для всей программы, аналогично команде #define препроцессора языка С.

*Директива =*

Директива = эквивалентна EQU, но определяемая ею метка может принимать только целочисленные значения. Кроме того, метка, указанная этой директивой, может быть переопределена.

Каждый ассемблер предлагает целый набор специальных предопределенных меток — это может быть текущая дата (@date или ??date), тип процессора (@cpu) или имя того или иного сегмента программы, но единственная предопределенная метка, поддерживаемая всеми рассматриваемыми нами ассемблерами, — $. Она всегда соответствует текущему адресу. Например, команда

jmp $

выполняет безусловный переход на саму себя, так что создается вечный цикл из одной команды.

# 3.4. Модульное, функциональное и структурное программирование применительно к специализированным вычислителям

*Определение 3.4.1*

**Модульное программирование** — это организация программы как совокупности небольших независимых блоков, называемых модулями, структура и поведение которых подчиняются определённым правилам. Использование модульного программирования позволяет упростить тестирование программы и обнаружение ошибок. Аппаратно-зависимые подзадачи могут быть строго отделены от других подзадач, что улучшает мобильность создаваемых программ.

*Определение 3.4.2*

**Модуль** — функционально законченный фрагмент [программы](https://ru.wikipedia.org/wiki/%D0%9A%D0%BE%D0%BC%D0%BF%D1%8C%D1%8E%D1%82%D0%B5%D1%80%D0%BD%D0%B0%D1%8F_%D0%BF%D1%80%D0%BE%D0%B3%D1%80%D0%B0%D0%BC%D0%BC%D0%B0). Во многих языках (но далеко не обязательно) оформляется в виде отдельного [файла](https://ru.wikipedia.org/wiki/%D0%A4%D0%B0%D0%B9%D0%BB) с [исходным кодом](https://ru.wikipedia.org/wiki/%D0%98%D1%81%D1%85%D0%BE%D0%B4%D0%BD%D1%8B%D0%B9_%D0%BA%D0%BE%D0%B4) или поименованной непрерывной её части. Некоторые языки программирования предусматривают объединение модулей в [пакеты](https://ru.wikipedia.org/wiki/%D0%9F%D0%B0%D0%BA%D0%B5%D1%82_(%D0%BF%D1%80%D0%BE%D0%B3%D1%80%D0%B0%D0%BC%D0%BC%D0%B8%D1%80%D0%BE%D0%B2%D0%B0%D0%BD%D0%B8%D0%B5)).

Основные концепции модульного программирования:

* Каждый модуль реализует единственную независимую функцию;
* Каждый модуль имеет единственную точку входа и выхода;
* Размер модуля по возможности должен быть минимизирован;
* Каждый модуль может быть разработан и закодирован различными членами бригады программистов и может быть отдельно протестирован;
* Вся система построена из модулей;
* Модуль не должен давать побочных эффектов;
* Каждый модуль не зависит от того, как реализованы другие модули.

При таком подходе сложная система разделяется на несколько частей, одновременно создаваемых различными программистами. Каждый модуль реализует единственную функцию. Размер модуля невелик, поэтому тестирование управляемо и может быть проведено тщательным образом. После кодирования и тестирования всех модулей происходит их интеграция, и тестируется вся система.

При сопровождении тестируется и отлаживается только тот модуль, который плохо работает. Очевидны преимущества в облегчении написания и тестирования программ, уменьшается стоимость их сопровождения.

*Определение 3.4.3*

**Функциональное программирование** — раздел [дискретной математики](https://ru.wikipedia.org/wiki/%D0%94%D0%B8%D1%81%D0%BA%D1%80%D0%B5%D1%82%D0%BD%D0%B0%D1%8F_%D0%BC%D0%B0%D1%82%D0%B5%D0%BC%D0%B0%D1%82%D0%B8%D0%BA%D0%B0) и [парадигма программирования](https://ru.wikipedia.org/wiki/%D0%9F%D0%B0%D1%80%D0%B0%D0%B4%D0%B8%D0%B3%D0%BC%D0%B0_%D0%BF%D1%80%D0%BE%D0%B3%D1%80%D0%B0%D0%BC%D0%BC%D0%B8%D1%80%D0%BE%D0%B2%D0%B0%D0%BD%D0%B8%D1%8F), в которой процесс [вычисления](https://ru.wikipedia.org/wiki/%D0%92%D1%8B%D1%87%D0%B8%D1%81%D0%BB%D0%B5%D0%BD%D0%B8%D0%B5) трактуется как вычисление значений [функций](https://ru.wikipedia.org/wiki/%D0%A4%D1%83%D0%BD%D0%BA%D1%86%D0%B8%D1%8F_(%D0%BC%D0%B0%D1%82%D0%B5%D0%BC%D0%B0%D1%82%D0%B8%D0%BA%D0%B0)) в математическом понимании последних (в отличие от [функций](https://ru.wikipedia.org/wiki/%D0%A4%D1%83%D0%BD%D0%BA%D1%86%D0%B8%D1%8F_(%D0%BF%D1%80%D0%BE%D0%B3%D1%80%D0%B0%D0%BC%D0%BC%D0%B8%D1%80%D0%BE%D0%B2%D0%B0%D0%BD%D0%B8%D0%B5)) как подпрограмм в [процедурном программировании](https://ru.wikipedia.org/wiki/%D0%9F%D1%80%D0%BE%D1%86%D0%B5%D0%B4%D1%83%D1%80%D0%BD%D0%BE%D0%B5_%D0%BF%D1%80%D0%BE%D0%B3%D1%80%D0%B0%D0%BC%D0%BC%D0%B8%D1%80%D0%BE%D0%B2%D0%B0%D0%BD%D0%B8%D0%B5)).

*Пример 3.4.1 (функционального программирования)*

Функция вычисления числа корней квадратного уравнения для случая a=3, b=5, c=4 (реализована на С++: в результате получим цифру 0):

#include <stdio.h>

#include <iostream>

using namespace std;

double discriminant(double a, double b, double c)

{

return b\*b - 4 \* a\*c;

}

int sign(double x)

{

return x < 0 ? -1 : x == 0 ? 0 : 1;

}

int roots(double a, double b, double c)

{

return sign(discriminant(a, b, c)) + 1;

}

int main()

{

int z = roots(3, 5, 4);

cout << z << endl;

}

*Определение 3.4.4*

**Структурное программирование** — методология разработки программного обеспечения, в основе которой лежит представление программы в виде иерархической структуры блоков. От ООП отличается тем, что в структурах хранятся только данные, а все функции снаружи.

*Пример 3.4.2 (структурного программирования)*

Функция вычисления числа корней квадратного уравнения для случая a=3, b=5, c=4 (реализована на С++: в результате получим цифру 0):

#include <stdio.h>

#include <iostream>

using namespace std;

int roots(double a, double b, double c)

{

double discriminant = b\*b - 4 \* a\*c;

if (discriminant < 0) return 0; else

if (discriminant == 0) return 1; else

return 2;

}

int main()

{

int z = roots(3, 5, 4);

cout << z << endl;

}

Применительно к специализированным вычислителям.

Специализированные вычислители обычно подразумевают необходимость вычислений в реальном времени, что означает использование Си и языка ассемблера, а они являются функциональными (в нашем понимании).

Так как передача переменных в функции производится через стек, то вместо передачи 15 переменных в функцию профессионалы создают структуру и передают указатель на экземпляр структуры.

Одним из самых распространённых способов использования языка ассемблера в проекте Си является inline assembly («ассемблерная вставка»). Способ вызова inline assembly зависит от конкретного компилятора.

Если исходный код на языке ассемблера собран ассемблером, и исходный код Си скомпилирован компилятором отдельно, то их объектные файлы могут быть слинкованы компоновщиком (linker) для последующего формирования финального исполняемого файла. Удобство такого подхода в том, что программист может использовать привычный для себя синтаксис языка ассемблера, и в том, что есть удобный доступ к исходному коду обоих частей. Минусы этого метода состоят в том, что одновременно должны быть запущены и ассемблер, и компилятор, и объектные файлы должны быть соединены вручную.

Передача параметров

Обычно, параметры передаются между функциями (написанными на Си или языке ассемблера) через стек. Например, если функция foo1() вызывает функцию foo2() с 2 параметрами (скажем, символами x и y), то перед тем, как control jumps на начало foo2(), два байта (нормальный размер символа в большинстве систем) заполняются значениями, которые должны быть переданы. Когда control jumps на новую функцию foo2(), и вы используете значения (переданные как параметры) в этой функции, они достаются из стека и используются.

Используются две техники передачи параметров:

* + - 1. Передача по значению
      2. Передача по адресу

Техники передачи параметров также могут использовать

right-to-left (C-style)

left-to-right (Pascal style)

Можно передавать и слева направо и справа налево.

На процессорах с большим количеством регистров, по стандартным договорённостям вызова помещают все параметры (даже return address) в регистры.

На процессорах с неадекватным количеством регистров, по стандартным договорённостям вызова приходится класть хотя бы некоторые параметры в стек или куда-нибудь ещё в оперативную память.

Некоторые договорённости вызова позволяют использовать "reentrant code".

Передача по значению

При передаче по значению, передаётся копия реального значения. Например, если есть функция, принимающая два значения:

void foo (char x, char y) {  
 x = x + 1;  
 y = y + 2;  
 putchar(x);  
 putchar(y);  
 }

и вы вызываете функцию таким образом:

char a,b;  
 a='A';  
 b='B';  
 foo(a,b);

то программа делает push копий ASCII значений 'A' и 'B' (65 и 66 соответсвенно) на стек до вызова функции. Можно видеть, что в функции foo() нет упоминания 'a' или 'b'. Таким образом, любые изменения, произошедшие с этими значениями в foo, не будут влиять на значения a и b в вызывающей функции.

Передача по адресу

Представьте себе ситуацию, когда вам надо передать в функцию много данных, и применить изменения, произведённые в функции к оригинальным переменным. Примером такой ситуации может служить функция, заменяющая в строке все маленькие буквы на большие (строчные на заглавные). Было бы неразумным передавать всю строку (особенно большую) функции, а когда изменения закончены, возвращать такой же большой результат. Здесь мы передаём адрес переменной в функцию. У этого метода есть два преимущества: нет необходимости передавать много данных, что сокращает время исполнения, и появляется возможность работать с данными напрямую, таким образом, к концу функции все данные из вызывающей функции уже модифицированы.

Но надо помнить, что любые изменения переменной, переданной по адресу происходят сразу с оригинальной переменной. Если этого не хочется, то надо вручную скопировать переменную перед её изменением.

# 3.5. Директивы и выражения Ассемблера: основные классы команд по назначению

*Определение 3.5.1*

**Директивы** - инструкции, не переводящиеся непосредственно в машинные команды, а управляющие работой компилятора.

Программа на языке ассемблера может содержать директивы. Набор и синтаксис их значительно разнятся и зависят не от аппаратной платформы, а от используемого транслятора.

В качестве основного набора директив можно выделить следующие:

* определение данных (констант и переменных),
* управление организацией программы в памяти и параметрами выходного файла,
* задание режима работы компилятора,
* всевозможные абстракции (то есть элементы языков высокого уровня) — от оформления процедур и функций до условных конструкций и циклов,
* макросы.

По функциональному назначению команды делят на следующие классы:

* команды пересылок данных; (MOV, CBW, CWDE, CWD, CDQ, XCHG, NOP, BSWAP)
* арифметические команды; (Mul, Div, Add, Sub, NEG, INC, DEC, IMUL)
* логические команды; (OR, XOR, AND, NOT)
* команды управления программой; (JE, JNE, JA, JB, SETE, SETNE, SETA, SETB, LOOP, LOOPE/LOOPZ, LOOPNE/LOOPNZ, JCXZ, JECXZ, JMP, CALL, RET, ENTER , LEAVE)
* команды обработки строк;(movs, cmps, seas, lods, stos)
* команды ввода/вывода и прерываний. (int, in , out)

Таблица 3.5.1

|  |  |
| --- | --- |
| BSWAP | изменяет порядок байтов в 32-битном регистре |
| NOP | команда-пустышка, занимающая один байт |
| XCHG | обменивает содержимое двух своих операндов |
| CBW, CWDE, CWD, CDQ | знаковое расширение операнда-источника, результатом является операнд удвоенного размера |
| MUL | беззнаковое умножение |
| DIV | беззнаковое деление |
| IMUL | знаковое умножение |
| IDIV | знаковое деление |
| ADD | Сложение |
| SUB | Вычитание |
| NEG | изменение знака |
| INC, DEC | увеличение или уменьшение на 1 операнда |
| AND | логическое «И» |
| OR | логическое включающее «ИЛИ» |
| XOR | логическое исключающее «ИЛИ» |
| NOT | Инвертирование |
| JE, JNE, JA, JB, … | условная передача управления |
| SETE, SETNE, SETA, SETB, ... | установка байта по условию (команда имеет один операнд, это может быть либо 8-битный регистр, либо адрес в памяти) |
| LOOP, LOOPE/LOOPZ, LOOPNE/LOOPNZ | переход для повторения цикла (если атрибут размера адреса равен 16 битам, то счетчиком служит регистр **CX**, если этот атрибут равен 32 битам, то регистр **ECX**) |
| JCXZ, JECXZ | переход, если значение счетчика равно нулю |
| JMP | безусловная передача управления |
| CALL | вызов процедуры |
| RET | возврат из процедуры |
| ENTER | образование стекового кадра для параметров процедуры |
| LEAVE | отмена действия команды ENTER перед выходом из процедуры |
| MOVS | пересылка строки |
| CMPS | сравнение двух строк |
| SEAS | поиск в строке заданного элемента |
| LODS | загрузка аккумулятора (регистров AL или АХ) из строки |
| STOS | запись элемента строки из аккумулятора (регистров АХ или AL). |

В памяти есть область, где хранится таблица прерываний. Там находятся указатели на подпрограммы прерываний. Адрес обработчика любого прерывания занимает 4 байта: сегмент (2 байта) и смещение (2 байта), причем первые 2 байта – это смещение, а вторые – сегмент. При вызове прерывания, ОС переходит на выполнение команд, которые расположены по адресу этого [сегмента:смещения]. После прерывания выполняется следующая после **INT** команда:

Таблица 3.5.2

|  |  |
| --- | --- |
| 00h | аппаратное прерывание от системного таймера |
| 0lh | внутреннее прерывание, пошаговое выполнение (при TF=1) |
| 02h | немаскируемое прерывание (вывод NMI процессора) |
| 08h | аппаратное прерывание от системного таймера |
| 09h | аппаратное прерывание от клавиатуры |
| 0Eh | аппаратное прерывание от гибкого диска |
| 10h | программное прерывание, программы BIOS управления видеосистемой |
| 13h | программное прерывание, программы BIOS управления дисками |
| 16h | программное прерывание, программы BIOS управления клавиатурой |
| IDh | не вектор, адрес таблицы видеопараметров, используемой BIOS |
| lEh | не вектор, адрес таблицы параметров дискеты, используемой BIOS |
| 18h | программное прерывание, диспетчер функций DOS |
| 22h | программное прерывание, адрес перехода при завершении процесса, используемый DOS |
| 23h | программное прерывание, обработчик прерываний по <Ctrl>/C, используемый DOS |
| 25h | программное прерывание, абсолютное чтение диска (функция DOS) |
| 26h | программное прерывание, абсолютная запись на диск (функция DOS) |
| 60h...66h | зарезервировано для программных прерываний пользователя |
| 68h...6Fh | программные прерывания, свободные векторы |
| 70h | аппаратное прерывание от часов реального времени (с питанием от аккумулятора) |
| 76h | аппаратное прерывание от жесткого диска |

Вывод строки на экран монитора осуществляется с помощью 9-ой функции прерывания 18h операционной системы MS DOS.

Программа, реализующая вывод строки, имеет вид:

MOU DX,0140

MOU AH,09

INT 21

INT 20

Ввод строки с клавиатуры осуществляется с помощью 10-ой (Ah) функции прерывания 18h операционной системы MS DOS.

MOU DX,0150

MOU AH,0A

INT 21

INT 20

\*\*\*

Для работы с портами ввода-вывода из ассемблера существуют команды процессора IN и OUT. Они позволяют отправлять в шину данные и читать из шины. Для отправки байта в порт ввода-вывода нужно записать отправляемое значение в регистр AL/AX/EAX и вызвать команду OUT.

*Определение 3.5.2*

**Оптимизация** – модификация кода с целью улучшения его эффективности. В конечном итоге эффективность определяет быстродействие программы.

Вот некоторые из самых общих процедур этой категории.

* Замещение универсальных инструкций на учитывающие конкретную ситуацию, например, замена команды умножения на степень двойки на команды сдвига (отказ от универсальности).
* Уменьшение количества передач в программе: за счет преобразования подпрограмм в макрокоманды для непосредственно включения в исполнимый код; за счет преобразования условных переходов, так чтобы условие перехода оказывалось истинным относительно реже, чем условие для его отсутствия; перемещение условий общего характера к началу разветвленной последовательности переходов; преобразование вызовов, сразу за которыми следует возврат в программу, в переходы ("сращивание хвостов" и "устранение рекурсивных хвостов") и так далее.
* Оптимизация циклов, в том числе перемещение вычислений не изменяющихся величин за пределы циклов, разворачивание циклов и "соединение" отдельных циклов, выполняемых одно и то же число раз, в единый цикл ("сжатие цикла").
* Максимальное использование всех доступных регистров, за счет хранения в них рабочих значений всякий раз, когда это возможно, чтобы минимизировать число обращений к памяти, упаковка множественных значений или флагов в регистры и устранение излишних продвижений стека (особенно на входах и выходах подпрограмм).
* Использование специфических для данного процессора инструкций
* Никогда не делайте в цикле ничего такого, что можно сделать за его пределами.
* Везде, где это возможно, избавляйтесь от передач управления внутри циклов.

В качестве примера рассмотрим не слишком удачную программу, которая суммирует все кратные 5 элементы массива со словной точностью и оставляет результат в регистре AX:

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| Исходный код | | | Оптимизированный код | | |
| Items | equ | 100 | Items | equ | 100 |
| Array | dw | Items dup(?) | Array | dw | Items dup(?) |
|  | xor | cx, cx |  | xor | ax, ax |
|  | xor | ax, ax |  | mov | bx,offset Array |
| Lab1: | mov | bx, cx | Lab1: | add | ax, [bx] |
|  | add | bx, cx |  | add | bx, 10 |
|  | add | bx, cx |  | cmp | bx, offset Array + Items/5 |
|  | add | bx, cx |  | jne | Lab1 |
|  | add | bx, cx |  |  |  |
|  | add | ax, Array[bx] |  |  |  |
|  | inc | Cx |  |  |  |
|  | cmp | cx, Items/5 |  |  |  |
|  | jne | Lab1 |  |  |  |

Один из способов избавиться от сравнений и условных переходов называется объединением, или сращиванием циклов. При этом коды реорганизуются так, чтобы сделать один цикл из нескольких, повторяющихся одинаковое число раз.

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| Исходный код | | | Оптимизированный код | | |
|  | mov | cx, 100 |  | mov | cx, 100 |
| Lab1: | xor | ax, ax | Lab1: | xor | ax, ax |
|  | … |  |  | … |  |
|  | loop | Lab1 |  | xor | bx, bx |
|  | mov | cx, 100 |  | … |  |
| Lab2: | Xor | bx, bx |  | loop | Lab1 |
|  | … |  |  |  |  |
|  | loop | Lab2 |  |  |  |

Другой способ избавиться от циклов - "размотать" их, т.е. устранить управляющие циклом кодовые последовательности, просто повторив содержимое цикла нужное число раз. Это дает особенно хорошие результаты в тех случаях, когда время, необходимое для исполнения содержимого цикла, оказывается меньше, чем время выполнения операций, управляющих циклом.

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| Исходный код | | | Оптимизированный код | | |
|  | mov | cx, 3 |  | add | ax, [bx+0] |
| Lab1: | mov | ax, [bx] |  | add | ax, [bx+2] |
|  | Add | bx, 2 |  | add | ax, [bx+4] |
|  | Loop | Lab1 |  |  |  |

Устранение рекурсивных хвостов очень похоже на сращивание хвостов. Когда программа последовательно вызывает сама себя и этот вызов расположен непосредственно перед последней командой RET в программе, вызов может быть преобразован в переход, что и увеличит скорость, и уменьшит необходимый объем памяти в стеке.

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| Исходный код | | | Оптимизированный код | | |
| Proc1 | proc | Near | Proc1 | proc | Near |
|  | … |  |  | … |  |
|  | cmp | ax, MyVar |  | cmp | ax, MyVar |
|  | Je | Exit |  | jne | Proc1 |
|  | Call | Proc1 |  | ret |  |
| Exit: | Ret | bx, bx | Proc1 | endp |  |
| Proc1 | endp |  |  |  |  |

*Определение 3.5.3*

**Такты ожидания** — простаивание процессора, ожидая готовности более медленных узлов компьютера.

Все процессоры содержат исполнительное устройство — конвейер. Современные процессоры имеют несколько конвейеров, которые могут исполнять команды параллельно. Можно две операции выполнять параллельно, или нельзя, решает аппаратура в процессе декодирования операции. Т.е., к примеру, если есть операция чтения из памяти в регистр %EAX и операция инкрементации регистра %EBX, то эти две операции аппаратура может исполнять параллельно, поскольку они не конфликтуют по ресурсам. Но вот если бы вторая операция была операцией инкрементирования регистра %EAX, то параллельное исполнение было бы невозможным: сначала нужно дождаться загрузки значения из памяти в %EAX и только потом значение регистра инкрементировать.

Простые операции типа целочисленного сложения-вычитания или битовой арифметики исполняются за один такт на любом более-менее приличном процессоре. Более сложные операции типа умножения или деления, как правило, исполняются несколько тактов. Связано это со сложным устройством аппаратных элементов, которые исполняют данные операции. Операции чтения из памяти на любом современном высокоскоростном процессоре исполняется несколько тактов. Точной причины этого я не знаю, но есть некоторый минимальный порог времени, требуемый для прочтения данных из транзисторной матрицы: нужно вычислить, из какого места нужно данные прочитать, а потом ещё и прочитать сами данные. Дальше эти данные через шину нужно передать на процессор. На всех железках процессор и память представляют физически разведённые друг от друга устройства на общей плате. Чтение из памяти может исполняться несколько десятков или даже сотен машинных тактов. Именно из-за этого в современных процессорах присутствует кэш (и даже не один), который физически находится на процессоре или очень близко к нему, а потому имеет более короткое время доступа к данным, но ограниченный размер памяти. И даже в тех случаях, когда данные оказываются в кэше первого уровня, их чтение занимает порядка трёх машинных тактов.

Есть много тонкостей при работе с конвейером типа того, что конвейер разбит на несколько стадий: декодирование, чтение входных аргументов, исполнение, запись результатов и целая куча промежуточных стадий. Те операции, которые исполняются один такт, на самом деле проводят в конвейере несколько тактов. Но в то время, когда первая операция проходит через вторую стадию конвейера, вторая операция уже попадает на первую стадию.

При параллельной обработке команд на разных конвейерах максимальный эффект достигается на однотипных командах, не зависящих друг от друга. Если в программе присутствуют команды разного типа, то на конвейере вводятся такты ожидания.

*Определение 3.5.4*

**Stall** (ступор) — остановка работы конвейера или одной или нескольких его стадий из-за того, что ресурс в данный момент не готов. Например, задержка после обращения к медленной памяти, которая на данный момент пустая.

Ступор одной стадии в течение одного такта называется пузырь (bubble). Во избежание ступоров и приближения достижимой производительности к её теоретическому максимуму применяются многочисленные методы поддержания конвейера в максимально загруженном состоянии.

# 3.6. Директивы и выражения Ассемблера для работы с памятью (куча и стек).

*Определение 3.6.1*

**Стек** - специально выделенная область памяти для передачи или сохранения данных. Его область используется многократно. Отличительной особенностью стека является своеобразный порядок выборки содержащихся в нем данных: в любой момент времени в стеке доступен только верхний элемент, т.е. элемент, загруженный в стек последним. Выгрузка из стека верхнего элемента делает доступным следующий элемент.

*Определение 3.6.2*

Под **стековой памятью** мы понимаем область ОЗУ или Data со старшими адресами. Если рассматривать ОЗУ как набор ячеек, где ячейки с младшими адресами расположены наверху, а со старшими – внизу, то при включении данных в стек (стопка бумаг) эти данные как бы кладутся сверху ранее занятых ячеек стека, а при извлечении данных из стека, они берутся из верхних ячеек стековой памяти. Контролирует стековую память специальный регистр SP (Stack-Pointer), который содержит адрес ячейки памяти соответствующей верхушке стека. Этот регистр автоматически изменяет свое содержимое на два (увеличивает или уменьшает) при выполнении команд обращения к стековой памяти (Push, Pop, Call, Ret).

Команда Call, также, как и любая другая команда ветвления, в качестве операнда использует адрес или метку точки перехода.

Команда Call вызывает выполнение последовательно двух действий:

1. Содержимое программного счетчика, равное адресу ячейки памяти ПЗУ с командой, которая будет выполняться на последующем шаге, запоминается в стековой памяти, чтобы использовать ее при возврате из подпрограммы.
2. В регистр «программный счетчик» загружается операнд команды Call, равный численному адресу первой команды вызываемой подпрограммы.

При возврате из подпрограммы по команде Ret происходит только одно действие: в программный счетчик загружается число из стековой памяти, которое было ранее туда загружено по команде Call. Соответственно после команды Ret будет выполняться команда, следующая в программе за командой Call.

Если в подпрограмме происходит обращение к другой подпрограмме (вложенные подпрограммы), то сохранение точек возврата происходит аналогичным образом.

**Директивы для работы с памятью.**

Ассемблер поддерживает множество директив. Директивы не транслируются непосредственно в коды операции. Напротив, они используются, чтобы корректировать местоположение программы в памяти, определять макрокоманды, инициализировать память и так далее. То есть это указания самому ассемблеру, а не команды микроконтроллера.

***Синтаксис*** всех директив следующий:

[.директива]

То есть перед директивой должна стоять точка. Иначе ассемблер воспринимает это как метку.

В таблице 3.6.1 представлено краткое описание директив, касающихся памяти.

Таблица 3.6.1

|  |  |
| --- | --- |
| **Директива** | **Описание** |
| ***BYTE*** | Зарезервировать байт под переменную |
| ***CSEG*** | Сегмент кодов |
| ***DSEG*** | Сегмент данных |
| ***DB*** | Задать постоянным(и) байт(ы) в памяти (т.е., выделение памяти для одного или нескольких байтов) |
| ***DW*** | Задать постоянное(ые) слово(а) в памяти (т.е., выделение памяти для одного или нескольких 16-разрядных полуслов) |
| ***DD*** | Выделение памяти для одного или нескольких 32-разрядных слов |
| ***DQ*** | Выделение памяти для одного или нескольких 64-разрядных двойных слов |
| ***ESEG*** | Сегмент EEPROM |
| ***ORG*** | Начальный адрес программы |

*Определение 3.6.3*

**Директива BYTE**. Это директива резервирует байты ОЗУ (в области Data). Имеет один обязательный операнд, указывающий количество выделенных байтов. Если необходимо чтобы обращение к выделенным байтам происходило по имени, то вводится метка (в данном случае это метка Var1).

**Пример №1:**

.DSEQ

Var1: .BYTE 1

.CSEQ

ldi r30, low(varl)

ldi r31, high(varl)

ld r1, z

*Определение 3.6.4*

**Директива .DSEQ**. Определяет, что нижеследующие строки относятся к области Data ОЗУ. Эта директива не имеет операндов.

*Определение 3.6.5*

**Директива .CSEQ**. Указывает компилятору, что последующие строчки программы должны компилироваться и располагаться в области ПЗУ.

В последующих командах используется переменная Var 1, определяющая содержимое ячеек памяти ОЗУ. Выражения low и high являются стандартными выражениями языка Assembler и предназначены для определения старшего и младшего байтов адреса переменной Var 1, указанного в качестве операнда этих выражений.

*Определение 3.6.6*

**Директива .ORG** устанавливает счетчик в положение равным заданной величине, которая входит как параметр в эту директиву. Для сегмента данных она устанавливает счетчик положения ОЗУ, для сегмента программ или памяти программ она устанавливает содержимое РС (программного счетчика), а для сегмента Eeprom устанавливает положение в области Eeprom.

Если директиве предшествует метка, то она становится равной адресу, указанному в параметре директивы при компиляции программы.

Перед началом компиляции РС и счетчик Eeprom нулю, а счетчик ОЗУ=32.

Для ОЗУ и Eeprom используются побайтные счетчики, а для программного сегмента пословные.

*Определение 3.6.7*

**Директива ESEG - EEPROM Segment.** Указывает на начало сегмента EEPROM памяти. Ассемблируемый файл может содержать несколько EEPROM сегментов, которые будут собраны в один сегмент при ассемблировании. Обычно сегмент EEPROM состоит из DB и DW директив (и меток). Сегмент EEPROM памяти имеет свой собственный счетчик. Директива ORG может использоваться для размещения переменных в нужной области EEPROM.

Синтаксис:.ESEG

**Пример:**

.DSEG ; Начало сегмента данных

var1: .BYTE 1 ; Резервировать 1 байт под переменную var1

table: .BYTE tab\_size ; Зарезервировать tab\_size байт.

.ESEG

eevar1: .DW 0xffff ; Записать 1 слово в EEPROM

*Определение 3.6.8*

**Директива DB -** определяет байт(ы) в программной памяти или в EEPROM.

Директива DB резервирует ресурсы памяти в программной памяти или в EEPROM. Директиве должна предшествовать метка. DB задает список выражений, и должна содержать по крайней мере одно выражение. Размещать директиву следует в сегменте кодов или в EEPROM сегменте. Список выражений представляет собой последовательность выражений, разделенных запятыми. Каждое выражение должно быть величиной между –128 и 255. Если директива указывается в сегменте кодов и список выражений содержит более двух величин, то выражения будут записаны так, что 2 байта будут размещаться в каждом слове Flash-памяти.

Синтаксис:

LABEL: .DB список выражений

**Пример:**

.CSEG

consts: .DB 0, 255, 0b01010101, -128, 0xaa

.ESEG

const2: .DB 1,2,3

*Определение 3.6.9*

**Директива DW –** определяет слово(а) в программной памяти или в EEPROM.

Директива DW резервирует ресурсы памяти в программной памяти или в EEPROM. Директиве должна предшествовать метка. DW задает список выражений, и должна содержать по крайней мере одно выражение. Размещать директиву следует в сегменте кодов или в EEPROM сегменте. Список выражений представляет собой последовательность выражений, разделенных запятыми. Каждое выражение должно быть величиной между –32768 и 65535.

Синтаксис:

LABEL: .DW список выражений

*Определение 3.6.10*

**Статическая память с произвольным доступом** (**SRAM,** анг. static random access memory) — полупроводниковая оперативная память, в которой каждый двоичный или троичный разряд хранится в схеме с положительной обратной связью, позволяющей поддерживать состояние без регенерации, необходимой в динамической памяти ([DRAM](https://ru.wikipedia.org/wiki/DRAM)).

Память данных почти полностью доступна программе пользователя и большинство команд ассемблера предназначено для обмена данными с ней. Команды пересылки данных предоставляют возможность непосредственной и косвенной адресации ячеек SRAM, непосредственной адресации регистров ввода/вывода и регистров общего назначения. Так как каждому регистру сопоставлена ячейка памяти, то обращаться к ним можно не только командами адресации регистров, но и командами адресации ячеек SRAM.

**Например**, команда:

MOV R10,R15 - скопировать регистр R15 в регистр R10

делает абсолютно то же самое, что и команда:

LDS R10,$0015 - загрузить в регистр R10 содержимое ячейки с адресом $0015

То же самое относится и к регистрам ввода/вывода. Для них предусмотрены специальные команды:

INRd,P - загрузить данные из порта I/O с номером Р в регистр Rd

OUT P,Rd - записать данные из регистра Rd в порт I/O с номером Р.

При использовании этих команд номер порта указывается в диапазоне 0<P<63. При использовании команд адресации ячеек памяти для работы с регистрами ввода/вывода указывается адрес регистра в памяти данных $0020-$005F.

**Пример применения разных команд:**

LDI R16,$FF

OUT $12,R16 - записать в PORTD число 255.

STS $0032,R16 - записать непосредственно в ячейку $0032 число 255.

Адрес регистра ввода/вывода в SRAM получается прибавлением к номеру порта числа $20.

Память программ является ПЗУ и изменяется только при программировании кристалла. Константы можно располагать в памяти программ в виде слов.

**Например**,

.dw $033f,$676d,$7653,$237e,$777f

Для работы с данными, расположенными в памяти программ, предусмотрена команда

*Определение 3.6.11*

**LPM** - загрузить байт памяти программ, на который указывает регистр Z в регистр R0.

Адрес байта константы определяется содержимым регистра Z. Старшие 15 битов определяют слово адреса (от 0 до 4к) состояние младшего бита определяет выбор младшего байта (0) или старшего байта (1).

При работе с портами ввода/вывода следует учитывать следующую особенность. Если вывод порта сконфигурирован как выход, то его переключение производится через регистр данных (PORTA, PORTB, PORTC, PORTD), если вывод сконфигурирован как вход, то его опрос следует производить через регистр выводов входа порта (PINA, PINB, PINC, PIND).

Особенностью использования арифметических и логических команд является то, что некоторые из них работают только с регистрами R16-R31.

**Пример:** CPI Rd,K - сравнить регистр Rd с константой К. 16<d<31.

Команды CBI и SBI работают только с младшими 32-мя регистрами ввода/вывода.

При использовании подпрограмм нужно обязательно определять стек. Для этого нужно занести значения адреса вершины стека в регистры SPH и SPL.

*Определение 3.6.12*

**Команда LD-** загрузить косвенно из SRAM в регистр с использованием индекса X.Загружает косвенно один байт из SRAM в регистр. Положение байта в SRAM указывается 16-разрядным регистром-указателем X в регистровом файле. Обращение к памяти ограничено текущей страницей объемом 64 Кбайта. Для обращения к другой странице SRAM необходимо изменить регистр RAMPX в I/O области. Регистр-указатель X может остаться неизменным после выполнения команды, но может быть инкрементирован или декрементирован. Использование регистра-указателя X обеспечивает удобную возможность обращения к матрицам, таблицам, указателю стека.

**Пример:**

clr r27 ; Очистить старший байт X

ldi r26, $20 ; Установить $20 в младший байт X

ld r0, X+ ; Загрузить в r0 содержимое SRAM по адресу $20 (X постинкрементируется)

ld r1, X ; Загрузить в r1 содержимое SRAM по адресу $21

ldi r26, $23 ; Установить $23 в младший байт X

ld r2, X ; Загрузить в r2 содержимое SRAM по адресу $23

ld r3, -X ; Загрузить в r3 содержимое SRAM по адресу $22 (X преддекрементируется)

(Слов: 1 (2 байта), циклов: 2)

*Определение 3.6.13*

**Команда LDS -** загрузить непосредственно из SRAM.Выполняется загрузка одного байта из SRAM в регистр. Можно использовать 16-разрядный адрес. Обращение к памяти ограничено текущей страницей SRAM объемом 64 Кбайта. Команда LDS использует для обращения к памяти выше 64 Кбайт регистр RAMPZ.

**Пример:**

lds r2, $FF00 ; Загрузить r2 содержимым SRAM по адресу $FF00

add r2, r1 ; Сложить r1 с r2

sts $FF00, r2 ; Записать обратно

(Слов: 2 (4 байта), циклов: 3)

*Определение 3.6.14*

**Команда LPM -** загрузить байт памяти программ.Загружает один байт, адресованный регистром Z, в регистр 0 (R0). Команда обеспечивает эффективную загрузку констант или выборку постоянных данных. Память программ организована из 16-разрядных слов и младший значащий разряд (LSB) 16-разрядного указателя Z выбирает или младший (0) или старший (1) байт. Команда может адресовать первые 64 Кбайта (32 Кслов) памяти программ.

**Пример:**

clr r31 ; Очистить старший байт Z

ldi r30, $F0 ; Установить младший байт Z

lpm ; Загрузить константу из памяти программ

отмеченную Z (r31 : r30)

(Слов: 1 (2 байта), циклов: 3)

*Определение 3.6.15*

**Команда MOV -** копировать регистр. Команда создает копию одного регистра в другом регистре. Исходный регистр Rr остается неизменным, в регистр назначения Rd загружается копия содержимого регистра Rr.

**Пример:**

mov r16, r0 ; Копировать r0 в r16

call check ; Вызвать подпрограмму

. . .

check cpi r16, $11 ; Сравнить r16 с $11

. . .

ret ; Вернуться из подпрограммы

(Слов: 1 (2 байта), циклов: 1)

*Определение 3.6.16*

**Команда CALL -** Выполнить длинный вызов подпрограммы. Вызов подпрограммы из памяти программ. Адрес возврата (к команде после CALL) сохраняется в стеке.

**Пример:**

mov r16, r0 ; Копировать r0 в r16

call check ; Вызвать подпрограмму

nop ; Продолжать (пустая операция)

. . .

check: cpi r16, $42 ; Проверить содержит ли r16 заданное значение

breq error ; Перейти если содержит

ret ; Вернуться из подпрограммы

. . .

error: rjmp error ; Бесконечная петля

(Слов: 2 (4 байта), циклов: 4)

*Определение 3.6.17*

**Команда RET -** вернуться из подпрограммы.Команда возвращает из подпрограммы. Адрес возврата загружается из стека.

**Пример:**

call routine ; Вызвать подпрограмму

. . .

routine: push r14 ; Сохранить r14 в стеке

. . .

pop r14 ; Восстановить r14

ret ; Вернуться из подпрограммы

(Слов: 1 (2 байта), циклов: 4)

*Определение 3.6.18*

**Команда STS -** загрузить непосредственно в SRAM. Выполняется запись одного байта из регистра в SRAM. Можно использовать 16- разрядный адрес. Обращение к памяти ограничено текущей страницей SRAM объемом 64 Кбайта. Команда STS использует для обращения к памяти выше 64 Кбайт регистр RAMPZ.

**Пример:**

lds r2, $FF00 ; Загрузить в r2 содержимое SRAM по адресу $FF00

add r2, r1 ; Сложить r1 с r2

sts $FF00, r2 ; Записать обратно

(Слов: 2 (4 байта), циклов: 3)

**Команды работы со стеком**

Работа со стеком имеет непосредственное отношение к процедурам, т.к. стек используется для передачи параметров и для хранения локальных данных процедур. В принципе, для работы со стеком существуют всего две операции: положить данные и взять данные. Для каждой операции существует несколько команд, которые отличаются тем, с какими данными они работают.

Для того чтобы положить данные в стек используется команда ***PUSH***:

*PUSH*<операнд>

Операнд может быть регистром, ячейкой памяти или непосредственным операндом. Размер операнда должен быть 2 или 4 байта. Операнд кладётся на вершину стека, а значение регистра ESP уменьшается на размер операнда (Рис. 3.6.1).

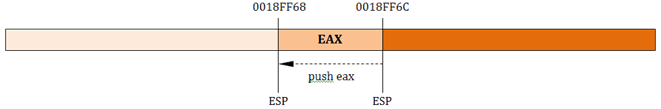


Рис. 3.6.1

Для того чтобы взять данные из стека используется команда ***POP***:

*POP*<операнд>

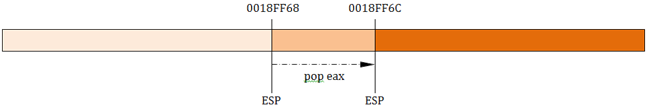
Операнд может быть регистром или ячейкой памяти. Размер операнда должен быть 2 или 4 байта. В соответствии с размером операнда из вершины стека берутся 2 или 4 байта и помещаются в указанный регистр или ячейку памяти. Значение регистра ESP увеличивается на размер операнда (Рис. 3.6.2).

Рис. 3.6.2

Кроме этих основных команд существуют ещё команды, которые позволяют сохранять в стеке и восстанавливать из стека содержимое всех регистров общего назначения, и команды, которые позволяют сохранять в стеке и восстанавливать из стека содержимое регистра флагов.

***PUSHA***

***PUSHAD***

Команда *PUSHA* сохраняет в стеке содержимое регистров AX, CX, DX, BX, SP, BP, SI, DI. Команда *PUSHAD* сохраняет в стеке содержимое регистров EAX, ECX, EDX, EBX, ESP, EBP, ESI, EDI. Для регистра (E)SP сохраняется значение, которое было до того, как мы положили регистры в стек. После этого значение регистра (E)SP изменяется как обычно.

***POPA***

***POPAD***

Эти команды противоположны предыдущим – они восстанавливают из стека значения регистров (E)DI, (E)SI, (E)BP, (E)SP, (E)BX, (E)DX, (E)CX, (E)AX. Содержимое регистра (E)SP не восстанавливается из стека, а изменяется как обычно.

*P****USHF***

***PUS****HFD*

Команда *PUSHF* сохраняет в стеке младшие 16 бит регистра флагов. Команда *PUSHFD* сохраняет в стеке все 32 бита регистра флагов.

***POPF***

***POPFD***

Команда *POPF* восстанавливает из стека младшие 16 бит регистра флагов. Команда *POPFD* восстанавливает из стека все 32 бита регистра флагов.

*Определение 3.6.19*

**Куча** — это область памяти, выделяемая и освобождаемая динамически (при запуске программы для хранения данных и глобальных переменных).

**Таблица с основными директивами ассемблера.**

Таблица 3.6.2

|  |  |
| --- | --- |
| ***Директива*** | ***Описание*** |
| SEGMENT | Начало нового сегмента (текста, данных и т. п.) с определенными атрибутами |
| ENDS | Завершение текущего сегмента |
| ALIGN | Управление выравниванием следующей команды или данных |
| DEVICE | Задать для какого типа микроконтроллера компилировать |
| DEF | Задать символическое имя регистру |
| EQU | Определение нового символа, равного данному выражению |
| EXIT | Выход из файла |
| INCLUDE | Включить исходный код из другого файла(или вызов другого файла и включение его в текущий файл) |
| LIST | Включить генерацию .lst – файла |
| NOLIST | Выключить генерацию .lst – файла |
| SET | Установите символ равный выражению |
| PROC | Начало процедуры |
| ENDP | Завершение процедуры |
| MACRO | Начало макроса |
| ENDM | Завершение макроса |
| PUBLIC | Экспорт имени, определенного в данном модуле |
| EXTERN | Импорт имени из другого модуля |
| IF | Начало условного ассемблирования программы на основе данного выражения |
| ELSE | Начало условного ассемблирования программы, если условие для директивы 1Р не выполнено |
| ENDIF | Завершение условного ассемблирования программы |
| COMMENT | Определение нового символа начала поля комментариев |
| END | Завершение ассемблерной программы |

# 3.7. Директивы и выражения Ассемблера для осуществления арифметических, побитовых и логических операций.

**Арифметические операции:**

* Сложение: ADD<операнд №1>, <операнд №2>. Команда *ADD* складывает операнды и записывает их сумму на место первого операнда.
* Вычитание: SUB<операнд №1>, <операнд №2>. Команда *SUB* вычитает из первого операнда второй и записывает полученную разность на место первого операнда. Возможно сложение и вычитание как знаковых, так и беззнаковых чисел любого размера. Команды меняют флаги AF, CF, OF, PF, SF и ZF.
* Команды инкремента: ***INC***<операнд> и декремента: DEC<операнд> увеличивают и уменьшают на 1 свой операнд. Операндом может быть регистр или ячейка памяти любого размера. Команды инкремента и декремента выгодны тем, что они занимают меньше места, чем соответствующие команды сложения и вычитания.
* Изменение знака: NEG<операнд>. Операндом может быть регистр или ячейка памяти любого размера. Команда *NEG* рассматривает свой операнд как число со знаком и меняет знак операнда на противоположный.
* Умножение: MUL<операнд> - для беззнакового умножения. Операнд может быть регистром или ячейкой памяти, но не может быть непосредственным операндом.

***IMUL***: для знакового умножения. Варианты:

***IMUL***<операнд>

***IMUL***<операнд>, <непосредственный операнд>

***IMUL***<операнд№1>, <операнд№2>, <непосредственный операнд>

***IMUL***<операнд№1>, <операнд№2>

* Деление: ***DIV***<операнд>; Беззнаковое деление ***IDIV***<операнд>; Знаковое деление.

В командах указывается только один операнд – делитель, который может быть регистром или ячейкой памяти, но не может быть непосредственным операндом. Местоположение делимого и результата для команд деления фиксировано.

*Определение 3.7.1*

**Поразрядные операции** — реализуют одну и ту же логическую операцию над всеми битами переменной. К поразрядным операциям относят также операции сдвига.

* **Логические команды**

Операция отрицания меняет значение всех битов переменной на противоположное.

***NOT***<операнд>

Операция поразрядное «и» выполняет логическое умножение всех пар бит операндов.

***AND***<операнд1>, <операнд2>

Операция поразрядное «или» выполняет логическое сложение всех пар бит операндов.

***OR***<операнд1>, <операнд2>

Операция поразрядное исключающее «или» выполняет сложение по модулю 2 всех пар бит операндов: XOR<операнд1>, <операнд2>

Операции *AND*, *OR* и *XOR* имеют по два операнда. Первый может быть регистром или ячейкой памяти, а второй – регистром, ячейкой памяти или непосредственным операндом. Операнды должны иметь одинаковый размер. Результат помещается на место первого операнда.

Операция *XOR* имеет интересную особенность – если значения операндов совпадают, то результатом будет значение 0. Поэтому операцию *XOR* используют для обнуления регистров – она выполняется быстрее, чем запись нуля с помощью команды *MOV*.

|  |  |
| --- | --- |
| xor eax, eax | При любом значении EAX результат будет  равен 0 |

Операцию *XOR* можно также использовать для обмена значений двух переменных.

|  |  |
| --- | --- |
| xor eax, ebx | EAX = EAX xor EBX |
| xor ebx, eax | Теперь EBX содержит исходное значение EAX |
| xor eax, ebx | А теперь EAX содержит исходное значение  EBX |

* **Команды сдвига:** Операции сдвига вправо и сдвига влево сдвигают биты в переменной на заданное количество позиций. Каждая команда сдвига имеет две разновидности:

<мнемокод><операнд>, <непосредственный операнд>

<мнемокод><операнд>, **CL**

Первый операнд должен быть регистром или ячейкой памяти. Именно в нём осуществляется сдвиг. Второй операнд определяет количество позиций для сдвига, которое задаётся непосредственным операндом или хранится в регистре CL (и только CL). Существует несколько разновидностей сдвигов, которые отличаются тем, как заполняются «освобождающиеся» биты.

*Определение 3.7.2*

**Логические сдвиги**

При *логическом сдвиге* «освобождающиеся» биты заполняются нулями. Последний ушедший бит сохраняется во флаге CF.

***SHL***<операнд>, <количество>; Логический сдвиг влево

***SHR***<операнд>, <количество>; Логический сдвиг вправо

*Определение 3.7.3*

**Арифметические сдвиги**

Арифметический сдвиг влево эквивалентен логическому сдвигу влево (это одна и та же команда) – «освобождающие» биты заполняются нулями. При арифметическом сдвиге вправо «освобождающиеся» биты заполняются знаковым битом. Последний ушедший бит сохраняется во флаге CF.

***SAL***<операнд>, <количество>; Арифметический сдвиг влево

***SAR***<операнд>, <количество>; Арифметический сдвиг вправо

*Определение 3.7.4*

**Циклические сдвиги**

При циклическом сдвиге «освобождающиеся» биты заполняются ушедшими битами. Последний ушедший бит сохраняется во флаге CF.

***ROL***<операнд>, <количество>; Циклический сдвиг влево

***ROR***<операнд>, <количество>; Циклический сдвиг вправо

*Определение 3.7.5*

**Расширенные сдвиги**

В расширенных сдвигах участвуют два регистра или ячейка памяти и регистр, которые как бы объединяются в единое целое и «освобождающиеся» биты одного операнда заполняются битами из другого операнда.

***SHLD***<операнд №1>, <операнд №2>, <количество>; Расширенный сдвиг влево

***SHRD***<операнд №1>, <операнд №2>, <количество>; Расширенный сдвиг вправо

Команда SHLD сдвигает влево биты операнда №1 на указанное количество позиций. Младшие («освободившиеся») биты операнда №1 заполняются старшими битами операнда №2. Сам операнд №2 не меняется.

Команда SHRD сдвигает вправо биты операнда №1 на указанное количество позиций. Старшие («освободившиеся») биты операнда №1 заполняются младшими битами операнда №2. Сам операнд №2 не меняется.

Количество, как и в других операциях сдвига, задаётся непосредственным операндом или хранится в регистре CL. Но используются только последние 5 бит операнда, определяющего количество, т.е. максимальное количество позиций сдвига равно 32.

**Основные регистры и флаги**

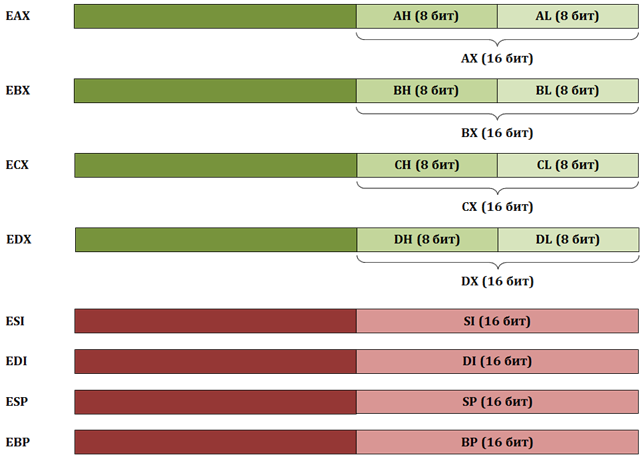
*Определение 3.7.6*

**Регистры** – это специальные ячейки памяти, расположенные непосредственно в процессоре. Работа с регистрами выполняется намного быстрее, чем с ячейками оперативной памяти, поэтому регистры активно используются как в программах на языке ассемблера, так и компиляторами языков высокого уровня. Регистры можно разделить на регистры общего назначения, указатель команд, регистр флагов и сегментные регистры.

*Определение 3.7.7*

**Регистры общего назначения**

К регистрам общего назначения относится группа из 8 регистров, которые можно использовать в программе на языке ассемблера. Все регистры имеют размер 32 бита и могут быть разделены на 2 или более частей.



Как видно из рисунка, регистры ESI, EDI, ESP и EBP позволяют обращаться к младшим 16 битам по именам SI, DI, SP и BP соответственно, а регистры EAX, EBX, ECX и EDX позволяют обращаться как к младшим 16 битам (по именам AX, BX, CX и DX), так и к двум младшим байтам по отдельности (по именам AH/AL, BH/BL, CH/CL и DH/DL).

Названия регистров происходят от их назначения:

* **EAX/AX/AH/AL** (*accumulator register*) – аккумулятор;
* **EBX/BX/BH/BL** (*base register*) –регистр базы;
* **ECX/CX/CH/CL** (*counter register*) – счётчик;
* **EDX/DX/DH/DL** (*data register*) – регистр данных;
* **ESI/SI** (*source index register*) – индекс источника;
* **EDI/DI** (*destination index register*) – индекс приёмника (получателя);
* **ESP/SP** (*stack pointer register*) – регистр указателя стека;
* **EBP/BP** (*base pointer register*) – регистр указателя базы кадра стека.

Несмотря на существующую специализацию, все регистры можно использовать в любых машинных операциях. Однако надо учитывать тот факт, что некоторые команды работают только с определёнными регистрами. Например, команды умножения и деления используют регистры EAX и EDX для хранения исходных данных и результата операции. Команды управления циклом используют регистр ECX в качестве счётчика цикла.

Ещё один нюанс состоит в использовании регистров в качестве базы, т.е. хранилища адреса оперативной памяти. В качестве регистров базы можно использовать любые регистры, но желательно использовать регистры EBX, ESI, EDI или EBP. В этом случае размер машинной команды обычно бывает меньше.

*Определение 3.7.8*

**Указатель команд**

Регистр EIP (*указатель команд*) содержит смещение следующей подлежащей выполнению команды. Этот регистр непосредственно недоступен программисту, но загрузка и изменение его значения производятся различными командами управления, к которым относятся команды условных и безусловных переходов, вызова процедур и возврата из процедур.

*Определение 3.7.9*

**Регистр флагов**

В данном случае флаг – это бит, принимающий значение 1 («флаг установлен»), если выполнено некоторое условие, и значение 0 («флаг сброшен») в противном случае. Процессор имеет регистр флагов, содержащий набор флагов, отражающий текущее состояние процессора.

Таблица 3.7.1

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| № бита | Обозначение | Название | Описание | Тип флага |
| ***FLAGS*** | | | | |
| 0 | CF | CarryFlag | Флаг переноса | Состояние |
| 1 | 1 | | Зарезервирован |  |
| 2 | PF | ParityFlag | Флаг чётности | Состояние |
| 3 | 0 | | Зарезервирован |  |
| 4 | AF | AuxiliaryCarryFlag | Вспомогательный флаг переноса | Состояние |
| 5 | 0 | | Зарезервирован |  |
| 6 | ZF | ZeroFlag | Флаг нуля | Состояние |
| 7 | SF | SignFlag | Флаг знака | Состояние |
| 8 | TF | TrapFlag | Флаг трассировки | Системный |
| 9 | IF | InterruptEnableFlag | Флаг разрешения прерываний | Системный |
| 10 | DF | DirectionFlag | Флаг направления | Управляющий |
| 11 | OF | OverflowFlag | Флаг переполнения | Состояние |
| 12 | IOPL | I/O PrivilegeLevel | Уровень приоритета ввода-вывода | Системный |
| 13 |
| 14 | NT | NestedTask | Флаг вложенности задач | Системный |
| 15 | 0 | | Зарезервирован |  |
| ***EFLAGS*** | | | | |
| 16 | RF | ResumeFlag | Флаг возобновления | Системный |
| 17 | VM | Virtual-8086 Mode | Режим виртуального процессора 8086 | Системный |
| 18 | AC | AlignmentCheck | Проверка выравнивания | Системный |
| 19 | VIF | VirtualInterruptFlag | Виртуальный флаг разрешения прерывания | Системный |
| 20 | VIP | VirtualInterruptPending | Ожидающее виртуальное прерывание | Системный |
| 21 | ID | ID Flag | Проверка на доступность инструкции CPUID | Системный |
| 22 |  | | Зарезервированы |  |
| **...** |  |
| 31 |  |

Значение флагов CF, DF и IF можно изменять напрямую в регистре флагов с помощью специальных инструкций (например, *CLD* для сброса флага направления), но нет инструкций, которые позволяют обратиться к регистру флагов как к обычному регистру. Однако можно сохранять регистр флагов в стек или регистр AH и восстанавливать регистр флагов из них с помощью инструкций *LAHF*, SAHF, PUSHF, *PUSHFD*, *POPF* и *POPFD*.

*Определение 3.7.10*

**Флаги состояния**

Флаги состояния (биты 0, 2, 4, 6, 7 и 11) отражают результат выполнения арифметических инструкций, таких как *ADD*, *SUB*, *MUL*, *DIV*.

* Флаг переноса CF устанавливается при переносе из старшего значащего бита/заёме в старший значащий бит и показывает наличие переполнения в беззнаковой целочисленной арифметике. Также используется в длинной арифметике.
* Флаг чётности PF устанавливается, если младший значащий байт результата содержит чётное число единичных битов. Изначально этот флаг был ориентирован на использование в коммуникационных программах: при передаче данных по линиям связи для контроля мог также передаваться бит чётности и инструкции для проверки флага чётности облегчали проверку целостности данных.
* Вспомогательный флаг переноса AF устанавливается при переносе из бита 3-го результата/заёме в 3-ий бит результата. Этот флаг ориентирован на использование в двоично-десятичной (binary coded decimal, BCD) арифметике.
* Флаг нуля ZF устанавливается, если результат равен нулю.
* Флаг знака SF равен значению старшего значащего бита результата, который является знаковым битом в знаковой арифметике.
* Флаг переполнения OF устанавливается, если целочисленный результат слишком длинный для размещения в целевом операнде (регистре или ячейке памяти). Этот флаг показывает наличие переполнения в знаковой целочисленной арифметике.

Из перечисленных флагов только флаг CF можно изменять напрямую с помощью инструкций *STC*, CLC и *CMC*.

Флаги состояния позволяют одной и той же арифметической инструкции выдавать результат трёх различных типов: беззнаковое, знаковое и двоично-десятичное (BCD) целое число. Если результат считать беззнаковым числом, то флаг CF показывает условие переполнения (перенос или заём), для знакового результата перенос или заём показывает флаг OF, а для BCD-результата перенос/заём показывает флаг AF. Флаг SF отражает знак знакового результата, флаг ZF отражает и беззнаковый, и знаковый нулевой результат.

В длинной целочисленной арифметике флаг CF используется совместно с инструкциями сложения с переносом (ADC) и вычитания с заёмом (SBB) для распространения переноса или заёма из одного вычисляемого разряда длинного числа в другой.

Инструкции условного перехода Jcc (переход по условию cc), SETcc (установить значение байта-результата в зависимости от условияcc), LOOPcc (организация цикла) и CMOVcc (условное копирование) используют один или несколько флагов состояния для проверки условия. Например, инструкция перехода JLE (jumpif lessor equal – переход, если «меньше или равно») проверяет условие «ZF = 1 или SF ≠ OF».

Флаг PF был введён для совместимости с другими микропроцессорными архитектурами и по прямому назначению используется редко. Более распространено его использование совместно с остальными флагами состояния в арифметике с плавающей запятой: инструкции сравнения (FCOM, FCOMP и т. п.) в математическом сопроцессоре устанавливают в нём флаги-условия C0, C1, C2 и C3, и эти флаги можно скопировать в регистр флагов. Для этого рекомендуется использовать инструкцию FSTSW AX для сохранения слова состояния сопроцессора в регистре AX и инструкцию SAHF для последующего копирования содержимого регистра AH в младшие 8 битов регистра флагов, при этом C0 попадает во флаг CF, C2 – в PF, а C3 – в ZF. Флаг C2 устанавливается, например, в случае несравнимых аргументов (NaN или неподдерживаемый формат) в инструкции сравнения FUCOM.

*Определение 3.7.11*

**Управляющий флаг**

**Флаг направления** DF (бит 10 в регистре флагов) управляет строковыми инструкциями (MOVS, CMPS, SCAS, LODS и STOS) – установка флага заставляет уменьшать адреса (обрабатывать строки от старших адресов к младшим), обнуление заставляет увеличивать адреса. Инструкции STD и CLD соответственно устанавливают и сбрасывают флаг DF.

*Определение 3.7.12*

**Системные флаги и поле IOPL**

Системные флаги и поле IOPL управляют операционной средой и не предназначены для использования в прикладных программах.

* Флаг разрешения прерываний IF – обнуление этого флага запрещает отвечать на маскируемые запросы на прерывание.
* Флаг трассировки TF – установка этого флага разрешает пошаговый режим отладки, когда после каждой выполненной инструкции происходит прерывание программы и вызов специального обработчика прерывания.
* Поле IOPL показывает уровень приоритета ввода-вывода исполняемой программы или задачи: чтобы программа или задача могла выполнять инструкции ввода-вывода или менять флаг IF, её текущий уровень приоритета (CPL) должен быть ≤ IOPL.
* Флаг вложенности задач NT – этот флаг устанавливается, когда текущая задача «вложена» в другую, прерванную задачу, и сегмент состояния TSS текущей задачи обеспечивает обратную связь с TSS предыдущей задачи. Флаг NT проверяется инструкцией IRET для определения типа возврата – межзадачного или внутризадачного.
* Флаг возобновления RF используется для маскирования ошибок отладки.
* VM – установка этого флага в защищённом режиме вызывает переключение в режим виртуального 8086.
* Флаг проверки выравнивания AC – установка этого флага вместе с битом AM в регистре CR0 включает контроль выравнивания операндов при обращениях к памяти: обращение к не выравненному операнду вызывает исключительную ситуацию.
* VIF – виртуальная копия флага IF; используется совместно с флагом VIP.
* VIP – устанавливается для указания наличия отложенного прерывания.
* ID – возможность программно изменить этот флаг в регистре флагов указывает на поддержку инструкции CPUID.

*Определение 3.7.13*

**Сегментные регистры**

Процессор имеет 6 так называемых сегментных регистров: CS, DS, SS, ES, FS и GS. Их существование обусловлено спецификой организации и использования оперативной памяти.

16-битные регистры могли адресовать только 64 Кб оперативной памяти, что явно недостаточно для более или менее приличной программы. Поэтому память программе выделялась в виде нескольких сегментов, которые имели размер 64 Кб. При этом абсолютные адреса были 20-битными, что позволяло адресовать уже 1 Мб оперативной памяти. Возникает вопрос – как имея 16-битные регистры хранить 20-битные адреса? Для решения этой задачи адрес разбивался на базу и смещение. База – это адрес начала сегмента, а смещение – это номер байта внутри сегмента. На адрес начала сегмента накладывалось ограничение – он должен был быть кратен 16. При этом последние 4 бита были равны 0 и не хранились, а подразумевались. Таким образом, получались две 16-битные части адреса. Для получения абсолютного адреса к базе добавлялись четыре нулевых бита, и полученное значение складывалось со смещением.

Сегментные регистры использовались для хранения адреса начала сегмента кода (CS – code segment), сегмента данных (DS – data segment) и сегмента стека (SS – stack segment). Регистры ES, FS и GS были добавлены позже. Существовало несколько моделей памяти, каждая из которых подразумевала выделение программе одного или нескольких сегментов кода и одного или нескольких сегментов данных: tiny, small, medium, compact, large и huge. Для команд языка ассемблера существовали определённые соглашения: адреса перехода сегментировались по регистру CS, обращения к данным сегментировались по регистру DS, а обращения к стеку – по регистру SS. Если программе выделялось несколько сегментов для кода или данных, то приходилось менять значения в регистрах CS и DS для обращения к другому сегменту. Существовали так называемые «ближние» и «дальние» переходы. Если команда, на которую надо совершить переход, находилась в том же сегменте, то для перехода достаточно было изменить только значение регистра IP. Такой переход назывался ближним. Если же команда, на которую надо совершить переход, находилась в другом сегменте, то для перехода необходимо было изменить как значение регистра CS, так и значение регистра IP. Такой переход назывался дальним и осуществлялся дольше.

32-битные регистры позволяют адресовать 4 Гб памяти, что уже достаточно для любой программы. Каждую Win32-программу Windows запускает в отдельном виртуальном пространстве. Это означает, что каждая Win32-программа будет иметь 4-х гигабайтовое адресное пространство, но вовсе не означает, что каждая программа имеет 4 Гб физической памяти, а только то, что программа может обращаться по любому адресу в этих пределах. А Windows сделает все необходимое, чтобы память, к которой программа обращается, «существовала». Конечно, программа должна придерживаться правил, установленных Windows, иначе возникает ошибка General Protection Fault.

Под архитектурой Win32 отпала необходимость в разделении адреса на базу и смещение, и необходимость в моделях памяти. На 32-битной архитектуре существует только одна модель памяти – flat (сплошная или плоская). Сегментные регистры остались, но используются по-другому[1](http://natalia.appmat.ru/C&C++/assembler.html#nota1). Раньше необходимо было связывать отдельные части программы с тем или иным сегментным регистром и сохранять/восстанавливать регистр DS при переходе к другому сегменту данных или явно сегментировать данные по другому регистру. При 32-битной архитектуре необходимость в этом отпала, и в простейшем случае про сегментные регистры можно забыть.

# 3.8. Директивы и выражения Ассемблера для управления вычислениями (ветвление, циклы, переходы, прерывания).

*Определение 3.8.1*

**Ветвление** — это место в программе, после которого в зависимости от какого-либо условия может начать выполняться тот или иной код.

*Определение 3.8.2*

**Безусловный переход** — это переход, который выполняется всегда. Безусловный переход осуществляется с помощью команды JMP. У этой команды один операнд, который может быть непосредственным адресом (меткой), регистром или ячейкой памяти, содержащей адрес. Существуют также «дальние» переходы — между сегментами.

Синтаксис: JMP <операнд>

Операнд указывает адрес перехода. Существует два способа указания этого адреса, соответственно различают **прямой** и **косвенный** переходы. Если в команде перехода указывается метка команды, на которую надо перейти, то переход называется **прямым**.

Синтаксис:

|  |
| --- |
| JMP L |
| **...** |
| L: mov eax, x |

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.

Синтаксис: JMP ebx

*Определение 3.8.3*

**Условный переход** осуществляется, если выполняется определённое условие. Если условие не выполняется, то управление переходит к следующей команде. Любая команда условного перехода начинается с символа j (от jump), а далее указывается конкретное условие, которое должно быть проанализировано (например, jcc). 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: CMP <операнд №1>, <операнд №2>.

|  |  |
| --- | --- |
| cmp al,5 | *Сравнение AL и 5* |
| jl c1 | *Переход, если AL < 5* |

Синтаксис:

Существует много команд для различных условных переходов. Также для некоторых команд есть синонимы (например, JZ и JE — это одно и то же). Для наглядности все команды условных переходов приведены в таблице 3.8.1:

Таблица 3.8.1

|  |  |
| --- | --- |
| **Команда** | **Переход, если** |
| JZ/JE | нуль или равно |
| JNZ/JNE | не нуль или не равно |
| JC/JNAE/JB | если ниже (для чисел без знака) |
| JNC/JAE/JNB | выше или равно (для чисел без знака) |
| JP | число единичных бит чётное |
| JNP | число единичных бит нечётное |
| JS | знак равен 1 |
| JNS | знак равен 0 |
| JO | есть переполнение |
| JNO | нет переполнения |
| JA/JNBE | выше (для чисел без знака) |
| JNA/JBE | ниже или равно (для чисел без знака) |
| JG/JNLE | больше |
| JGE/JNL | больше или равно |
| JL/JNGE | меньше |
| JLE/JNG | меньше или равно |
| JCXZ | содержимое CX равно нулю |

Команда LOOP позволяет организовать **цикл с известным числом повторений**:

|  |
| --- |
| mov ecx, n |
| L: **...** |
| **...** |
| loop L |

Команда LOOP требует, чтобы в качестве счётчика цикла использовался регистр ECX. Собственно, команда LOOP вычитает единицу именно из этого регистра, сравнивает полученное значение с нулём и осуществляет переход на указанную метку, если значение в регистре ECX больше 0.

**Команды LOOPE/LOOPZ и LOOPNE/LOOPNZ**

Эти команды похожи на команду LOOP, но позволяют также организовать и досрочный выход из цикла.

Синтаксис: LOOPE <метка>

Действие этой команды можно описать следующим образом: ECX = ECX - 1; if (ECX != 0 && ZF == 1) goto <метка>. До начала цикла в регистр ECX необходимо записать число повторений цикла. Команда LOOPE/LOOPZ, как и команда LOOP ставится в конце цикла, а перед ней помещается команда, которая меняет флаг ZF (обычно это команда сравнения CMP). Команда LOOPE/LOOPZ заставляет цикл повторяться ECX раз, но только если предыдущая команда фиксирует равенство сравниваемых величин (вырабатывает нулевой результат, т.е. ZF = 1).

Примеры циклов и их синтаксис:

|  |  |
| --- | --- |
| **Пример цикла** | mov cx, 10  mov dl, '0'  mov ah,02  cikl:  int 21h  inc dl  loop cikl |
| **Цикл с предусловием** | mov dl, '0'  mov ah, 02  cikl: cmp dl, ‘9’  ja end\_cikl  int 21h  inc dl  jmp cikl  end\_cikl: |
| **Цикл с постусловием** | mov dl, '0'  mov ah, 02  cikl:  ja end\_cikl  int 21h  inc dl  cmp dl, ‘9’  jbe cikl |

*Определение 3.8.4*

**Прерывание** — сигнал, сообщающий процессору о наступлении какого-либо события. При этом выполнение текущей последовательности команд приостанавливается, и управление передаётся обработчику прерывания, который реагирует на событие и обслуживает его, после чего возвращает управление в прерванный код. **Int** - это сокращение слова **Interrupt** что на русский переводиться как **прерывание**.

Синтаксис: Int <номер>

Как правило, номер прерывания записывается в виде шестнадцатеричного числа с суффиксом h. BIOS проверяет различные порты компьютера для определения и инициализации подключенных устройств. Затем BIOS создает в начале памяти (по адресу 0) таблицу прерываний, которая содержит адреса обработчиков прерываний, и выполняет две операции INT 11H (запрос списка присоединенного оборудования) и INT 12H (запрос размера физической памяти). Следующим шагом BIOS определяет имеется ли на диске или дискете операционная система DOS. Если обнаружена системная дискета, то BIOS выполняет прерывание INT 19H для доступа к первому сектору диска, содержащему блок начальной загрузки.

Внешние устройства передают сигнал внимания через контакт INTR в процессор. Процессор реагирует на этот запрос, если флаг прерывания IF установлен в 1 (прерывание разрешено), и (в большинстве случаев) игнорирует запрос, если флаг IF установлен в 0 (прерывание запрещено). Операнд в команде прерывания, например, INT 12H, содержит тип прерывания, который идентифицирует запрос. Для каждого типа система содержит адрес в таблице векторов прерываний, начинающейся по адресу 0000. Каждый элемент таблицы указывает на подпрограмму обработки указанного типа прерывания и содержит адрес кодового сегмента и смещение, которые при прерывании устанавливаются в регистры CS и IP соответственно.

Таблица 3.8.2 Адреса прерываний

|  |  |
| --- | --- |
| Адрес (шест.) | Функция прерываний (шест.) |
| 0-3 | Деление на нуль |
| 4-7 | Пошаговый режим (трассировка DEBUG) |
| 8-B | Немаскированное прерывание (NMI) |
| C-F | Точка останова (используется в DEBUG) |
| 10-13 | Переполнение регистра |
| 14-17 | Печать экрана |
| 18-1F | Зарезервировано |
| 20-23 | Сигнал от таймера |
| 24-27 | Сигнал от клавиатуры |
| 28-37 | Используются в компьютерах AT |
| 38-3B | Сигнал от дискетного дисковода |
| 3C-3F | Используется для принтера |
| 40-43 | Управление дисплеем |
| 44-47 | Запрос оборудования |
| 48-4B | Запрос размера памяти |
| 4C-4F | Дисковые операции ввода-вывода |
| 50-53 | Управление коммуникационным адаптером |
| 54-57 | Кассетные операции и спец. функции AT |
| 58-5B | Ввод с клавиатуры |
| 5C-5F | Вывод на принтер |
| 60-63 | Обращение к BASIC, встроенному в ROM |
| 64-67 | Перезапуск системы |
| 68-6B | Запрос и установка времени и даты |
| 6C-6F | Прерывание от клавиатуры |
| 70-73 | Прерывание от таймера |
| 74-77 | Адрес таблицы параметров дисплея |
| 78-7B | Адрес таблицы параметров дисковода |
| 7C-7F | Адрес таблицы графических символов |
| 80-83 | Нормальное завершение программы (DOS) |
| 84-87 | Обращение к функциям DOS |
| 88-8B | Адрес обработки завершения задачи (DOS) |
| 8C-8F | Адрес реакции по Ctrl/Break (DOS) |
| 90-93 | Адрес реакции на фатальную ошибку (DOS) |
| 94-97 | Абсолютное чтение с диска (DOS) |
| 98-9B | Абсолютная запись на диск (DOS) |
| 9A-9F | Создание резидентной программы (DOS) |
| A0-FF | Другие функции DOS |
| 100-1FF | Зарезервировано |
| 200-217 | Зарезервировано для BASIC |
| 218-3C3 | Используются BASIC-интерпретатором |
| 3C4-3FF | Зарезервировано |

Примечание:  
Прерывания 00-1F относятся к BIOS  
прерывания 20-FF относятся к DOS и BASIC.

Таблица 3.8.3 Основные прерывания BIOS.

|  |  |  |
| --- | --- | --- |
| INT 05H | Печать экрана. Выполняет вывод содержимого экрана на печатающее устройство. Команда INT 05H выполняет данную операцию из программы, а нажатие клавишей Ctrl/PrtSc - с клавиатуры. Операция запрещает прерывания и сохраняет позицию курсора. | |
| INT 10H | Управление дисплеем. Обеспечивает экранные и клавиатурные операции. | |
| INT 11H | Запрос списка присоединенного оборудования. Определяет наличие различных устройств в системе, результирующее значение возвращает в регистре AX. При включении компьютера система выполняет эту операцию и сохраняет содержимое AX в памяти по адресу шест.410. Значения битов в регистре AX: | |
| Бит | Устройство |
| 15,14 | Число подключенных принтеров. |
| 13 | Последовательный принтер. |
| 12 | Игровой адаптер. |
| 11-9 | Число последовательных адаптеров стыка RS232. |
| 7,6 | Число дискетных дисководов, при бите 0=1: 00=1, 01=2, 10=3 и 11=4. |
| 5,4 | Начальный видео режим: 00 = не используется, 01 = 40х25 плюс цвет, 10 = 80х25 плюс цвет,11 = 80х25 черно-белый режим. |
| 1 | Значение 1 говорит о наличии сопроцессора. |
| 0 | Значение 1 говорит о наличии одного или более дисковых устройств, и загрузка операционной системы должна осуществляться с диска. |
| INT 12H | Запрос размера физической памяти. Возвращает в регистре AX размер памяти в килобайтах шест.200 соответствует памяти в 512 К. Данная операция полезна для выравнивания размера программы в соответствии с доступной памятью. | |
| INT 13H | Дисковые операции ввода - вывода. Обеспечивает операции ввода-вывода для дискет и винчестера. | |
| INT 14H | Управление коммуникационным адаптером. Обеспечивает последовательный ввод-вывод через коммуникационный порт RS232. Регистр DX должен содержать номер (0 или 1) адаптера стыка RS232. Четыре типа операции, определяемые регистром AH, выполняют прием и передачу символов и возвращают в регистре AX байт состояния коммуникационного порта. | |
| INT 15H | Кассетные операции ввода - вывода и специальные функции для компьютеров AT. Обеспечивает операции ввода-вывода для кассетного магнитофона, а также расширенные операции для компьютеров AT. | |
| INT 16H | Ввод с клавиатуры. Обеспечивает три типа команд ввода с клавиатуры. | |
| INT 17H | Вывод на принтер. Обеспечивает вывод данных на печатающее устройство. | |
| INT 18H | Обращение к BASIC, встроенному в ROM. Вызывает BASIC-интерпретатор, находящийся в постоянной памяти ROM. | |
| INT 19H | Перезапуск системы. Данная операция при доступном диске считывает сектор 1 с дорожки 0 в область начальной загрузки в памяти (сегмент 0, смещение 7C00) и передает управление по этому адресу. Если дисковод не доступен, то операция передает управление через INT 18H в ROM BASIC. Данная операция не очищает экран и не инициализирует данные в ROM BASIC, поэтому ее можно использовать из программы. | |
| INT 1AH | Запрос и установка текущего времени и даты. Считывает и записывает показание часов в соответствии со значением в регистре AH. Для определения продолжительности выполнения программы можно перед началом выполнения установить часы в 0, а после считать текущее время. Отсчет времени идет примерно 18,2 раза в секунду. Значение в регистре AH соответствует следующим операциям:  AH=00 Запрос времени. В регистре CX устанавливается старшая часть значения, а в регистре DX - младшая. Если после последнего запроса прошло 24 часа, то в регистре AL будет не нулевое значение.  AH=01 Установка времени. Время устанавливается по регистрам CX (старшая часть значения) и DX (младшая часть значения).  Коды 02 и 06 управляют временем и датой. | |
| INT 1FH | Адрес таблицы графических символов. В графическом режиме имеется доступ к символам с кодами 128-255 в 1К таблице, содержащей по восемь байт на каждый символ. Прямой доступ в графическом режиме обеспечивается только к первым 128 ASCII-символам (от 0 до 127). | |

Таблица 3.8.4 ПРЕРЫВАНИЯ DOS

|  |  |
| --- | --- |
| INT 20H | Завершение программы. Запрос завершает выполнение программы и передает управление в DOS. Данный запрос обычно находится в основной процедуре. |
| INT 21H | Запрос функций DOS. Основная операция DOS, вызывающая определенную функцию в соответствии с кодом в регистре AH. Назначение функций DOS описано в следующем разделе. |
| INT 22H | Адрес подпрограммы обработки завершения задачи. |
| INT 23H | Адрес подпрограммы реакции на Ctrl/Break. |
| INT 24H | Адрес подпрограммы реакции на фатальную ошибку. В этом элементе и в двух предыдущих содержатся адреса, которые инициализируются системой в префиксе программного сегмента и, которые можно изменить для своих целей. |
| INT 25H | Абсолютное чтение с диска. |
| INT 26H | Абсолютная запись на диск. |
| INT 27H | Завершение программы, оставляющее ее резидентной. Позволяет сохранить COM-программу в памяти. |

# 3.9. Алгоритм ассемблирования, внутренние таблицы, выходные файлы.

Процесс трансляции программы на языке ассемблера в объектный код принято называть ассемблированием. В языке ассемблера каждой символьной команде соответствует одна машинная инструкция. Процесс ассемблирования осуществляется за два прохода по программе. Главная цель первого прохода составить таблицу символических имен. Во время второго прохода на основании таблицы символических имен и таблицы кодов операций формируется объектный файл, содержащий машинный код программы. Выходными файлами ассемблирования являются **объектные файлы.**

Объектные файлы представляют собой блоки машинного кода и данных, с неопределенными адресами ссылок на данные и процедуры в других объектных модулях, а также список своих процедур и данных. Компоновщик собирает код и данные каждого объектного модуля в итоговую программу, вычисляет и заполняет адреса перекрестных ссылок между модулями. Связывание со статическими библиотеками выполняется редактором связей или компоновщиком (который может представлять собой отдельную программу или быть частью компилятора), а с операционной системой и динамическими библиотеками связывание выполняется при исполнении программы после её загрузки в память.

*Определение 3.9.1*

**Таблица символических имен** — содержит по одной записи для каждого символического имени(метки), определённого в программе. Каждая строчка таблицы содержит

* символическое имя
* его численное значение
* дополнительную информацию (длину поля данных, связанного с именем; биты перераспределения памяти; можно ли получить доступ к символическому имени извне процедуры). Например,

|  |  |  |
| --- | --- | --- |
| MARK | 111 | Прочая информация |

*Определение 3.9.2*

Ассемблер создаёт таблицу имён отслеживая адрес следующего доступного байта памяти, который хранится в переменной, называемой **счётчиком адресов**. Изначально счетчик адресов равен 0, и увеличивается после обработки каждой команды на ее длину, которая хранится в таблице кодов операций.

*Определение 3.9.3*

**Таблица кодов операций** — является встроенной в конкретный ассемблер константной таблицей. В каждой записи содержится

* символьный код операции
* первый операнд
* второй операнд
* числовое значение кода операции (шестнадцатеричное число)
* длина команды
* номер типа операции - идентифицирует процедуру, которая вызывается для всех команд данного типа, например

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| Символьный код операции | первый операнд | второй операнд | числовое значение кода операции | длина команды | номер типа операции |
| ADD | EAX | immed32 | 0x05 | 5 | 4 |
| ADD | reg | reg | 0x01 | 2 | 19 |
| AND | EAX | immed32 | 0x25 | 5 | 4 |

*Определение 3.9.4*

**Дизассемблирование** — преобразование машинного кода, объектного файла или библиотечных модулей в текст [программы](https://ru.wikipedia.org/wiki/%D0%9A%D0%BE%D0%BC%D0%BF%D1%8C%D1%8E%D1%82%D0%B5%D1%80%D0%BD%D0%B0%D1%8F_%D0%BF%D1%80%D0%BE%D0%B3%D1%80%D0%B0%D0%BC%D0%BC%D0%B0) на языке ассемблера.

Чаще всего дизассемблирование используют для анализа программы (или ее части), исходный текст которой неизвестен — с целью модификации, копирования или взлома. Реже — для поиска ошибок (багов) в программах и компиляторах, а также для анализа и оптимизации создаваемого компилятором машинного кода.

*Определение 3.9.5*

**Обратная разработка** (**обратный инжиниринг**, **реверс-инжиниринг**) — исследование некоторого готового устройства или программы, а также документации на него с целью понять принцип его работы; например, чтобы обнаружить недокументированные возможности (в том числе программные закладки), сделать изменение, или воспроизвести устройство, программу или иной объект с аналогичными функциями, но без копирования как такового.

Защита программ от дизассемблирования.

Универсальным методом противодействия дизассемблированию программы является шифрование. Очевидно, что дизассемблирование зашифрованного кода бесполезно. Применяя шифрование кода программы для противодействия дизассемблированию, следует учитывать распространенные ошибки реализации данного метода.

* Неэффективной является такая реализация, когда исполняемый код в полном объеме и однократно шифруется/дешифруется (так как легко найти момент после дешифрования).
* Необходимо осуществить выбор эффективного ключа и, если необходимо, надежно хранить ключ.
* Следует учесть, что для защиты программ от дизассемблирования, не рекомендуется использование симметричных криптографических алгоритмов.

Так же интересный прием защиты от дизассемблирования — использование нестандартной структуры программы. В этом случае дизассемблер «не поймет» нестандартную сегментацию программы.

# 3.10. Загрузчики и компоновщики. Форматы объектных модулей. Оверлейные загрузчики и оверлеи.

*Определение 3.10.1*

**Загрузчик** - программа, отвечающая за загрузку исполнимых файлов и запуск соответствующих новых процессов. Обычно является частью операционной системы, но может быть и самостоятельной программой, к примеру, позволяющей операционной системе запускать программы, скомпилированные для других операционных систем.

При запуске новой программы загрузчик должен:

* Считать данные из запускаемого файла.
* Если необходимо — загрузить в память недостающие динамические библиотеки.
* Заменить в коде новой программы относительные адреса и символические ссылки на точные, с учётом текущего размещения в памяти, то есть выполнить связывание адресов.
* Создать в памяти образ нового процесса и запланировать его к исполнению.

Загрузчик операционной системы действует по схожему принципу, но обычно является отдельной программой, поскольку решает специфическую задачу — запуск самой операционной системы.

*Определение 3.10.2*

**Компоновщик** (также **редактор связей** или **линкер**) — это инструментальная программа, которая производит **компоновку** (*«***линковку***»*): принимает на вход один или несколько объектных модулей и собирает по ним исполнимый модуль.

Для связывания модулей компоновщик использует таблицы символов, созданные компилятором в каждом из объектных модулей.

*Определение 3.11.3*

**Таблица символов** — это структура данных, используемая транслятором (компилятором или интерпретатором), в которой каждый идентификатор переменной или функции из исходного кода ассоциируется с информацией, связанной с его объявлением или появлением в коде: типом данных, областью видимости и в некоторых случаях местом в памяти.

Эти таблицы могут содержать символы следующих типов:

* Определённые или экспортируемые имена — функции и переменные, определённые в данном модуле и предоставляемые для использования другим модулям;
* Неопределённые или импортируемые имена — функции и переменные, на которые ссылается модуль, но не определяет их внутри себя;
* Локальные — могут использоваться внутри объектного файла для упрощения процесса настройки адресов.

Для большинства компиляторов, один объектный файл является результатом компиляции одного файла с исходным кодом. Если программа собирается из нескольких объектных файлов, компоновщик собирает эти файлы в единый исполнимый модуль, вычисляя и подставляя адреса вместо символов, в течение времени компоновки(статическая компоновка) или во время исполнения (динамическая компоновка).

Компоновщик может извлекать объектные файлы из специальных коллекций, называемых библиотеками. Если не все символы, на которые ссылаются пользовательские объектные файлы, определены, то компоновщик ищет их определения в библиотеках, которые пользователь подал ему на вход. Обычно, одна или несколько системных библиотек используются компоновщиком по умолчанию. Когда объектный файл, в котором содержится определение какого-либо искомого символа, найден, компоновщик может включить его в исполнимый модуль (в случае статической компоновки) или отложить это до момента запуска программы (в случае динамической компоновки).

Работа компоновщика заключается в том, чтобы в каждом модуле определить и связать ссылки на неопределённые имена. Для каждого импортируемого имени находится его определение в других модулях, упоминание имени заменяется на его адрес.

Компоновщик обычно не выполняет проверку типов и количества параметров процедур и функций. Если надо объединить объектные модули программ, написанные на языках со строгой типизацией, то необходимые проверки должны быть выполнены дополнительной утилитой перед запуском редактора связей.

*Определение 3.10.4*

**Объектный модуль** - файл с промежуточным представлением отдельного модуля программы, полученный в результате обработки исходного кода компилятором. Объектный файл содержит в себе особым образом подготовленный код (часто называемый двоичным или бинарным), который может быть объединён с другими объектными файлами при помощи компоновщика для получения готового исполнимого модуля, либо библиотеки.

*Определение 3.10.5*

**Исполнимый (исполняемый) модуль, исполняемый файл** — файл, содержащий программу в виде, в котором она может быть исполнена компьютером. Перед исполнением программа загружается в память, и выполняются некоторые подготовительные операции.

Объектные файлы представляют собой блоки машинного кода и данных, с неопределенными адресами ссылок на данные и процедуры в других объектных модулях, а также список своих процедур и данных. Компоновщик собирает код и данные каждого объектного модуля в итоговую программу, вычисляет и заполняет адреса перекрестных ссылок между модулями. Связывание со статическими библиотеками выполняется редактором связей или компоновщиком (который может представлять собой отдельную программу или быть частью компилятора), а с операционной системой и динамическими библиотеками связывание выполняется при исполнении программы после её загрузки в память.

**Форматы объектных модулей.** Файл объектного модуля состоит из отдельных секций данных, каждая из которых содержит определенный тип данных:

* Заголовок (описание файла и информация необходимая компоновщику)
* Сегмент кода (исполнимый код)
* Сегмент данных (инициализация переменных)
* Сегмент констант (инициализация констант, описание глобальных переменных)
* Внешние определения и ссылки на связи
* Информация о динамических связях
* Отладочная информация

*Определение 3.10.6*

**Overlay** — метод программирования, позволяющий создавать программы, занимающие больше памяти, чем установлено в системе.

Метод предполагает разделение программы на фрагменты, называемые оверлеями (overlays). Размер каждого оверлея ограничен, согласно размеру доступной памяти. Место в памяти, куда будет загружен оверлей называется регионом (region, destination region). Хотя часто программы используют только один блок памяти для загрузки различных оверлеев, возможно определение нескольких регионов различного размера. Загрузчик оверлеев, иногда являющийся частью ОС, подгружает запрашиваемый оверлей из внешней памяти в регион, обеспечивает все необходимые связи, например, с библиотеками.

# Предметный указатель

A

ALU *См.* АЛУ

ASIP 49, 60

AVX 21

B

BIOS 25, 93

broadcast-шина 75

D

DRAM 66, 67, 103

E

EPIC 57

F

FIFO 39

FPU 42

L

LPM 105

M

MIMD 16

MISD 16

multicast-шина 75

N

NE555 51

O

overlay 140

R

RISC 54, 55

RTE 26, 27

S

SIMD 56, 57

SISD 56

SRAM 103, 104, 105, 106, 108

stall 98

V

VLIW 54, 55, 56, 57

volatile 75

А

АВМ 3

адресные регистры 34

АЗУ *См.* ассоциативное запоминающее устройство

АЛУ 4, 8, 37, 38, 39, 48, 59

АП *См.*ассоциативная память

аппаратное обеспечение, аппаратные средства 31

аппаратный порт 50

арбитр 48

арифметические операции 112

арифметические сдвиги 114

архитектура SISD 56

архитектура вычислительного средства 8

архитектура компьютера 4

архитектура набора команд 59

асинхронный режим 48, 64

ассемблеры 78

ассоциативная память 39

ассоциативноезапоминающее устройство 39

АЦП 52

Б

безусловный переход 124

библиотеки подпрограмм 79

блокировки 78

В

ветвление 124

внешняя симуляция 78

внутренняя симуляция 78

временная избыточность 15

временная многопоточность 45

встраиваемые операционные системы 26

Г

гарвардская архитектура 5, 6

ГВМ 3

гибернация 10

гибкие архитектуры 16

гибридная архитектура 14

грид-вычисления 74

Д

дизассемблирование 136

директива .CSEQ 101

директива .DSEQ 101

директива .ORG 102

директива BYTE 101

директива DB 102

директива DW 103

директива ESEG 102

директивы 91

Ж

жесткий процессор 49

З

загрузчик 137, 138

заказной микропроцессор 60

звезда-кольцо 70

зеркалированние памяти 75

И

избыточность пространства 15

индексный регистр 35

исполнимый (исполняемый) модуль 139

исполняемый файл 139

К

к.-с. р. *См.* кросс-средства разработки

коды условий См**.** флаги

команда CALL 107

команда LD 105

команда LDS 106

команда LPM 106

команда MOV 107

команда RET 107

команда STS 108

команда сдвига 114

комбинированная шина 63

комбинированные топологии 69

компоновщик 135, 138, 139, 140

компоновщики 79

компьютер общего назначения 28

конвейерная архитектура 14

контроллер 36, 52

косвенный переход 125

критическая секция 76

кросс-средства разработки 30

куча 110

кэш-память 9, 47

Л

линкер 138

логические команды 113

логические сдвиги 114

М

математический сопроцессор 40

микроархитектура 59

микрокод 61

микрокомпьютеры 6

микроконтроллеры 6

микросхемы обвязки процессоров 50

миникомпьютеры 7

многопортовая память 74

многопоточность 45

многоядерный процессор 44

модуль 85

мэйнфреймы 7

О

обвязка 50

обратная разработка 136

обратный инжиниринг 136

объектные файлы 134

Объектные файлы 140

объектный модуль 139

одновременная многопоточность 46

Одновременнаямногопоточность 45

однокристальная система 49

ОЗУ 9, 42, 43, 44, 47, 74, 75

оперативная память 9

операционные системы реального времени 26

оптимизация 94, 95

ОСРВ 26, 27

отказоустойчивость 15

отладчик 79

очередь 39

П

пакетный режим 66

параллельная передача 62

параллельные вычисления 73

периферийные устройства 10

персональные компьютеры 6

ПЗУ 41, 44, 99, 101, 104

плезиосинхронная передача 65

поле IOPL 121

поразрядные операции 113

порты 9

постоянноезапоминающее устройство 41

препроцессоры исходных текстов 79

прерывание 76, 129

прикладное программное обеспечение 29

принстонская архитектура 5

программируемая пользователем вентильная матрица 60

профилирование 79

процессор 8, 12, 18, 43, 54, 59, 60, 76

прямой код 19

прямой переход 125

Р

рабочие станции 6

разряд кода 19

разрядность процессора 18

распределенная система 73

распределённые вычисления 73

расширенные 115

реверс-инжиниринг 136

регистр стека 35

регистр флагов *См.* флаг

регистры 32, 33, 34, 37, 115

регистры общего назначения 116

редактор связей 138

резервирование 15

С

северный мост 51

сегментные регистры 35, 122

семафор 76

серверы 7

синхронный режим 48, 65

Система на кристалле *См.однокристальная система*

системные флаги 121

сопроцессор 9

сопряжение 78

специализированная вычислительная машина 29

специализированные вычислители 12, 87

специализированные редакторы исходных текстов 79

специализированный процессор 13

стек 99

стековая память 99

суперкомпьютеры 7, 28

счетчик адресов 135

счетчик команд 33, 43

Т

таблица кодов операций 135

таблица символических имен 135

таблица символов 138

такты ожидания 97

текстовые редакторы 79

топология "звезда" 68, 69

топология "кольцо" 69

топология "шина" 67

точка останова 80

трансляторы 79

У

указатель команд 117

универсальная машина Тьюринга 28

управляющий флаг 121

условный переход 125

УУ 4, 8, 39

Ф

флаг 34

флаги состояния 119

фон Неймана 5, 6

Ц

ЦАП 52

ЦВМ 3

центральный процессор 38

циклические сдвиги 115

цифровая последовательная передача 62

цифровой сигнальный процессор 13

ЦП 34, 38, 54, 59

ЦСП 13

Ч

чипсет 50

Ш

шина 5, 9, 10, 14, 18, 36, 63, 67, 69

шина адреса 36

Э

ЭВМ *См.* электронная вычислительная машина

электронная вычислительная машина 3

энергонезависимая память 9

Ю

южный мост 51

Я

ядро процессора 41