Поволжский государственный университет
телекоммуникаций и информатики

Факультет ИСТ
Кафедра ИВТ

Акчурин Э.А.

Машинно-зависимые языки программирования

Учебное пособие для студентов направлений

230100 «Информатика и вычислительная техника»
220400 «Управление в технических системах»
231000 «Программная инженерия»

Самара
2011

Факультет информационных систем и технологий

Кафедра «Информатика и вычислительная техника»

Автор - д.т.н., профессор Акчурин Э.А.

Другие материалы по дисциплине Вы найдете на сайте

www.ivt.psati.ru

Оглавление

Введение 6
1. Архитектура ЭВМ 8
1.1. Структура МПС
1.2. Основные понятия в архитектуре МПС
1.3. Архитектура фон Неймана
1.4. Гарвардская архитектура
1.5. Параллельная архитектура
1.6. Конвейерная архитектура
1.7. Суперскалярная архитектура
1.8. Архитектура VLIW
1.9. Архитектуры CISC, RISC
1.10. Ассемблеры
1.11. Сообщения Windows
1.12. Версии ассемблеров
1.13. Среды разработки
2. Представление данных в ЭВМ 50
2.1. Системы счисления и преобразования между ними
2.2. Форматы представления чисел
2.3. Типы адресаций операндов
2.4. Интерфейсы
3. Архитектура CISC от Intel 75
3.1. Введение
3.2. Микроархитектура Intel
3.3. Программная модель IA-32
3.4. Целочисленный процессор
3.5. Математический сопроцессор
3.6. MMX-технология
3.7. XMM технология
3.8. Система команд
3.9. Цикл трансляции, компоновки и выполнения
3.10. Ассемблер CISC
3.11. Описание MASM
3.12. Структура программы на ассемблере
3.13. Типы данных
3.14. Макросредства
3.15. Директивы
4. Архитектура RISC 314

5. Архитектура VLIW 316
5.1. Архитектура вычислительных систем со сверхдлинными командами
5.2. Архитектура IA-64
5.3. Itanium
6. Многоядерные архитектуры 326
7. Микроконтроллер AVR от Atmel 329
7.1. Архитектура AVR от Atmel
7.2. Ассемблер
7.3. ИСР AVR Studio
8. Микроконтроллеры C28x 367
8.1. Архитектура C28x
8.2. Архитектура F28x
8.3. Инструментальные средства разработки ПО
8.4. Ассемблер
8.5. Команды ассемблера
8.6. Листинги программ
8.7. Формат объектного файла
8.8. Директивы ассемблера
8.9. Макроязык и макрокоманды
8.10. Компоновщик
8.11. Архиватор
8.12. Абсолютный листер
8.13. Листер перекрестных ссылок
8.14. Утилита 16-ричного преобразования
8.15. Согласование заголовочных C/C++ файлов с ассемблером
8.16. ИСР Code Composer Studio (CCS)
9. TMS320C6000 446
9.1. Архитектура VelociTI
9.2. Структура и состав ЦСП C6x
9.3. Средства разработки ЦСП C6x
9.4. Ассемблер ЦСП C6x
9.5. Команды ассемблера
9.6. Константы
9.7. Выражения
9.8. Листинги
9.9. Листинги программ
9.10. Директивы ассемблера
9.11. Макроязык и макрокоманды
9.12. Компоновщик

9.13. Утилиты
10. Поддержка в MATLAB.....491
10.1. Введение
10.2. Встроенные платы для ЦСП ‘C6x

Введение

Целью преподавания дисциплины является изучение машинно-зависимых языков программирования (Ассемблеров). В курсе изучаются:

- Архитектуры процессоров.
- Принципы построения языка Ассемблера.
- Ассемблеры разного типа.
- Интегрированные среды разработки, поддерживающие работу на Ассемблере.

Дисциплина относится к циклу профессиональных компонент основной образовательной программы (ООП).

Изучение данной дисциплины базируется на следующих дисциплинах:

- ЭВМ и периферийные устройства.
- Математическая логика и теория алгоритмов.

Основные положения дисциплины должны быть использованы в дальнейшем при изучении следующих дисциплин:

- Теория автоматов и формальных языков.
- Управление сложными техническими системами.

Студенты, успешно выполнившие учебный план, **должны знать**:

- Основы построения и архитектуры ЭВМ.
- Знать основы современных языков Ассемблера.
- Знать Ассемблер микроконтроллеров Atmel.
- Знать Ассемблер микроконтроллеров C2x от Texas Instruments.
- Знать Ассемблер цифрового сигнального процессора C6x от Texas Instruments.

Студенты, успешно выполнившие учебный план, **должны уметь**:

- Выбирать, создавать и отлаживать программно-аппаратные средства
- Программировать на зыке Ассемблера Atmel в интегрированной среде разработки (ИСР).
- Программировать на зыке Ассемблера C2x в ИСР.
- Программировать на зыке Ассемблера C6x в симуляторе C6xTools.

Рекомендуемая литература

1. Юров В.И. Assembler. Практикум. 2-е изд. – СПб. – Питер, 2006. – 399с.
2. Трамперт В. AVR-RISC микроконтроллеры. Пер. с нем. – К.: “МК-Пресс”, 2006. – 464с.
3. Хартов В.Я. Микроконтроллеры AVR. Практикум для начинающих. - М.: Изд-во МГТУ им. Н.Э. Баумана, 2007. – 240с.
4. Солонина А.И. и др. Алгоритмы и процессоры цифровой обработки сигналов.– СПб.: БХВ-Петербург, 2001, 464 с.
5. Корнеев В.В., Киселев А.В. Современные микропроцессоры. СпБ.:БХВ-Петербург, 2003, 448с.
6. Магда Ю. Ассемблер для процессоров Intel Pentium. – СПб. – Питер, 2006. – 410с.
7. Калашников О.А. Ассемблер? Это просто! Учимся программировать. – Спб.: БХВ=Петербург, 2006. – 384с.
8. Иванова В.Г., Тяжев А.И. Цифровая обработка сигналов и сигнальные процессоры. Самара: ООО Офорт, 2008, 262 с.

Программное обеспечение для выполнения лабораторных работ:

- ИСР AVR Studio.
- ИСР CCS для C2x.
- Инструментарий C6xTools.

1. Архитектура ЭВМ

Процессором называют программно-управляемое устройство, осуществляющее процесс обработки информации и управление им. Первые процессоры строились с использованием элементной базы общего назначения. В настоящее время процессоры используют специализированные большие или сверхбольшие интегральные схемы (БИС/СБИС),

Микропроцессром (МП) называют построенное на одной или нескольких БИС/СБИС программно-управляемое устройство, осуществляющее процесс обработки информации и управление им. МП появились, когда уровень интеграции ИС достиг значений, при которых необходимые для программной реализации алгоритмов блоки удалось разместить на одном или нескольких кристаллах. В настоящее время понятия процессор и МП эквивалентны.

МП система (МПС) – совокупность МП, памяти и устройства ввода/вывода (внешние устройства). Решаемая задача определяется реализуемой МП программой, структура МПС остается неизменной, что и определяет ее универсальность.

Совокупность БИС/СБИС, пригодных для совместного применения в МПС, называют МП комплектом (МПК). Понятие МПК задает номенклатуру микросхем с точки зрения возможностей их совместного применения (совместимость по архитектуре, электрическим параметрам, конструктивным признакам и др.). В состав МПК могут входить микросхемы различных серий и схемотехнологических типов при условии их совместимости.

1.1. Структура МПС

Практически всегда структура МПС является магистрально-модульной. В такой структуре имеется группа магистралей (шин), к которым подключаются различные модули (блоки), обменивающиеся между собой информацией поочередно, в режиме разделения времени.

Термин "шины" относится к совокупности цепей (линий), число которых определяет разрядность шины.

Типична 3-шинная структура МПС с шинами адресов ША, данных ШД и управления ШУ. Наряду с русскими терминами применяются английские АВ (Address Bus), DB (Data Bus) и СВ (Control Bus).

Структура МПС с простым МП от Intel, который имеет мультиплексируемую шину адресов/данных.

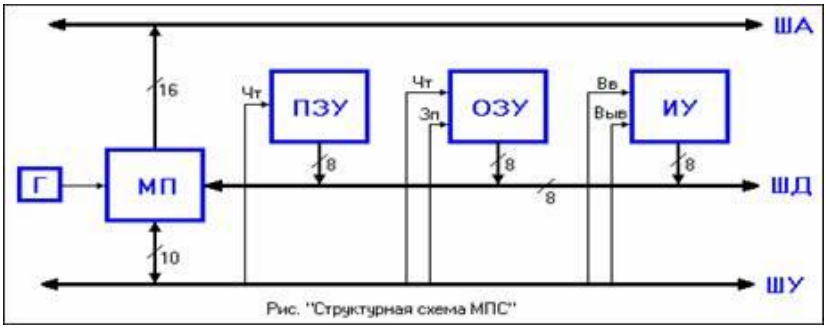


Рис. "Структурная схема МПС"

После передачи младшего байта адреса шина $AD_{7\text{-}0}$ отдается для передачи данных. Эти передачи двунаправлены, направление задается сигналом RD.. При активном состоянии сигнала чтения RD (Read) данные передаются справа налево, при пассивном - в обратном направлении. К шине данных подключены информационные выводы всех модулей МПС.

Микропроцессор МП. Выполняя программу, МП обрабатывает команду за командой. Команда задает выполняемую операцию и содержит сведения об участвующих в ней операндах. После приема команды происходит ее расшифровка и выполнение, в ходе которого МП получает необходимые данные из памяти или внешних устройств. Ячейки памяти и внешние устройства (порты) имеют номера, называемые адресами, которыми они обозначаются в программе.

Генкератор Г задает МП тактовые импульсы. По каждому МП выполняет команду.

Однонаправленная адресная шина ША. По ней МП посылает адреса, определяя объект, с которым будет обмен.

Ддвунаправленная шина данных ШД. По ней МП обменивается данными с модулями (блоками) системы.

Шина управления ШУ. По ней идет обмен управляющей информацией.

Постоянное запоминающее устройство - ПЗУ (ROM – Read Only Memory) хранит фиксированные программы и данные, оно является энергонезависимым и при выключении питания информацию не теряет.

Оперативное запоминающее устройство - ОЗУ (RAM - Random Access Memory) хранит оперативные данные (изменяемые программы, промежуточные результаты вычислений и др.), является энергозависимым и теряет информа-

цию при выключении питания. Для приведения системы в работоспособное состояние после включения питания ОЗУ следует загрузить необходимой информацией.

Интерфейс управления ИУ. Осуществляет взаимодействие с устройствами ввода-вывода (УВВ) или внешними устройствами (ВУ) - техническими средствами для передачи данных извне в МП или память либо из МП или памяти во внешнюю среду. Для подключения ВУ необходимо привести их сигналы, форматы слов, скорость передачи и т. п. к стандартному виду, воспринимаемому данным МП. Это выполняется специальными блоками, называемыми адаптерами (интерфейсными блоками ввода-вывода). Интерфейсом называют совокупность аппаратных и программных средств, унифицирующих процессы обмена между модулями системы.

Кроме обозначенных блоков, в состав систем входят обычно и более сложные, чем адаптеры, блоки управления внешними устройствами - контроллеры. К их числу относятся контроллеры прерываний, прямого доступа к памяти,оллеры клавиатуры, дисплея, дисковой памяти и т. д.

Контроллеры прерываний обеспечивают обмен с внешними устройствами в режиме прерывания (временной остановки) выполняемой программы для обслуживания запроса от внешнего устройства.

Контроллеры прямого доступа к памяти обслуживают режим прямой связи между внешними устройствами и памятью без участия МП. При управлении обменом со стороны МП пересылка данных между внешними устройствами и памятью происходит в два этапа — сначала данные принимаются МПом, а затем выдаются им на приемник данных. В режиме прямого доступа к памяти МП отключается от шин системы и передает управление ими контроллеру прямого доступа, а передачи данных осуществляются в один этап — непосредственно от источника к приемнику.

В состав МПС часто входят также программируемые таймеры, формирующие различные сигналы (интервалы, последовательности импульсов и т. д.) для проведения операций, связанных со временем.

В современных МП системах используются наборы микросхем (Chip sets), которые соединяют компонеты:

- Северный мост для связи МП с памятью.
- Южный мост для ввода-вывода.

1.2. Основные понятия в архитектуре МПС

Архитектура вычислительной машины — концептуальная структура вычислительной машины, определяющая проведение обработки информации и включающая методы преобразования информации в данные и принципы взаимодействия технических средств и программного обеспечения. В более подробное описание, определяющее конкретную архитектуру, также входят:

- структурная схема ЭВМ,
- средства и способы доступа к элементам этой структурной схемы,
- организация и разрядность интерфейсов ЭВМ,
- набор и доступность регистров,
- организация памяти и способы её адресации,
- набор и формат машинных команд процессора,
- способы представления и форматы данных, правила обработки прерываний.

По перечисленным признакам и их сочетаниям среди архитектур выделяют:

- По разрядности интерфейсов и машинных слов: 8-, 16-, 32-, 64-, 128- разрядные (ряд ЭВМ имеет и иные разрядности).
- По особенностям набора регистров, формата команд и данных: CISC, RISC, VLIW;
- По количеству центральных процессоров: однопроцессорные, многопроцессорные, суперскалярные.

Многопроцессорные по принципу взаимодействия с памятью делят на:

- симметричные многопроцессорные (SMP),
- массивно-параллельные (MPP),
- распределенные.

Аппаратная платформа включает:

- **АСК - Архитектура системы команд** (ISA - instruction set architecture). АСК — это приблизительно то же самое, что и модель программирования, с точки зрения программиста на языке ассемблера или создателя компилятора.
- **Микропрограмма** (firmware - микрокод). Это системное программное обеспечение, встроенное («зашитое») в аппаратное устройство, и хранящееся в его энергонезависимой памяти ПЗУ.
- **Микроархитектура** (иногда сокращаемая до μarch или uarch) — это способ, которым данная архитектура набора команд реализована в процессоре. Каждая архитектура может быть реализована с помощью различных микроархитектур. Реализации могут варьироваться в зависимости от целей данного дизайна или в результате изменений в технологиях. Архитектура компьютера является комбинацией микроархитектуры, микрокода и архитектуры.

Центральный процеессор ЦП (или центральное процессорное устройство — ЦП; central processing unit - CPU) - микросхема, исполнитель машинных инст-

рукций (кода программ), главная часть аппаратного обеспечения компьютера. Иногда называют МП или просто процессором. На компьютерном сленге его называют либо "проц", либо "камень".

1.3. Архитектура фон Неймана

Большинство современных процессоров основаны на той или иной версии циклического процесса последовательной обработки данных, изобретённого Джоном фон Нейманом в 1946.

Отличительной особенностью архитектуры фон Неймана (иначе принстонской) является то, что инструкции и данные хранятся в одной и той же памяти по разным адресам.

Важнейшие этапы процесса работы МП приведены ниже. В различных архитектурах и для различных команд могут потребоваться дополнительные этапы. Например, для арифметических команд могут потребоваться дополнительные обращения к памяти, во время которых производится считывание операндов и запись результатов.

Этапы цикла выполнения:

- МП выставляет число, хранящееся в регистре счётчика команд, на шину адреса ША и по шине управления ШУ отдаёт памяти команду чтения.
- Выставленное число является для памяти адресом. Память, получив адрес и команду чтения, выставляет содержимое, хранящееся по этому адресу, на шину данных ШД и по шине ШУ сообщает о готовности.
- МП получает число с шины данных, интерпретирует его как команду (машинную инструкцию) из своей системы команд и исполняет её (по подпрограмме, хранимой в ПЗУ).
- Если последняя команда не является командой перехода, то МП увеличивает на единицу (в предположении, что длина каждой команды равна еди-

нице) число, хранящееся в счётчике команд; в результате там образуется адрес следующей команды.

Данный цикл выполняется неизменно, и именно он называется **процессом** (откуда и произошло название устройства).

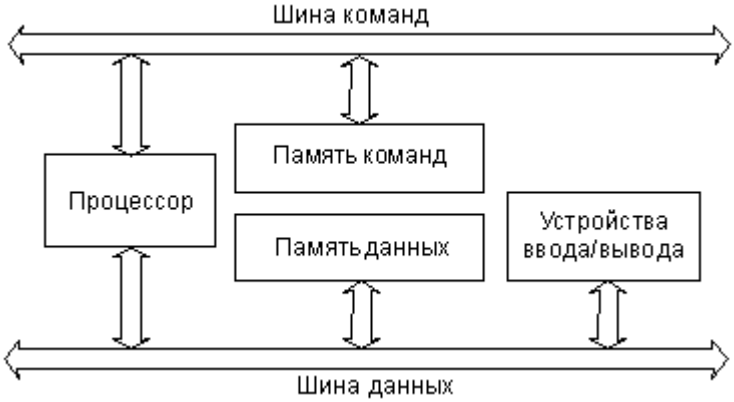
Во время процесса МП считывает последовательность команд, содержащихся в памяти, и исполняет их. Такая последовательность команд называется программой и представляет алгоритм работы МП. Очерёдность считывания команд изменяется в случае, если процессор считывает команду перехода, тогда адрес следующей команды может оказаться другим. Другим примером изменения процесса может служить случай получения команды останова или переключение в режим обработки прерывания.

Команды МП являются самым нижним уровнем управления компьютером, поэтому выполнение каждой команды неизбежно и безусловно. Не производится никакой проверки на допустимость выполняемых действий, в частности, не проверяется возможная потеря ценных данных. Чтобы компьютер выполнял только допустимые действия, команды должны быть соответствующим образом организованы в виде необходимой программы.

Скорость перехода от одного этапа цикла к другому определяется тактовым генератором. Тактовый генератор вырабатывает импульсы, служащие ритмом для центрального процессора. Частота тактовых импульсов в секунду называется тактовой частотой.

1.4. Гарвардская архитектура

Существует альтернативный тип архитектуры МП системы — это архитектура с раздельными шинами данных и команд (двухшинная, или гарвардская). Эта архитектура предполагает наличие в системе отдельной памяти для данных и отдельной памяти для команд. Обмен процессора с каждым из двух типов памяти происходит по своей шине.



В случае двухшинной архитектуры обмен по обеим шинам может быть независимым, параллельным во времени. Соответственно, структуры шин (количество разрядов кода адреса и кода данных, порядок и скорость обмена информацией и т.д.) могут быть выбраны оптимально для той задачи, которая решается каждой шиной. Поэтому при прочих равных условиях переход на двухшинную архитектуру ускоряет работу МП системы, хотя и требует дополнительных затрат на аппаратуру, усложнения структуры процессора. Память данных в этом случае имеет свое распределение адресов, а память команд — свое.

Архитектура с раздельными шинами данных и команд сложнее, она заставляет процессор работать одновременно с двумя потоками кодов, обслуживать обмен по двум шинам одновременно. Программа может размещаться только в памяти команд, данные — только в памяти данных. Такая узкая специализация ограничивает круг задач, решаемых системой, так как не дает возможности гибкого перераспределения памяти. Память данных и память команд в этом случае имеют не слишком большой объем, поэтому применение систем с данной архитектурой ограничивается обычно не слишком сложными задачами.

В чем же преимущество архитектуры с двумя шинами (гарвардской)? В первую очередь, в быстродействии. Проще всего преимущества двухшинной архитектуры реализуются внутри одной микросхемы. В этом случае можно также существенно уменьшить влияние недостатков этой архитектуры. Поэтому основное ее применение — в микроконтроллерах, от которых не требуется решения слишком сложных задач, но зато необходимо максимальное быстродействие при заданной тактовой частоте.

1.5. Параллельная архитектура

Архитектура фон Неймана обладает тем недостатком, что она последовательная. Какой бы огромный массив данных ни требовалось обработать, каждый его байт должен будет пройти через МП, даже если над всеми байтами требуется провести одну и ту же операцию. Этот эффект называется **узким горлышком фон Неймана**.

Для преодоления этого недостатка предлагались и предлагаются архитектуры процессоров, которые называются **параллельными**. Параллельные процессоры используются в суперкомпьютерах. Возможными вариантами параллельной архитектуры могут служить (по классификации Флинна):

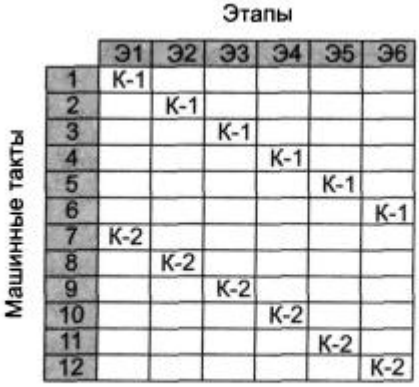
- SISD — один поток команд, один поток данных;
- SIMD — один поток команд, много потоков данных;
- MISD — много потоков команд, один поток данных;
- MIMD — много потоков команд, много потоков данных.

1.6. Конвейерная архитектура

Конвейерная архитектура (Pipelining) была введена в ЦП с целью повышения быстродействия. Обычно для выполнения каждой команды требуется осуществить некоторое количество однотипных операций, например: выборка команды из ОЗУ, дешифровка команды, адресация операнда в ОЗУ, выборка операнда из ОЗУ, выполнение команды, запись результата в ОЗУ. Каждую из этих операций сопоставляют одной ступени конвейера. Например, конвейер МП с архитектурой IA-32 содержит 6 модулей, корые выполняют стадии обработки:

- Модуль шинного интерфейса , ввод/вывод данных.
- Модуль предварительной выборки, считывание инструкции и помещение ее в очередь.
- Модуль декодирования, выборка инструкции из очереди и ее декодирование.
- Модуль выполнения, исполняет последовательность инструкций, полученных от модуля декодирования. операций,
- Модуль сегментации, преобразует логические адреса в линейные адреса и выполняет проверки, связанные с защитой памяти.
- Модуль страничной организации преобразует линейные адреса в физические адреса памяти, выполняет проверки адресов, связанные с защитой страниц памяти, а также ведет список страниц, к которым недавно осуществлялся

Предположим, что каждый этап выполнения команды в процессоре длится ровно 1 машинный такт. На рисунке показана матрица шестиступенчатого выполнения команд в процессоре, не поддерживающем режим конвейерной обработки. Подобный режим выполнения команд был реализован в процессорах фирмы Intel до появления на свет модели Intel486.



Шестиступенчатый цикл выполнения команды в процессоре, не поддерживающем режим конвейерной обработки

Как только завершается этап Э6 выполнения команды К-1, начинается выполнение этапа Э1 команды К-2. При этом для выполнения двух команд К-1 и К-2 требуется 12 машинных тактов. Другими словами, если цикл выполнения команды состоит из к этапов, то для выполнения последовательности из n команд потребуется n x к машинных тактов. Очевидно, что подобный ЦП работает крайне неэффективно, поскольку за 1 машинный такт выполняется только одна шестая часть команды.

В то же время, если в процессоре поддерживается режим конвейерной обработки, то уже на втором машинном такте процессор может приступить к этапу Э1 выполнения новой команды. При этом предыдущая команда будет находиться на этапе Э2 своего выполнения. Таким образом, конвейерная обработка позволяет совместить выполнение двух машинных команд во времени. Как только процессор переходит к этапу Э2 выполнения команды К1, начинается выполнение этапа Э1 команды К-2. Вследствие этогодля выполнения 2 машинных команд требуется уже не 12, а всего лишь 7 машинных тактов. При полной загрузке конвейера, в текуший момент времени работают все 6 его ступеней.

Шестиступенчатый цикл выполнения команды в процессоре, поддерживающем режим конвейерной обработки

Факторы, снижающие эффективность конвейера:

- Простой конвейера, когда некоторые ступени не используются (например, адресация и выборка операнда из ОЗУ не нужны, если команда работает с регистрами).
- Ожидание: если следующая команда использует результат предыдущей, то последняя не может начать выполняться до выполнения первой (это преодолевается при использовании внеочередного выполнения команд — out-of-order execution).
- Очистка конвейера при попадании в него команды перехода (эту проблему удаётся сгладить, используя предсказание переходов).

Некоторые современные процессоры имеют более 30 ступеней в конвейере, что увеличивает производительность процессора, однако приводит к большому времени простоя (например, в случае ошибки в предсказании условного перехода). Не существует единого мнения по поводу оптимальной длины конвейера: различные программы могут иметь существенно различные требования.

Процессор с одним конвейером называется **скалярным**.

1.7. Суперскалярная архитектура

Процессор, построенный по суперскалярной архитектуре, имеет 2 (или больше) конвейера для выполнения команд. Это позволяет одновременно выполнять 2 (или больше) команды. Чтобы лучше понять целесообразность применения суперскалярной архитектуры в процессоре, давайте рассмотрим предыдущий пример конвейерной обработки, в котором мы для упрощения предполагали, что этап выполнения команды (Э4) длится всего 1 машинный такт. А что же

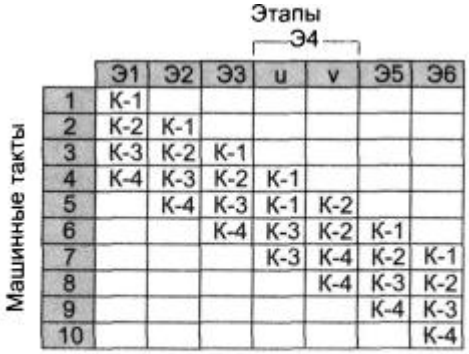
произойдет, если этап выполнения команды Э4 длится 2 машинных такта? Тогда в работе конвейера возникнут сбои, как показано на рисунке.

Цикл выполнения команды на одном конвейере

Процессор не сможет перейти к фазе выполнения Э4 команды К2, пока он полностью не завершит фазу выполнения команды К1. В результате цикл выполнения команды К-2 увеличится на 1 машинный такт, т.е. на время ожидания освобождения конвейера на этапе Э4. По мере поступления на конвейер дополнительных команд, некоторые его ступени будут работать вхолостую (на рисунке они выделены серым цветом).

Для борьбы с простоями оборудования используются нескольео конвеййеров. В процессоре Intel Pentium было применено 2 конвейера. Он стал первым процессором семейства IA-32, построенным по суперскалярной архитектуре. В процессоре Pentium Pro впервые было применено 3 конвейера.

Продолжим рассмотрение нашего примера шестиступенчатого конвейера и введем в него еше один (т.е. второй) конвейер. Как и раньше мы будем предполагать, что фаза выполнения команды Э4 длится 2 машинных такта. Как показано на рисунке, команда с нечетным номером поступает на u-конвейер, а команда с четным номером — на v-конвейер. Подобный подход позволяет ликвидировать простои в работе конвейера.



Принцип работы шестиступенчатого конвейера процессора с суперскалярной архитектурой

В МП с такой архитектурой применяется распараллеливание исполнения команд между несколькими конвейерами, причём решение о параллельном исполнении команд принимается **аппаратурой процессора на этапе исполнения**. Эффективное использование такой архитектуры требует специальной оптимизации машинного кода в компиляторе для генерации пар независимых команд (когда результат одной команды не является аргументом другой).

Суперскалярные МП могут выдавать на выполнение в каждом такте переменное число команд, и работа их конвейеров может планироваться как статически с помощью компилятора, так и с помощью аппаратных средств динамической оптимизации. Суперскалярные машины используют параллелизм на уровне команд путем посылки нескольких команд из обычного потока команд в несколько функциональных устройств.

В типичной суперскалярной машине аппаратура может осуществлять выдачу от 1 до 8 команд в одном такте. Обычно эти команды должны быть независимыми и удовлетворять некоторым ограничениям, например таким, что в каждом такте не может выдаваться более одной команды обращения к памяти. Если какая-либо команда в потоке команд является логически зависимой или не удовлетворяет критериям выдачи, на выполнение будут выданы только команды, предшествующие данной. Поэтому скорость выдачи команд в суперскалярных машинах является переменной

1.8. Архитектура VLIW

Архитектуры VLIW (Very Long Instruction Word) используют очень длинное слово команды. Отличаются от суперскалярной архитектуры тем, что решение о распараллеливании принимается **компилятором на этапе генерации кода**, а не аппаратурой на этапе исполнения. Команды очень длинны и содержат явные инструкции по распараллеливанию нескольких субкоманд на несколько устройств исполнения.

VLIW процессором в его классическом виде является Itanium. Разработка эффективного компилятора для VLIW является сложнейшей задачей. Преимущество VLIW перед суперскалярной архитектурой заключается в том, что компилятор может быть более развитым, нежели устройства управления процессора, и он способен хранить больше контекстной информации для принятия более верных решений по оптимизации.

VLIW - архитектура процессоров с несколькими вычислительными устройствами. Характеризуется тем, что одна инструкция процессора содержит несколько операций, которые должны выполняться параллельно. Задача распределения между ними работы решается **программно**

1.9. Архитектуры CISC, RISC

Архитектура CISC – это классическая архитектура. В ней использована полная система команд (около 300 команд), из которых практически используется третья часть. В настоящее время применяется для совместимости с созданными ранее программами.

Пример использования - иямикроконтроллер MCS-51 (Intel 8051) . Для него существует кросс-ассемблер ASM51, выпущенный корпорацией MetaLink. Кроме того, многие фирмы — разработчики программного обеспечения, такие как IAR или Keil, представили свои варианты ассемблеров. В ряде случаев применение этих ассемблеров оказывается более эффективным благодаря удобному набору директив и наличию среды программирования, объединяющей в себе профессиональный ассемблер и язык программирования Си, отладчик и менеджер программных проектов.

Архитектура RISC – это современная архитектура. В ней использована сокращенная система команд (около ё00 команд), из которых практически все используются. Наиболее употребим реализация AVR. На данный момент существуют 2 компилятора производства Atmel (AVRStudio 3 и AVRStudio4). Вторая версия — попытка исправить не очень удачную первую.

1.10. Ассемблеры

1.10.1. Программа Ассемблер

Ассемблер (assembler — сборщик) — компьютерная программа, компилятор исходного текста программы, написанной на языке ассемблера, в программу на машинном языке.

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

Ассемблирование может быть не первым и не последним этапом на пути получения исполнимого модуля программы. Так, многие компиляторы с языков программирования высокого уровня выдают результат в виде программы на языке ассемблера, которую в дальнейшем обрабатывает ассемблер. Также результатом ассемблирования может быть не исполнимый, а **объектный** модуль, содержащий разрозненные и непривязанные друг к другу части машинного кода и данных программы, из которого (или из нескольких объектных модулей) в дальнейшем с помощью программы-компоновщика («линкера») может быть скомпонован исполнимый файл.

Ассемблеры для DOS. Наиболее известными ассемблерами для операционной системы DOS являлись Borland Turbo Assembler (TASM), Microsoft Macro Assembler (MASM) и Watcom Assembler (WASM). Также в своё время был популярен простой ассемблер A86.

Ассемблеры для Windows. При появлении операционной системы Windows появилось расширение TASM, именуемое TASM 5+ (неофициальный пакет, созданный человеком с ником !tE), позволившее создавать программы для выполнения в среде Windows. Последняя известная версия TASM — 5.3, поддерживающая инструкции MMX, на данный момент включена в **Turbo C++ Explorer**. Но официально развитие программы полностью остановлено.

Microsoft поддерживает свой продукт под названием **Microsoft Macro Assembler** (MASM). Она продолжает развиваться и по сей день.. Но версия программы, направленная на создание программ для DOS, не развивается. Кроме того, Стивен Хатчессон создал пакет для программирования на MASM под названием «MASM32».

Ассемблеры для GNU и GNU/Linux. В состав операционной системы GNU входит пакет binutils, включающий в себя ассемблер **gas** (GNU Assembler), использующий **AT&T-синтаксис**, в отличие от большинства других популярных ассемблеров, которые используют **Intel-синтаксис** (поддерживается с версии 2.10).

Переносимые ассемблеры.

Также существует открытый проект ассемблера, версии которого доступны под различные операционные системы, и который позволяет получать объектные файлы для этих систем. Это:

- **NASM** (Netwide Assembler).
- **YASM** — это переписанная с нуля версия NASM под лицензией BSD (с некоторыми исключениями).
- **FASM** (flat assembler) — молодой ассемблер под модифицированной для запрета перелицензирования BSD-лицензией. Есть версии для Linux, DOS и Windows; использует **Intel-синтаксис** .

1.10.2. Язык Ассемблер

Язы́к Ассе́мблер — язык программирования низкого уровня, мнемонические команды которого (за редким исключением) соответствуют инструкциям процессора вычислительной системы. Трансляция программы в исполняемый машинный код производится программой Аассемблер - транслятором, которая и дала языку ассемблера его название.

Команды языка ассемблера один к одному соответствуют командам процессора, фактически, они представляют собой более удобную для человека символьную форму записи (**мнемокод**) команд и их **аргументов**. При этом одной команде языка ассемблера может соответствовать несколько вариантов команд процессора, в зависимости от операндов.

Кроме того, язык ассемблера позволяет использовать **символические метки** вместо адресов ячеек памяти, которые при ассемблировании заменяются на автоматически рассчитываемые абсолютные или относительные адреса, а так-

же так называемые **директивы** (команды, не переводящиеся в процессорные инструкции, а выполняемые самим ассемблером).

Директивы ассемблера позволяют, в частности, включать блоки данных, задать ассемблирование фрагмента программы по условию, задать значения меток, использовать **макроопределения** с параметрами.

Каждая модель (или семейство) процессоров имеет свой набор команд и соответствующий ему язык ассемблера. Наиболее популярные синтаксисы — **Intel-синтаксис** и **AT&T-синтаксис**.

Достоинства:

- При достаточной квалификации программиста, язык ассемблера позволяет писать самый **быстрый и компактный** код. Возможно, даже лучше, чем генерируемый трансляторами языков более высокого уровня.
- Если код программы достаточно большой, данные, которыми он оперирует, не помещаются целиком в регистрах процессора, то есть частично или полностью находятся в оперативной памяти, то искусный программист, как правило, способен **значительно** оптимизировать программу по сравнению с высокоуровневыми трансляторами по одному или нескольким параметрам: скорость работы (за счёт оптимизации вычислений и/или более рационального обращения к ОП, перераспределения данных), объём кода (в том числе за счёт эффективного использования промежуточных результатов).
- Обеспечение максимального использования специфических возможностей **конкретной платформы**, что также позволяет создавать более эффективные программы с меньшими затратами ресурсов.
- При программировании на языке ассемблера возможен **непосредственный доступ к аппаратуре**, в том числе портам ввода-вывода, регистрам процессора и др.
- Язык ассемблера применяется для создания драйверов оборудования и ядра операционной системы.
- Язык ассемблера используется для создания «прошивок» BIOS.
- С помощью языка ассемблера создаются компиляторы и интерпретаторы языков высокого уровня, а также реализуется совместимость платформ.
- Существует возможность исследования других программ с отсутствующим исходным кодом с помощью дизассемблера.

Недостатки:

- В силу машинной ориентации («низкого» уровня) языка ассемблера человеку сложнее читать и понимать программу на нём по сравнению с языками

программирования высокого уровня; программа состоит из слишком «мелких» элементов — машинных команд, соответственно, усложняются программирование и отладка, растёт трудоёмкость, велика вероятность внесения ошибок.

- Требуется высокая квалификация программиста. Код на ассемблере выполняется быстрее, но написанный неопытным программистом, обычно оказывается хуже сгенерированного компилятором[2]
- Как правило, меньшее количество доступных библиотек по сравнению с современными индустриальными языками программирования.
- Отсутствует переносимость программ на компьютеры с другой архитектурой и системой команд.

1.10.3. Основы 32-битного программирования в Windows

Вводная информация по средствам программирования на языке ассемблера предназначена для начинающих программирование на ассемблере, поэтому программистам более опытным ее можно пропустить без особого ущерба для себя. Технологии трансляции и в MS DOS, и в Windows весьма схожи. Однако программирование в MS DOS уходит в прошлое.

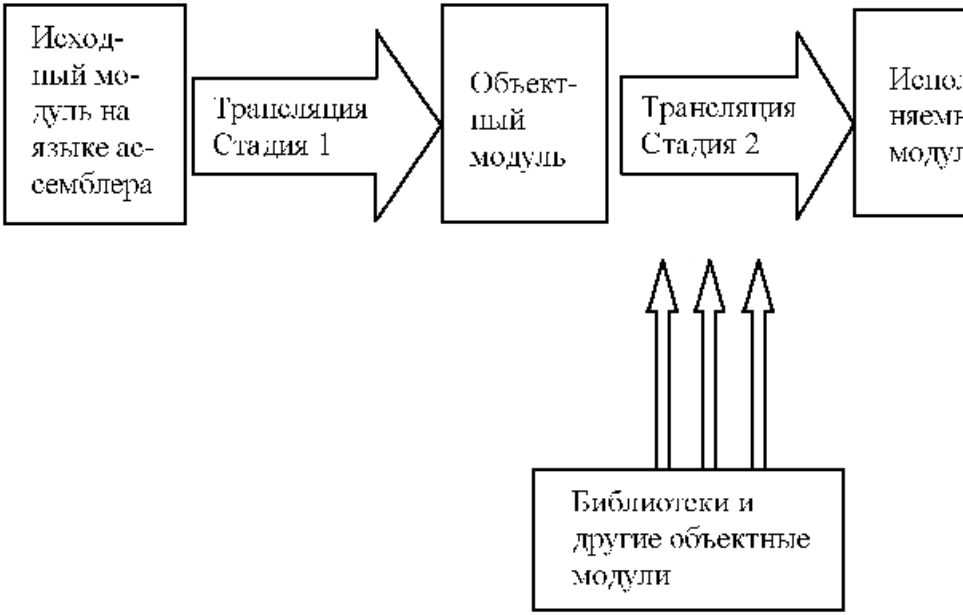


Схема трансляции ассемблерного модуля.

Двум стадиям трансляции соответствуют две основные программы: ассемблер ML.EXE и редактор связей LINK.EXE. Пусть файл с текстом программы на языке ассемблера называется PROG.ASM, тогда две стадии трансляции будут выглядеть следующим образом:

- Стадия 1 - в результате появляется модуль PROG.OBJ.
- Стадия 2 - в результате появляется исполняемый модуль PROG.EXE.

Формат конечного модуля зависит от операционной системы. Установив стандарт на структуру объектного модуля, мы получаем возможность:

- использовать уже готовые объектные модули,
- стыковать между собой программы, написанные на разных языках.

Если стандарт **объектного** модуля распространить на разные операционные системы, то можно использовать модули, написанные в разных операционных системах.

Ниже краткий обзор ряда других программ, которые часто используются при программировании на ассемблере.

Редакторы. Начну с редактора QEDITOR.EXE, который поставляется вместе с пакетом MASM32. Сам редактор и все сопутствующие ему утилиты написаны на ассемблере. Анализ их размера и возможностей действительно впечатляет. Например, сам редактор имеет длину всего 27 Кб, а утилита, используемая для просмотра отчетов о трансляции — всего 6 Кб. Редактор вполне годится для работы с небольшими одномодульными приложениями. Для работы с несколькими модулями он не очень удобен. Работа редактора основана на взаимодействии с различными утилитами посредством пакетных файлов. Например, трансляцию программ осуществляет пакетный файл ASSMBL.BAT, который использует ассемблер ML.EXE, а результат ассемблирования направляется в текстовый файл ASMBL.TXT. Далее для просмотра этого файла используется простая утилита THEGUN.EXE. Аналогично осуществляется редактирование связей. Для дизассемблирования исполняемого модуля используется утилита DUMPPE.EXE, результат работы этой утилиты помещается в текстовый файл DISASM.TXT.

Вторая программа, с которой я хочу познакомить читателя, это EAS.EXE (Easy Assembler Shell). Редактор, а точнее оболочка, позволяет создавать и транслировать довольно сложные проекты, состоящие из ASM-,OBJ-,RC-,RES-,DEF-файлов. Программа позволяет работать как с TASM, так и MASM, а также с другими утилитами (отладчиками, редакторами ресурсов и т.д.). Непосредственно в программе можно настроить компиляторы и редакторы связей на определенный режим работы путем задания ключей этих утилит.

Отладчики. Отладчики позволяют исполнять программу в пошаговом режиме. Несколько наиболее известных отладчиков CodeView (Микрософт), Turbo Debugger (Borland), Ice.

Дизассемблеры. Дизассемблеры переводят исполняемый модуль в ассемблерный код. Примером простейшего дизассемблера является программа DUMPPE.EXE, работающая в строковом режиме. Отмечу также дизассемблер W32Dasm и знаменитый дизассемблер IDA Pro.

Hex-редакторы. Hex-редакторы позволяют просматривать и редактировать загружаемые модули в шестнадцатеричном виде. Их великое множество, к тому же отладчики и дизассемблеры, как правило, имеют встроенные HEX-редакторы. Отмечу только, весьма популярную в хакерских кругах программу HIEW.EXE. Эта программа позволяет просматривать загружаемые модули как в

шестнадцатеричном виде, так и в виде ассемблерного кода. И не только просматривать, но и редактировать.

Редакторы ресурсов. Ресурсы - это готовые шаюлоны, которые можно включать в коды. Простые ресурсы можно создавать в обычном текстовом редакторе. Язык описания ресурсов будет подробно рассмотрен далее.

Компиляторы ресурсов. Они превращают текст ресурса в модуль. В пакетах MASM32 и TASM32 есть компиляторы ресурсов, которые будут описаны ниже. Это программы RC.EXE и BRC32.EXE соответственно.

Для начала программирования на ассемблере в среде Windows важны два момента:

- Вызов системных функций API (Application Program Interface, т.е. интерфейс программного приложения).
- Возможные структуры программ для Windows. Можно выделить 3 типа структуры программ - классическая, диалоговая (основное окно — диалоговое), консольная (или безоконная)структура.

Начнем с нескольких общих положений о программировании в Windows. Те, кто уже имеет опыт программирования в среде Windows, могут на этом не останавливаться.

Программирование в Windows основывается на использовании функций API. Их количество достигает двух тысяч. Ваша программа в значительной степени будет состоять из таких вызовов. Все взаимодействие с внешними устройствами и ресурсами операционной системы будет происходить посредством таких функций.

- Список функций API и их описание лучше всего брать из файла WIN32.HLP, который поставляется, например, с пакетом Borland C++.
- Главным элементом программы в среде Windows является окно. Для каждого окна определяется своя процедура обработки сообщений (см. ниже).
- Окно может содержать элементы управления: кнопки, списки, окна редактирования и др. Эти элементы, по сути, также являются окнами, но обладающими особыми свойствами. События, происходящие с этими элементами (и самим окном), приводят к приходу сообщений в процедуру окна.
- Операционная система Windows использует линейную модель памяти. Другими словами, всю память можно рассматривать как один сегмент. Для программиста на языке ассемблера это означает, что адрес любой ячейки

памяти будет определяться содержимым одного 32-битного регистра, например EBX.

- Мы фактически не ограничены в объеме данных, кода или стека (объеме локальных переменных). Выделение в тексте программы сегмента кода и сегмента данных является теперь простой формальностью, улучшающей читаемость программы.
- Операционная система Windows является многозадачной средой. Каждая задача имеет свое адресное пространство и свою очередь сообщений. Более того, даже в рамках одной программы может быть осуществлена многозадачность - любая процедура может быть запущена как самостоятельная задача.

Начнем с того, как можно вызвать функции API, например, MessageBox. Это ее синтаксис:

```
int MessageBox ( HWND hWnd, LPCTSTR lpText, LPCTSTR lpCaption, UINT uType );
```

Данная функция выводит на экран окно с сообщением и кнопкой (или кнопками) выхода. Для каждого аргумента определены тип (32-битные целые числа) и значение:

- hWnd -дескриптор окна, в котором будет появляться окно-сообщение, тип HWND — 32-битное целое.
- lpText - текст, который будет появляться в окне, тип LPCTSTR — 32-битный указатель на строку.
- lpCaption - текст в заголовке окна, тип LPCTSTR — 32-битный указатель на строку.
- uType - тип окна, в частности можно определить количество кнопок выхода. Тип UINT — 32-битное целое.

К имени функций нужно добавлять суффикс "A" при использовании кодировки символов ANSI, кроме того, при использовании MASM необходимо также в конце имени добавить @16. Вызов указанной функции будет выглядеть так:

```
CALL MessageBoxA@16.
```

Аргументы будут извлекаться из стека в порядке их упоминанию в функции. Чтобы это было возможно надлежит ввести в стек **до вызова** функции в **обратном** порядке.

Пример для MessageBox.

```
; Создать аргументы

MB_OK   equ 0                          ; uType создать
STR1    DB "Неверный ввод! ",0         ; lpText создать
STR2    DB "Сообщение об ошибке.",0    ; lpCaption создать
HW      DWORD ?                        ; hwind создать

; Поместить аргументы в стек


PUSH    MB_OK                          ; uType в стек
PUSH    OFFSET STR2                    ; lpCaption в стек
PUSH    OFFSET STR1                    ; lpText в стек
PUSH    HW                             ; hwind в стек

CALL    MessageBoxA@16                 ; вызов функции
```

Теперь обратимся к структуре всей программы.В классической структуре программы под Windows имеется главное окно, а следовательно, и процедура главного окна. В целом, в коде программы можно выделить следующие секции:

- Регистрация класса окон.
- Создание главного окна.
- Цикл обработки очереди сообщений.
- Процедура главного окна.

Регистрация класса окон. Регистрация класса окон осуществляется с помощью функции RegisterClassA, единственным параметром которой является указатель на структуру WNDCLASS, содержащую информацию об окне (см. пример ниже).

Создание окна. На основе зарегистрированного класса с помощью функции CreateWindowExA (или CreateWindowA) можно создать экземпляр окна.

Цикл обработки очереди сообщений. В нем используются функции:

- GetMessage(), которая "отлавливает" очередное сообщение из ряда сообщений данного приложения и помещает его в структуру MSG.
- TranslateMessage, которая используется для сообщений WM_KEYDOWN и WM_KEYUP, которые транслируются в WM_CHAR и WM_DEADCHAR,
- а также WM_SYSKEYDOWN и WM_SYSKEYUP, преобразующиеся в WM_SYSCHAR и WM_SYSDEADCHAR.

Смысл трансляции заключается не в замене, а в отправке дополнительных сообщений. Так, например, при нажатии и отпускании алфавитно-цифровой клавиши в окно сначала придет сообщение WM_KEYDOWN, затем WM_KEYUP, а затем уже WM_CHAR. Выход из цикла ожиданий имеет место только в том случае, если функция GetMessage возвращает 0. Это происходит только при получении сообщения о выходе (сообщение WM_QUIT). Таким образом, цикл ожидания играет двоякую роль: определенным образом преобразуются сообщения, предназначенные для какого-либо окна, и ожидается сообщение о выходе из программы.

Процедура главного окна. Используется функция WindowFunc. Ее прототип на языке С

WindowFunc(HWND hwnd, UINT message, WPARAM wParam, LPARAM lParam)

Для каждого аргумента определены тип (32-битные целые числа) и значение:

- hWnd - дескриптор окна, в котором будет появляться окно-сообщение, тип HWND — 32-битное целое.
- Message – идентификатор сообщения. Тип UINT — 32-битное целое.
- WPARAM – уточнение. Тип wParam — 32-битное целое.
- LPARAM - уточнение. Тип lParam — 32-битное целое..

Рассмотрим "скелет" этой функции на языке ассемблера.

```
; Создать аргументы

MB_OK    equ 0                          ; uType создать
STR1     DB "Неверный ввод! ",0         ; lpText создать
STR2     DB "Сообщение об ошибке.",0    ; lpCaption создать
HW       DWORD ?                        ; hwind создать
WNDPROC PROC                            ; определение функции WndProc
     PUSH EBP                           ; получить адрес вершины стека
     MOV EBP, ESP                       ; теперь EBP указывает на вершину
стека
; поместить в стек контекст предыдущей программы
     PUSH EBX
     PUSH ESI
     PUSH EDI
; поместить в стек аргументы функции окна через регистр EBP
     PUSH DWORD PTR [EBP+14H]           ; LPARAM (lParam) в стек
```

```
    PUSH DWORD PTR [EBP+10H]        ; WPARAM (wParam) в стек
    PUSH DWORD PTR [EBP+0CH]        ; MES (message) в стек
    PUSH DWORD PTR [EBP+08H]        ; HWND (hwnd) в стек
   CALL DefWindowProcA@16           ; вызов внешней функции окна
; вернуть из стека контекст предыдущей программы (в обратном порядке)
    POP EDI
    POP ESI
    POP EBX
    POP EBP
    RET 16                          ; выход из функции освобождением
стека от четырех параметров (16=4*4).
    WNDPROC ENDP                    ; конец функции WndProc
```

1.10.4. API функции

Ниже перечислены основные функции API, применяемые в ассемблере.

Функция	Назначение функции
AllocConsole	Создать консоль
Arc	Рисовать дугу
BeginPaint	Получить контекст при получении сообщения WM_PAINT
BitBlt	Скопировать виртуальную прямоугольную область в окно
CallNextHookEx	Продолжить выполнение других фильтров.
CallWindowProc	Вызвать процедуру окна.
CharToOem	Функция перекодировки строки.
CloseHandle	Закрыть объект: файл, консоль, коммуникационный канал.
CreateCompatibleBitmap	Создать карту бит, совместимую с заданным контекстом.
CreateCompatibleDC	Создать контекст, совместимый с данным окном.
CreateDialogParam	Создать немодальное диалоговое окно.
CreateEvent	Создать событие.
CreateFile	Создать или открыть файл, консоль, коммуникационный канал и т.п.
CreateFileMapping	Создать отображаемый файл.
CreateFont	Задать параметры шрифта.
CreateFontIndirect	Задать параметры шрифта.

CreateMutex	Создать объект синхронизации "взаимоисключение"
CreatePen	Создать перо.
CreatePipe	Создать канал обмена информацией.
CreateProcess	Создать новый процесс.
CreateSemaphore	Создать семафор.
CreateSolidBrush	Определить кисть.
CreateThread	Создать поток.
CreateWindow	Создать окно.
CreateWindowEx	Расширенное создание окна.
DefWindowProc	Вызывается для сообщений, которые не обрабатываются функцией окна.
DeleteCriticalSection	Удалить объект "критическая секция".
DeleteDC	Удалить контекст, полученный посредством функций типа CreatePen или CreateDC.
DeleteObject	Удалить объект, выбранный функцией SelectObject.
DestroyMenu	Удалить меню из памяти.
DestroyWindow	Удалить окно из памяти.
DeviceIoControl	Вызов сервиса динамического виртуального драйвера.
DialogBox	Создать модальное диалоговое окно.
DialogBoxParam	Создать немодальное диалоговое окно.
DispatchMessage	Вернуть управление Windows с передачей сообщения предназначенному окну.
Ellipse	Рисовать эллипс.
EndDialog	Удалить модальное диалоговое окно.
EndPaint	Удалить контекст, полученный при помощи BeginPaint.
EnterCriticalSection	Войти в критическую секцию.
EnumWindows	Пересчитать окна.
ExitProcess	Закончить данный процесс со всеми подзадачами (потоками).
ExitThread	Выход из потока с указанием кода выхода.
FindFirstFile	Первый поиск файлов в каталоге.
FindNextFile	Осуществить последующий поиск в каталоге.
FlushViewOfFile	Сохранить отображаемый файл или его часть на диск.

FreeConsole	Освободить консоль.
FreeLibrary	Выгрузить динамическую библиотеку.
GetCommandLine	Получить командную строку программы.
GetCursorPos	Получить положение курсора в экранных координатах.
GetDC	Получить контекст окна.
GetDiskFreeSpace	Определяет объем свободного пространства на диске.
GetDlgItem	Получить дескриптор управляющего элемента в окне.
GetDriveType	Получить тип устройства.
GetLocalTime	Получить местное время.
GetMenuItemInfo	Получить информацию о выбранном пункте меню.
GetMessage	Получить очередное сообщение из очереди сообщений данного приложения.
GetModuleHandle	Получить дескриптор приложения.
GetProcAddress	Получить адрес процедуры (в динамической библиотеке).
GetStdHandle	Получить дескриптор консоли.
GetStockObject	Определить дескриптор стандартного объекта.
GetSystemDirectory	Получить системный каталог.
GetSystemMetrics	Определить значение системных характеристик.
GetSystemTime	Получить время по Гринвичу.
GetTextExtentPoint32	Определить параметры текста в данном окне.
GetWindowRect	Определить размер окна.
GetWindowsDirectory	Получить каталог Windows.
GetWindowText	Получить заголовок окна.
GetWindowThreadProcessId	Получить идентификатор процесса.
GlobalAlloc	Выделить блок памяти.
GlobalDiscard	Удалить удаляемый блок памяти.
GlobalFree	Освободить блок памяти.
GlobalLock	Фиксировать перемещаемый блок памяти.
GlobalReAlloc	Изменить размер блока памяти.
GlobalUnlock	Снять фиксацию блока памяти.
InitializeCriticalSection	Создать объект критическая секция.
InvalidateRect	Перерисовать окно.
KillTimer	Удалить таймер.

LeaveCriticalSection	Покинуть критическую секцию.
LineTo	Провести линию от текущей точки к заданной.
LoadAccelerators	Загрузить таблицу акселераторов.
LoadCursor	Загрузить системный курсор или курсор, определенный в файле ресурсов.
LoadIcon	Загрузить системную иконку или иконку, определенную в файле ресурсов.
LoadLibrary	Загрузить динамическую библиотеку.
LoadMenu	Загрузить меню, которое определено в файле ресурсов.
LoadString	Загрузить строку, определенную в файле ресурсов.
lstrcat	Производит конкатенацию двух строк.
lstrcpy	Скопировать одну строку в другую.
lstrlen	Получить длину строки.
MapViewOfFile	Скопировать файл или части файла в память.
MessageBox	Выдать окно сообщения.
MoveToEx	Сменить текущую точку.
MoveWindow	Установить новое положение программа окна.
OpenEvent	Открыть событие.
OpenSemaphore	Открыть семафор.
PatBlt	Заполнить заданную прямоугольную область.
Pie	Рисовать сектор эллипса.
PostMessage	Аналогична SendMessage, но сразу возвращает управление.
PostQuitMessage	Послать текущему приложению сообщение WM_QUIT.
ReadConsole	Читать из консоли.
ReadFile	Читать из файла или того, что было создано функцией CreateFile.
Rectangle	Рисовать прямоугольник.
RegisterClass	Зарегистрировать класс окон.
RegisterHotKey	Зарегистрировать горячую клавишу.
ReleaseDC	Удалить контекст, полученный при помощи GetDC
ReleaseSemaphore	Освободить семафор
ResetEvent	Сбросить событие
ResumeThread	Запустить "спящий" процесс.

RoundRect	Рисовать прямоугольник с округленными углами.
RtlMoveMemory	Копировать блок памяти в другой блок. В помощи по API-функциям она называется MoveMemory.
SelectObject	Выбрать объект (перо, кисть) в указанном контексте.
SendDlgItemMessage	Послать сообщение управляющему элементу окна.
SendMessage	Послать сообщение окну.
SetBkColor	Установить цвет фона для вывода текста.
SetConsoleCursorPosition	Установить курсор в заданную позицию в консоли.
SetConsoleScreenBufferSize	Установить размер буфера консоли.
SetConsoleTextAttribute	Установить цвет текста в консоли.
SetConsoleTitle	Установить название окна консоли.
SetEvent	Подать сигнал о наступлении события.
SetFocus	Установить фокус на заданное окно.
SetLocalTime	Установить время и дату.
SetMapMode	Установить соотношение между логическими единицами и пикселями.
SetMenu	Назначить новое меню данному окну.
SetPixel	Установить заданный цвет пикселя.
SetSystemTime	Установить время, используя гринвичские координаты.
SetTextColor	Установить цвет текста.
SetTimer	Установить таймер.
SetViewportExtEx	Установить область вывода.
SetViewportOrgEx	Установить начало области вывода.
SetWindowLong	Изменить атрибут уже созданного окна.
SetWindowsHookEx	Установить процедуру-фильтр.
Shell_NotifyIcon	Посредством данной функции можно поместить иконку приложения на системную панель.
SHFileOperation	Осуществляет групповую операцию над файлами и каталогами.
SHGetDesktopFolder	Выводит диалоговое окно для выбора каталогов и файлов.
ShowWindow	Показать окно, установить статус показа.
Sleep	Вызывает задержку.
TerminateProcess	Уничтожить процесс.

TerminateThread	Удалить поток.
TextOut	Вывести текст в окно.
timeKillEvent	Удалить таймер.
timeSetEvent	Установить таймер.
TranslateAccelerator	Транслирует акселераторные клавиши в команду выбора пункта меню.
TranslateMessage	Транслировать клавиатурные сообщения в ASCII-коды.
UnhookWindowsHookEx	Снять процедуру-фильтр.
UnmapViewOfFile	Сделать указатель на отображаемый файл недействительным.
UnregisterHotKey	Снять регистрацию горячей клавиши.
UpdateWindow	Обновить рабочую область окна.
VirtualAlloc	Зарезервировать блок виртуальной памяти или отобразить на него физическую память.
VirtualFree	Снять резервирование с блока виртуальной памяти или сделать блок виртуальной памяти неотображенным.
WaitForSingleObject	Ожидает одно из двух событий: определенный объект сигнализирует о своем состоянии, вышло время ожидания (TimeOut). Работает с такими объектами, как семафор, событие, взаимоисключение, процесс, консольный ввод и др.
WNetAddConnection2	Осуществляет соединение с сетевым ресурсом локальной сети.
WNetCancelConnection2	Отсоединить от ресурса локальной сети.
WNetCloseEnum	Найти все ресурсы локальной сети данного уровня.
WNetGetConnection	Получить информацию о данном соединении.
WNetOpenEnum	Открыть поиск ресурсов в локальной сети.
WriteConsole	Вывод в консоль.
wsprintf	Преобразовать последовательность параметров в строку.

1.11. Сообщения Windows

Ниже перечислены основные сообщения Windows, применяемые в ассемблере.

Сообщение системы	Назначение

WM_ACTIVATE	Посылается функции окна перед активизацией и деактивизацией этого окна.
WM_ACTIVATEAPP	Посылается функции окна перед активизацией окна другого приложения.
WM_CHAR	Сообщение, возникающее при трансляции сообщения WM_KEYDOWN функцией TranslateMessage.
WM_CLOSE	Сообщение, приходящее на процедуру окна при его закрытии. Приходит до WM_DESTROY. Дальнейшее выполнение DefWindowProc, EndDialog или WindowsDestroy и вызывает появление сообщения WM_DESTROY.
WM_COMMAND	Сообщение, приходящее на функцию окна, при наступлении события с управляющим элементом, пунктом меню, а также от акселератора.
WM_CREATE	Первое сообщение, приходящее на функцию окна при его создании. Приходит один раз.
WM_DEADCHAR	Сообщение, возникающее при трансляции сообщения WM_KEYUP функцией TranslateMessage.
WM_DESTROY	Сообщение, приходящее на функцию окна при его уничтожении.
WM_GETTEXT	Посылается окну для получения текстовой строки, ассоциированной с данным окном (строка редактирования, заголовок окна и т.д.).
WM_HOTKEY	Генерируется при нажатии горячей клавиши.
WM_INITDIALOG	Сообщение, приходящее на функцию диалогового окна вместо сообщения WM_CREATE.
WM_KEYDOWN	Сообщение, генерируемое при нажатии клавиши клавиатуры и посылаемое окну, имеющему фокус ввода.
WM_KEYUP	Сообщение, генерируемое при отпускании клавиши клавиатуры и посылаемое окну, имеющему фокус ввода.
WM_LBUTTONDOWN	Сообщение генерируется при нажатии левой кнопки мыши.
WM_MENUSELECT	Посылается окну, содержащему меню, при выборе пункта меню.
WM_PAINT	Сообщение посылается окну перед его перерисовкой.
WM_QUIT	Сообщение, приходящее приложению (не окну) при выполнении функции PostQuitMessage. При получении

	этого сообщения происходит выход из цикла ожидания и, как следствие, выход из программы.
WM_RBUTTONDOWN	Сообщение генерируется при нажатии правой кнопки мыши.
WM_SETFOCUS	Сообщение, посылаемое окну, после того, как оно получило фокус.
WM_SETICON	Приложение посылает окну данное сообщение, чтобы ассоциировать с ним новую иконку (значок).
WM_SETTEXT	Сообщение, используемое приложением для посылки текстовой строки окну и интерпретируемое в зависимости от типа окна (обычное окно - заголовок, кнопка — надпись на кнопке, окно редактирования - содержимое этого окна и т.д.).
WM_SIZE	Посылается функции окна после изменения его размера.
WM_SYSCHAR	Сообщение, возникающее при трансляции сообщения WM_SYSKEYDOWN функцией TranslateMessage.
WM_SYSCOMMAND	Генерируется при выборе пунктов системного меню или меню окна.
WM_SYSDEADCHAR	Сообщение, возникающее при трансляции сообщения WM_SYSKEYUP функцией TranslateMessage.
WM_SYSKEYDOWN	Сообщение аналогично WM_KEYDOWN, но генерируется, когда нажата и удерживается еще и клавиша Alt.
WM_SYSKEYUP	Сообщение аналогично WM_SYSKEYDOWN, но генерируется при отпускании клавиши.
WM_TIMER	Сообщение, приходящее на функцию окна или специально определенную таймерную процедуру после определения интервала таймера при помощи функции SetTimer.
WM_VKEYTOITEM	Сообщение окну приложения, когда нажимается какая-либо клавиша при наличии фокуса на данном списке. Список должен иметь свойство LBS_WANTKEYBOARDINPUT.

1.12. Версии ассемблеров

1.12.1. Microsoft Macro Assembler (MASM)

MASM — ассемблер для процессоров семейства x86. Первоначально был произведён компанией Microsoft для написания программ в операционной системе MS-DOS и был в течение некоторого времени самым популярным ассемблером, доступным для неё. Это поддерживало широкое разнообразие макросредств и структурированность программных идиом, включая конструкции высокого уровня для повторов, вызовов процедур и чередований (поэтому MASM — ассемблер высокого уровня). Позднее была добавлена возможность написания программ для Windows. MASM — один из немногих инструментов разработки Microsoft, для которых не было отдельных 16- и 32-битных версий.

В начале 1990-х годов альтернативные ассемблеры, вроде Borland TASM и свободного ассемблера NASM, начали отбирать часть доли рынка MASM. Однако два события в конце 1990-х позволили MASM сохранить большую часть своей доли: сначала Microsoft прекратила продавать MASM как коммерческий продукт. Во-вторых, благодаря пакету MASM32 оказалось, что программирование на MASM возможно и в среде Microsoft Windows. В 2000 году MASM 6.15 был выпущен как часть пакета разработки Visual C++ и все версии Visual C++ после 6.0 включали в себя версию MASM, равную версии Visual C++. Позже в Visual C++ 2005 появилась 64-битная версия MASM. Вместе с большим сообществом программистов MASM эти события помогли остановить снижение популярности MASM по сравнению с другими ассемблерами. Сегодня MASM продолжает использоваться на платформе Win32, несмотря на конкуренцию с новыми продуктами, такими как NASM, FASM, TASM, HLASM.

Есть много развивающихся проектов для разработки программного обеспечения, которые поддерживают MASM, включая интегрированные среды разработки (ИСР), например RadASM.

Версии MASM. Хотя MASM больше не является коммерческим продуктом, Microsoft продолжает поддерживать исходный код, используемый и в других продуктах Microsoft. С тех пор как Microsoft прекратила продавать MASM отдельно, было выпущено несколько обновлений к производственной линии MASM 6.x (последнее обновление — версия 6.15, которая была включена в Visual C++ 6.0), а после этого — MASM 7.0 в составе Visual C++ .NET 2002, MASM 7.1 в составе Visual C++ .NET 2003, MASM 8.0 в составе Visual C++ 2005 и MASM 9.0 в составе Visual C++ 2008, поддерживающие платформу x64.

1.12.2. Flat assembler (FASM)

FASM - свободно распространяемый многопроходной ассемблер, написанный Томашем Грыштаром (польск. Tomasz Grysztar). FASM написан на самом себе, обладает небольшими размерами и очень высокой скоростью компиляции, имеет богатый и ёмкий макро-синтаксис, позволяющий автоматизировать множество рутинных задач. Поддерживаются как объектные форматы, так и форматы исполняемых файлов. Это позволяет в большинстве случаев обойтись без компоновщика. В остальных случаях нужно использовать сторонние компоновщики, поскольку таковой вместе с FASM не распространяется.

Компиляция программы в FASM состоит из 2 стадий: препроцессирование и ассемблирование. На стадии препроцессора раскрываются все макросы, символические константы, обрабатываются директивы препроцессора. В отличие от стадии ассемблирования, препроцессирование выполняется только 1 раз. Смешивание стадий ассемблирования и препроцессирования — распространённая ошибка начинающих.

На стадии ассемблирования определяются адреса меток, обрабатываются условные директивы, раскрываются циклы и генерируется собственно программа. FASM — многопроходной ассемблер, что позволяет ему делать некоторую оптимизацию, например, генерирование короткого перехода на метку вместо длинного. Во время прохода компилятор не всегда может вычислить выражение в условных директивах. В этом случае он делает какой-нибудь выбор и пытается скомпилировать дальше. Благодаря тому, что адреса меток, вычисленные на N-ном проходе, используются на N+1-проходе, этот процесс обычно сходится.

Используется **Intel-синтаксис** записи инструкций. Единственное существенное отличие от формата, принятого в других ассемблерах (MASM, TASM в режиме совместимости с MASM) — значение ячейки памяти всегда записывается в квадратных скобках - [label_name], а просто label_name означает адрес (то есть порядковый номер) ячейки. Это позволяет обходиться без ключевого слова offset. Также в FASM при переопределении размера операнда вместо byte ptr пишется просто byte, вместо word ptr — word и т. д. Не позволяется использовать несколько квадратных скобок в одном операнде, таким образом вместо [bx][si] необходимо писать [bx+si]. Эти изменения синтаксиса привели к более унифицированному и лёгкому для чтения коду.

Использование FASM поддерживают многие специализированные ИСР, такие как RadASM, WinAsm Studio, Fresh (IDE) (специально спроектированный под FASM) и т. д.

1.12.3. NASM (Netwide Assembler)

NASM — свободный (LGPL и лицензия BSD) ассемблер для архитектуры Intel x86. Используется для написания 16-, 32- и 64-битных программ.

NASM был создан Саймоном Тэтхемом совместно с Юлианом Холлом и в настоящее время развивается небольшой командой разработчиков на SourceForge.net. Первоначально он был выпущен согласно его собственной лицензии, но позже эта лицензия была заменена на GNU LGPL после множества проблем, вызванных выбором лицензии. Начиная с версии 2.07 лицензия заменена на «упрощённую BSD» (BSD из 2 пунктов).

NASM может работать на платформах, отличных от x86, таких как SPARC и PowerPC, однако код он генерирует только для x86 и x86-64.

NASM успешно конкурирует со стандартным в Linux- и многих других UNIX-системах ассемблером **gas**. Считается, что качество документации у NASM выше. Кроме того, ассемблер gas по умолчанию[5] использует AT&T-синтаксис, ориентированный на процессоры не от Intel, в то время как NASM использует вариант традиционного для x86-ассемблера Intel-синтаксиса; Intel-синтаксис используется всеми ассемблерами для Windows-систем, например MASM, TASM, FASM.

NASM компилирует программы под различные операционные системы в пределах x86-совместимых процессоров. Находясь в одной операционной системе, можно беспрепятственно откомпилировать исполняемый файл для другой.

Компиляция программ в NASM состоит из двух этапов. Первый — ассемблирование, второй — компоновка. На этапе ассемблирования создаётся объектный код. В нём содержится машинный код программы и данные, в соответствии с исходным кодом, но идентификаторы (переменные, символы) пока не привязаны к адресам памяти. На этапе компоновки из одного или нескольких объектных модулей создаётся исполняемый файл (программа). Операция компоновки связывает идентификаторы, определённые в основной программе, с идентификаторами, определёнными в остальных модулях, после чего всем идентификаторам даются окончательные адреса памяти или обеспечивается их динамическое выделение.

Для компоновки объектных файлов в исполняемые в Windows можно использовать свободный бесплатно распространяемый компоновщик alink, а в Linux — компоновщик ld, который есть в любой версии этой операционной системы.

В NASM используется Intel-синтаксис записи инструкций.

1.12.4. Turbo Assembler (TASM)

TASM — программный пакет компании Borland, предназначенный для разработки программ на языке ассемблера для архитектуры x86. Кроме того, TASM может работать совместно с трансляторами с языков высокого уровня фирмы Borland, такими как Turbo C и Turbo Pascal. Как и прочие программные пакеты серии Turbo, Турбо Ассемблер больше не поддерживается.

TASM до сих пор используется для обучения программированию на ассемблере под архитектуру x86. Многие находят его очень удобным и продолжают его использовать, расширяя набором дополнительных макросов.

Пакет TASM поставляется вместе с компоновщиком Turbo Linker и порождает код, который можно отлаживать с помощью Turbo Debugger.

По умолчанию TASM работает в режиме совместимости с другим распространённым ассемблером — MASMi, то есть TASM умеет транслировать исходники, разработанные под MASM. Кроме того, TASM имеет режим IDEAL, улучшающий синтаксис языка и расширяющий его функциональные возможности.

1.12.5. GoAsm

GoAsm — ассемблер для процессоров семейства x86, созданный Джереми Гордоном (англ. Jeremy Gordon) для написания программ для операционных систем семейства Windows, способен создавать 32- и 64-битных версий, а также программы с поддержкой Unicode. GoAsm является проприетарным ПО и распространяется в бинарном формате.

GoAsm создавался с целью создать компилятор с простым и ясным синтаксисом, производящий как можно более компактный код, скромными потребностями для обработки скриптов и возможностью добавления расширений. Особенности:

- GoAsm не создаёт 16-разрядный код и способен работать только в «плоском» режиме (без сегментов), благодаря этому синтаксис очень прост.
- В качестве формата выходных данных используется COFF (Portable Executable format), и для создания исполняемых файлов необходимо использовать дополнительный компоновщик (например — GoLink или ALINK) и компилятор ресурсов (GoRC).
- GoAsm способен создавать файлы в формате Unicode (UTF-16 или UTF-8).

Несмотря на то что используется Intel-синтаксис, синтаксис GoAsm несовместим ни с одним из существующих компиляторов. GoAsm использует препроцессор сходный по синтаксису с препроцессором языка программирования C. В GoAsm необходимо использовать квадратные скобки для чтения и записи памяти. Для того чтобы получить смещение необходимо записать:

```
MOV   EBX,ADDR    wParam
MOV   EBX,OFFSET  wParam
```

А для того чтобы получить доступ к памяти:

```
MOV   EBX,[wParam]
```

1.13. Среды разработки

1.13.1. RadASM

RadASM — бесплатная среда разработки программного обеспечения для ОС Windows и не только, изначально предназначенная для написания программ на языке ассемблера.

Имеет гибкую систему файлов настроек, благодаря чему может быть использована как среда разработки программного обеспечения на высокоуровневых языках, а также документов, основанных на языках разметки.

Создана программистом Ketil Olsen (KetilO).

Возможности:

- Есть русифицированный интерфейс.
- Подсветка синтаксиса**.** Это выделение синтаксических конструкций текста с использованием различных цветов, шрифтов и начертаний. Обычно применяется в текстовых редакторах для облегчения чтения исходного текста, улучшения визуального восприятия. Часто применяется при публикации исходных кодов в Интернете.
- Хорошая интеграция справочной системы
- Проекты (собственный .rap формат)
- Редактор ресурсов
- Поддержка команд Make. Это утилита, автоматизирующая процесс преобразования файлов из одной формы в другую. Чаще всего это компиляция исходного кода в объектные файлы и последующая компоновка в исполняемые файлы или библиотеки.
- Окно вывода.

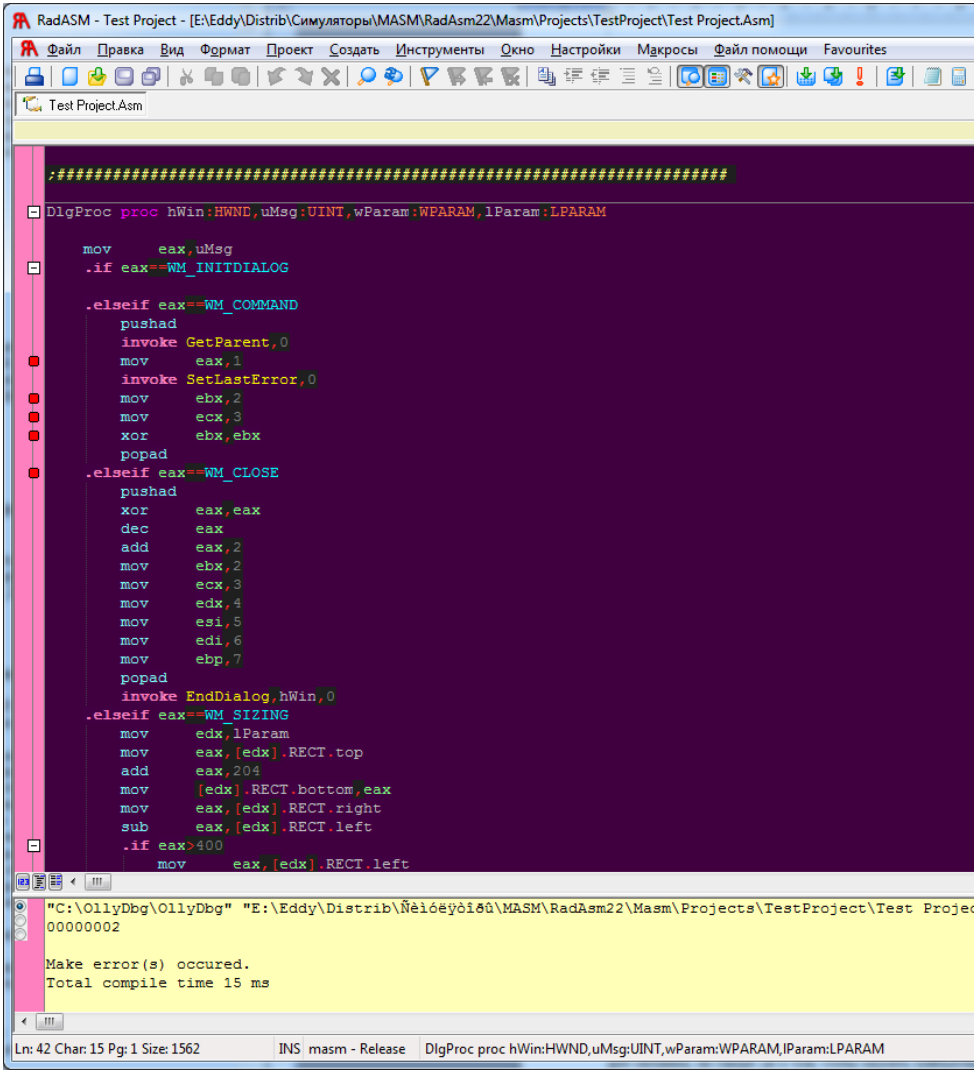
- Поддержка отладки программ. Имеется **отладчик** (дебаггер, англ. debugger), это компьютерная программа, предназначенная для поиска ошибок в других программах, ядрах операционных систем, SQL-запросах и других видах кода. Отладчик позволяет выполнять пошаговую трассировку, отслеживать, устанавливать или изменять значения переменных в процессе выполнения кода, устанавливать и удалять контрольные точки или условия остановки и т.д.
- Полная настройка ИСР.
- Есть окно вывода.
- Макросы, (от англ. macros, множественное число от macro) — программный объект, который во время вычисления заменяется на новый объект, создаваемый определением макроса на основе его аргументов, затем выражается обычным образом.
- Шаблоны, это спецификация формы представления и правил редактирования элемента данных с помощью строки символов, в которой каждый символ указывает на допустимый вид символа или на подлежащее выполнению редактирование для соответствующей позиции значения элемента.
- Поддержка сниппетов. **Сниппет** (англ. snippet — фрагмент, отрывок) — программный термин, обозначающий небольшой фрагмент исходного кода или текста, пригодного для повторного использования. Сниппеты не являются заменой процедур, функций или других подобных понятий структурного программирования. Они обычно используются для более лёгкой читаемости кода функций, которые без их использования выглядят слишком перегруженными деталями, или для устранения повторения одного и того же общего участка кода.
- Поддержка текстовых ссылок.
- Поддержка плагинов. **Плагин** (от англ. plug-in) — независимо компилируемый программный модуль, динамически подключаемый к основной программе, предназначенный для расширения и/или использования её возможностей. Также может переводиться как «модуль». Плагины обычно выполняются в виде разделяемых библиотек.
- Есть примеры использования.
- Есть возможность настроить под другие языки программирования.

Поддерживаемые ассемблеры:

- MASM
- FASM
- NASM
- TASM

- GoAsm

Скриншот ИСР:



1.13.2. WinAsm Studio

WinAsm Studio — бесплатная ИСР для Windows и DOS, изначально предназначенная для написания программ на языке ассемблера.

Создана программистом Антонисом Киприану.

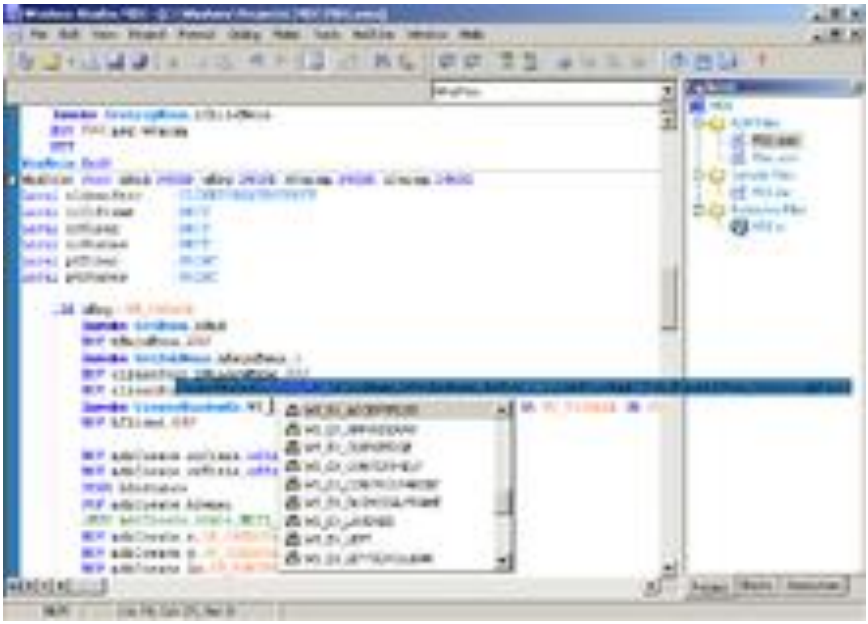
Возможности
- Подсветка синтаксиса.
- Автодополнение кода.
- Менеджер проектов.
- Полная настройка ИСР.
- Есть окно вывода.
- Поддержка плагинов.
- Редактор ресурсов.
- Есть примеры использования.

[Поддерживаемые ассемблеры. По умолчанию, среда разработки ориентирована на работу с MASM, но также возможно и подключение других ассемблеров, к примеру FASM.

Надстройка для подключения ассемблера FASM позволяет выполнять последовательность команд (до 5), что позволяет организовать поддержку практически любого ассемблера.

Также данная надстройка позволяет выполнять пакетные файлы, самокомпилирующиеся пакетные файлы и вызывать утилиты MAKE, что снимает ограничения на использование других компиляторов. Консольные сообщения при этом могут перенаправляться в окно вывода для удобства последующего анализа ошибок.

Скриншот ИСР:



Fresh

Fresh —ИСР на визуальном языке ассемблера для Microsoft Windows со встроенным FASM.

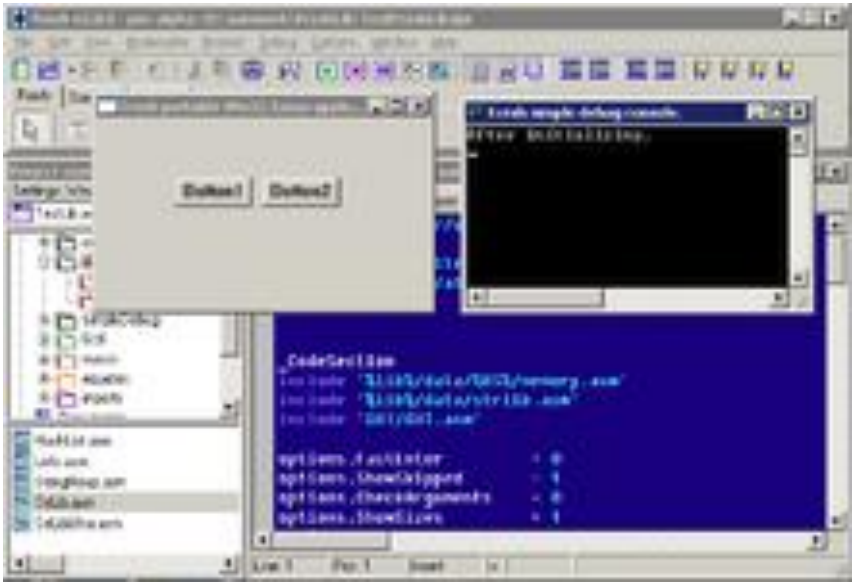
Поддерживает сборку программ для тех же платформ, что и FASM: DOS, Linux, FreeBSD.

В мае 2005 года разработка Fresh была приостановлена[2] и возобновлена лишь через 5 лет в ноябре 2010 года, когда была выпущена версия **Fresh 2.0.**

Разработчики

- Джон Фаунд — основоположник проекта, написавший значительную часть кода последних версий;
- Фредрик Классон (scientica);
- Томми Лиллехаген;
- Виктор Ло (roticv);
- Юнус Сина Гюльшен (VeSCeRa);
- Матьяш Тымек (decard).

Скриншот ИСР:



1.13.3. Easy Code

Скриншот ИСР:

2. Представление данных в ЭВМ

2.1. Системы счисления и преобразования между ними

Различные системы счисления отличаются не только базовым набором чисел, но и основными концепциями, которые лежат в их основе. Взять, например, систему счисления, которая использовалась древними римлянами: она довольно трудна для восприятия, в ней очень сложно производить вычисления и невозможно представить 0. Данная система неудобна даже для человека, не говоря уж о том, чтобы научить компьютер «понимать» ее. Говорят, что римским коммерсантам не был нужун 0 и отрицательные числа, так как они манипулировали числом «штук».

- Десятичные числа

Десятичная система, которую мы используем всю жизнь, относится к классу так называемых **позиционных** систем, в которых число A может быть представлено в виде:

$$A = A_n * Z^n + A_{n-1}*Z^{n-1} + \ldots + A_1*Z^1 + A_0*Z^0$$

Здесь:

- A_n - цифры числа

- Z - основание системы счисления

Например, десятичное число 1234 можно представить так:

$1234 = 1*10^3 + 2*10^2 + 3*10^2 + 4*10^0$.

«Вес» каждой цифры определяется позицией цифры в числе и равен степени основания, соответствующей ее позиции. Позиции нумеруются слева, начиная с 0.

- Двоичные числа

При работе с различными системами счисления мы будем записывать само число в скобках, а за скобками — основание системы. Например, если написать просто число 1100, то не понятно, в какой системе оно записано — это может быть одна тысяча сто, а может быть 12, если число записано в двоичной системе. А если представить число в виде $(1100)_2$, то сразу все становится на свои места: число записано в двоичной системе.

Двоичная система тоже позиционная, поэтому, например, число 1100 в двоичной системе мы можем представить так:

$(1100)_2 = 1*2^3 + 1*2^2 + 0*2^1 + 0*2^0 = 4 + 8 = (12)_{10}$.

Обратите внимание, что для представления числа 12 в двоичной системе использованы только 4 разряда. Наибольшее число, которое можно записать 4 двоичными цифрами, равно 15, потому что

$(1111)_2 = 1*2^3 + 1*2^2 + 1*2^1 + 1*2^0 = 8 + 4 + 2 + 1 = (15)_{10}$.

В программах двоичные числа завершаются суффиксом b. В примере это 1100b.

Для кодирования двоичных чисел применяют несколько кодов.:

Прямой код: 0 в старшем разряде соответствует положительным числам, 1 – отрицательным. Остальные разряды представляют модуль числа. В таком коде удобно осуществлять операции умножения (модули чисел перемножаются, а знаковые разряды складываются по модулю два), но неудобно реализовывать сложение. Примеры:

Разряды	7	6	5	4	3	2	1	0	Число
Биты	0	1	1	0	0	0	1	1	01100011b
Число	+	$64=2^6$	$32=2^5$	0	0	0	$2=2^1$	$1=2^0$	+99 = 64 + 32 +2 + 1
Биты	1	1	1	0	0	0	1	1	11100011b

Число	-	$64=2^6$	$32=2^5$	0	0	0	$2=2^1$	$1=2^0$	-99 = -(64 + 32 +2 + 1)

Дополнительный код: 0 в старшем разряде соответствует положительным числам, 1 – отрицательным. Дополнительный код положительного числа, есть само число. Дополнительный код отрицательного числа образуется путем инверсии всех битов положительного числа (включая знаковый) и прибавления 1. В дополнительном коде удобно выполнять операции сложения – числа со знаком складываются точно так же, как беззнаковые. Обратное преобразование производится точно по тому же правилу. Дополнительное преимущество дополнительного кода: +0 и -0 имеют одинаковый код 000b. Примеры:

Разряды	7	6	5	4	3	2	1	0	Число
Биты	0	1	1	0	0	0	1	1	01100011b
Число	+	$64=2^6$	$32=2^5$	0	0	0	$2=2^1$	$1=2^0$	+99 = 64 + 32 +2 + 1
Биты	1	0	0	1	1	1	0	1	10011101b
Число	-	$64=2^6$	$32=2^5$	0	0	0	$2=2^1$	$1=2^0$	-99 = -(64 + 32 +2 + 1)

Обратный код: 0 в старшем разряде соответствует положительным числам, 1 – отрицательным. Обратный код положительного числа, есть само число. Совпадает с прямым кодом. Обратный код отрицательного числа образуется путем вычитания символа каждого разряда (включая знаковый) из числа, на 1 меньшего основания системы счисления. Обратное преобразование производится точно по тому же правилу. Практически не применяется.

Смещенный код: 1 в старшем разряде соответствует положительным числам, 0 – отрицательным. Представления чисел получаются путем прибавления к ним константы $2^N–1$, где N – число двоичных разрядов (не считая знакового). Применяется для кодирования вещественных чисел (с дробной частью) в формате с плавающей точкой.

- 8-ричные чила

Двоичная система счисления, в которой работают все цифровые электронные устройства, неудобна для человека. Для удобства представления двоичного содержимого ячеек памяти или регистров процессора используется позиционая 8-ричная система счисления. Ее алфавит состоит из 8 арабских цифр от 0 до 7. Пример 8-ричного числа:

$(247)_8 = 2*8^2 + 4*8^1 + 7*8^0 = 128 + 32 + 7 = 167.$

В программах 8-ричные числа завершаются суффиксом o. В примере это 247o.

Переход от десятичных чисел к 8-ричным и обратно производится аналогично случаю двоичных чисел. Преобразование двоичных чисел в 8-ричные производится следующим образом:

- Двоичное число, начиная с младших разрядов, разбивается на триады (тройки символов). Если длина числа не кратна трём, то оно дополняется старшими нулевыми разрядами.
- Каждая триада записывается символами 8-ричного алфавита.

При обратном переходе каждый символ представляется 3-разрядным двоичном числом. Единая последовательность этих чисел и представляет собой искомое двоичное число.Знак 8-ричного числа отображается в старшем разряде: 0 соответствует положительному числу, 1 – отрицательному.

8-ричные числа сейчас почти не применяются, так как в большинстве процессоров адресуются байты, в котоых 8 позиций (не кратно 3).

- 16-ричные чила

Двоичная система счисления, в которой работают все цифровые электронные устройства, неудобна для человека. Для удобства представления двоичного содержимого ячеек памяти или регистров процессора используется позиционая 16-ричная система счисления. Ее алфавит состоит из 10 арабских цифр от 0 до 9 и шести латинских букв: A (вес 10), B (вес 11), C (вес 12), D (вес 13), (вес 14), F (вес 15). Пример 16-ричного числа:

$(524D)_{16} = 5*16^3 + 2*16^2 + 4*16^1 + 13*16^0 = 20480 + 512 + 64 + 13 = 21069.$

В программах 16-ричные числа завершаются суффиксом h. В примере это 524Dh.

То же самое число в двоичном представлении содержит в 4 раза больше символов – 0101 0010 0100 1101. !6-ричное представление более компактно.

Переход от десятичных чисел к 16-ричным и обратно производится аналогично случаю двоичных чисел. Преобразование двоичных чисел в 16-ричные производится следующим образом:

- Двоичное число, начиная с младших разрядов, разбивается на тетрады (четвёрки символов). Если длина числа не кратна четырём, то оно дополняется старшими нулевыми разрядами.
- Каждая тетрада записывается символами 16-ричного алфавита.

При обратном переходе каждый символ представляется 4-разрядным двоичном числом. Единая последовательность этих чисел и представляет собой искомое двоичное число.Знак 16-ричного числа отображается в старшем разряде: 0 соответствует положительному числу, 1 – отрицательному.

В программах 16-ричные эквиваленты сопровождаются суффиксом h.

2.2. Форматы представления чисел

2.2.1. Форматы представления двоичных чисел

Формат чисел определяет возможную длину (количество битов) и форму представления чисел. С точки зрения длины представления чисел различают:

- Полубайт (Нибл). Содержит 4 бит. Отображает содержимое половинки байта. Применяется, например, для зпоминания двоичного кода одной десятичной цифры.
- Байт. Содержит 8 бит. Отображает содержимое одной из 8-разрядных ячеек памяти или одного из 8-разрядных регистров. Это минимальный размер адресуемой в МП ячейки памяти. Побитовая адресация прямо не применяется, так как длина адреса окажется непомерно большой.
- Слово. Содержит 2 байта, 16 бит. Отображает содержимое одной из 16-разрядных ячеек памяти или одного из 16-разрядных регистров.
- Двойное слово. Содержит 2 слова, 4 байта, 32 бит. Отображает содержимое 32-разрядных ячеек памяти или регистров, поэтому характеризует представление чисел с удвоенной точностью.
- Учетверенное слово. Содержит 2 двойных слова, 4 слова, 8 байт, 64 бит. Отображает содержимое 64-разрядных ячеек памяти или регистров, поэтому характеризует представление чисел с повышенной точностью.

Двум алгебраическим формам записи вещественных чисел – обычной и показательной – различают две формы представления чисел:

- с фиксированной точкой (ФТ), например, 12.34 – обычное представление вещественного числа..
- с плавающей точкой (ПТ), например, 1.234 E 2. Это представление числа 12.34 в показательной форме: $1.234 * 10^2$. 1.234 – значащая часть (или мантисса), E – разделитель, 2 – порядок.
- Формат с фиксированной точкой

Применяемые термины:

- MSB (Most Significant Bit) – наиболее значащий бит.
- LSB (Least Significant Bit) – наименее значащий бит.

В формате с фиксированной точкой в представлении данных в поле числа присутствует логическая позиция точки (бита точки нет, он логически подразумевается), задающая начало или конец значащей части. Возможны варианты:

Число целое со знаком. Бит знака S размещается в MSB. Значащие биты выравниваются по правому краю формата. Логическая точка справа от LSB. Например, для 8-и разрядного процессора двоичное целое число 1101. Его десятичный эквивалент 8+4+1 = 13.

Биты	7	6	5	4	3	2	1	0
Значения	0	0	0	0	1	1	0	1
	MSB=S							LSB

Число целое без знака. Бит знака S=0 по умолчанию. Значащие биты начинаются с MSB (Most Significant Bit). При одинаковом N число битов значащей части в 2 раза больше. Значащие биты выравниваются по правому краю формата. Логическая точка справа от LSB. Например, для 8-и разрядного процессора двоичное целое число 101. Его десятичный эквивалент 4+1 = 5.

Биты	7	6	5	4	3	2	1	0
Значения	0	0	0	0	1	1	0	1
	MSB							LSB

Число дробное. Значащие биты выравниваются по левому краю формата Логическая точка справа от бита знака S. Например, для 8-и разрядного процессора двоичное дробное число 0.101. Его десятичный эквивалент 0.5+0.125 = 0.625.

Биты	7	6	5	4	3	2	1	0
Значения	0	1	0	1	0	0	0	0
	MSB=S							LSB

Число дробное без знака. Значащие биты начинаются с MSB. При одинаковом N число битов значащей части в 2 раза больше. Значащие биты выравниваются по левому краю формата Логическая точка слева от бита знака S. Например, для 8-и разрядного процессора двоичное дробное число 0.0101. Его десятичный эквивалент 0.25+0.0625 = 0.3125.

Биты	7	6	5	4	3	2	1	0
Значения	0	1	0	1	0	0	0	0
	MSB							LSB

В процессорах с ФТ, как правило, используются только дробные числа. Смешанные числа могут появляться только в промежуточных вычислениях.

2.2.2. Формат с плавающей точкой

Формат с плавающей точкой предназначен для компактного отображения вещественных чисел в очень широком диапазоне. Число представляется в алгебраическом формате: $(S)(F)*2^P$, где

- S (Sign) – знак числа. Для положительного S = 0, для отрицательного S = 1.
- F (Fraction) - мантисса (значащая часть).
- P (Power) – порядок.

В 1985 институт инженеров по электротехнике и электронике IEEE (Institute of Electrical and Electronics Engineers) в США разработал стандарт представления чисел с плавающей точкой IEEE 754. Согласно этому стандарту слово данных разбивается на три поля.

- **Однобитовое поле S** (sign - знак) используется для указания знака числа. Для положительного числа S = 0, для отрицательного S = 1.
- **Поле F (fraction)**. В нем записывается дробная часть мантиссы (fraction). Мантисса наряду с дробной частью содержит целую часть (1 или 0). Бит целой части мантиссы в памяти не хранится для уменьшения объема запоминаемых данных, при отображении данных он автоматически учитывается.
- **Поле экспоненты (E – exponent),** содержит смещённый порядок E=P+Bias. Biass – смещение, выбирается так, чтобы смещённый порядок был положительным или равным нулю.

Если целая часть мантиссы равна единице, то число считается нормализованным, а если она равна нулю, то ненормализованным. Целая часть мантиссы считается равной нулю, только в том случае, когда смещённый порядок числа также равен нулю. Во всех остальных случаях целая часть мантиссы равна единице.

В зависимости от точности представления форма с плавающей точкой имеет 3 стандарта:

- С одинарной точностью SP (Single Precision floating-point format).
- С двойной точностью DP (Double Precision floating-point format).
- С расширенной одинарной точностью формат SEP (Single Extended Precision floating-point format). Это формат для представления результатов промежуточных и конечных вычислений с расширенной одинарной точностью. Применяется для данных, которые не могут быть представлены в формах SP или DP.

Сравнительные данные форм в стандарте IEEE 754:

Параметр	SP	DP	SEP
Длина	32	64	44
Знак числа	1	1	1
Мантисса, всего	24	53	32
Мантисса, дробная часть	23	52	31
Мантисса, целая часть	Неявная 1	Неявная 1	Явная 1
Смещенный порядок E	8	11	11
Смещение Biass	127	1023	1023
Порядок Рмин	-126	-1022	-1022
Емин	1	1	1
Порядок Рмакс	127	1023	1023
Емакс	254	2046	2046

Пример. Положительное двоичное число в формате с плавающей точкой SP.

0	01111100	10000000000000000000000
Знак, 1 бит	Смещенный порядок, 8 бит	Мантисса, дробь – 23 бита

Определим десятичный эквивалент этого числа. Количество разрядов смещенного порядка E=8, величина смещения Biass=127. Десятичный эквивалент смещённого порядка равен $2^6+2^5+2^4+2^3+2^2$ = 124. Следовательно, порядок P (несмещённый) двоичного числа P=124-127= -3.

Десятичный эквивалент дробной части мантиссы равен 0.5. Так как смещённый порядок больше нуля, то скрытая целая часть мантиссы равна единице. Следовательно, десятичный эквивалент мантиссы равен 1.5. Поскольку знаковый разряд равен нулю, число положительно. Окончательно получим

$N_{10} = 1.5*2^{-3} = 1.5/8 = 0.1875$.

Стандарт IEEE 754 поддерживает представление специальных данных.

Тип	E	Мантисса, дробь	Значение
Нуль	0000 0000	Нуль	0
Бесконечность	1111 1111	Нуль	∞
Не число	1111 1111	Не нуль	Не число

Представление двоичных вещественных чисел в форме с ФТ означает, что как для целой, так и дробной части отведено фиксированное число разрядов. То есть местоположение точки, отделяющей целую и дробную части числа, всегда одинаково. Представление двоичных целых чисел в форме с ФТ означает, что точка по существу отсутствует. Достоинством формы с ФТ является простота реализации арифметических операций, а недостатком – ограниченный динамический диапазон. Динамическим диапазоном называют отношение самого большого к самому малому по модулю (но отличным от нуля) чисел, которые можно представить с помощью данной формы. Для формы с ФТ это отношение равно $2^{n-1}-1$.

В общем случае двоичные вещественные числа в форме с ПТ представляются в виде $x = \mu \cdot 2^P$, где μ – мантисса (вещественное двоичное число со знаком, представленное в форме с ФТ); P – порядок (целое двоичное число со знаком); 2 – основание двоичной системы счисления. Однако присутствие двух параметров μ и P приводят к неоднозначности представления чисел: одно и то же число можно представить по-разному, например $2 = 2 \cdot 2^0$ или $2 = 1 \cdot 2^1$ и т.д.

Поэтому и для упрощения арифметики чисел с ПТ применяют нормализованные формы чисел с ПТ. В цифровой технике часто используется нормализованная форма, в которой целая часть мантиссы всегда равна нулю, а первая значащая цифра дробной части отлична от нуля. То есть мантисса ограничена значениями $0{,}5 \le |\mu| < 1$.

Целочисленная и дробная арифметики

Сложение двух чисел любой позиционной системы счисления производится по единому правилу:

- сложение производится поразрядно, начиная с младших разрядов;
- если сумма Si чисел в i-м разряде превышает или равна основанию E системы счисления, то в этот разряд записывается разность Si–E, а в следующий, более старший разряд, переносится 1 в виде дополнительного слагаемого.

Вычитание чисел с целью упрощения технической реализации заменяется сложением. Для этого вычитаемое представляется в дополнительном коде и результат складывается с уменьшаемым.

Операции сложения и вычитания над двумя числами с ПТ выполняются в следующей последовательности:

- Выравнивание порядков. В качестве общего выбирается больший порядок, мантисса числа с меньшим порядком логически сдвигается вправо на количество разрядов, равное разности порядков чисел.
- Производится требуемая операция.
- Нормализация результата, т.е. выбор такого значения порядка, при котором старший разряд мантиссы имеет единичное значение.

2.3. Типы адресаций операндов

Адресацией называется обращение к операнду (число, участвующее в операции), указание на который содержится в команде. Операнды, в зависимости от места своего хранения, могут указываться разными способами, которым соответствуют разные типы адресации, или, коротко, разные адресации.

При описании различных адресаций операндов используют понятия адресного кода и исполнительного адреса. Адресный код АК – это информация об адресе операнда, содержащаяся в команде. Исполнительный адрес АИ – это номер физической ячейки памяти, к которой производится обращение.

Первая группа адресаций устанавливает $А_И$ по значению $А_К$. Сюда входят:

- **Непосредственная адресация**. Операнд указывается в команде константой. Эта адресация используется только для указания исходных данных.
- **Прямая адресация**. АИ совпадает с АК.
- **Регистровая адресация**. В команде указывается имя регистра процессора, в котором хранится операнд.
- **Косвенная адресация**. Используется в целях сокращения длины команды. В этом случае АК указывает имя регистра процессора, в котором находится АИ. Такой регистр называют регистром адреса..
- **Автоинкрементная и автодекрементная адресация**. В команде указывается имя регистра процессора, содержимое которого автоматически увеличивается (уменьшается) на 1, причем изменение адреса может производиться как до (преинкремент / предекремент), так и после (постинкремент / постдекремент) выполнения основной команды. Следовательно, преинкре-

мент / предекремент означает вычисление нового АИ перед выполнением команды, а постинкремент / постдекремент – что АИ в данной команде не изменяется.

Вторая группа адресаций устанавливает $А_И$ по $А_К$ и содержимому регистров процессора. Сюда входят индексная и базовая адресации. Оба типа адресаций позволяют при меньшей длине адресного кода команды обеспечить доступ к любой ячейке памяти.

- **Индексация** (указание) означает автоматическое изменение АИ без изменения содержимого регистра адреса, называемого индексом, причем АИ вычисляется как алгебраическая сумма содержимого индекса и смещения. Таким образом, содержимое индекса задает начало некоторой области ячеек памяти, а смещение – конкр
- етную ячейку памяти в этой области. В команде АК указывает имя индекса и сравнительно короткое смещение или имя регистра процессора, в котором оно содержится. Индексация используется при работе с массивами данных.
- **Базирование** является развитием индексации. Здесь АИ также определяется алгебраической суммой содержимого регистра адреса, называемого базой, и смещения, но с изменением содержимого базы. При этом используются постинкремент / постдекремент и преинкремент / предекремент на величину смещения.

2.4. Интерфейсы

Интерфейс (Interface = Inter face – между лицами) опеределяет правила взаимодействия компонент и модулей системы. Типы интерфейсов:

- Последовательный. Биты данных передаются последовательно во времени по одному каналу.
- Параллельный. Данные передаются группами битов, для каждого бита свой канал.
- Инфракрасный. Данные передаются последовательно с использованием канала с инфракрасным лучом.
- Bluetouth. Данные передаются последовательно с использованием радиоканала.
- USB. Представляет собой шину, по которой к периферийному устройству подводится питание и осуществляется двунаправленный побитовый обмен данными.
- Последовательный интерфейс

2.4.1. Последовательный интерфейс RS-232C

Стандарт был опубликован в 1969. Ассоциацией электронной промышленности (EIA). Первоначально этот интерфейс использовался для подключения ЭВМ и терминалов к системе связи через модемы, а также для непосредственного подключения терминалов к машинам. До недавнего времени последовательный интерфейс использовался для широкого спектра периферийных устройств (плоттеры, принтеры, мыши, модемы и др.), но сейчас активно вытесняется интерфейсом USB.

Стандарт RS-232C определяет:

- механические характеристики интерфейса - разъемы и соединители;
- электрические характеристики сигналов - логические уровни;
- функциональные описания интерфейсных схем - протоколы передачи;
- стандартные интерфейсы для выбранных конфигураций систем связи.

В 1975 были приняты стандарты RS-422 (электрические характеристики симметричных цепей цифрового интерфейса) и RS-423 (электрические характеристики несимметричных цепей цифрового интерфейса), позволяющие увеличить скорость передачи данных по последовательному интерфейсу.

Обычно ПК имеют в своем составе два интерфейса RS-232C, которые обозначаются COM1 и COM2. Возможна установка дополнительного оборудования, которое обеспечивает функционирование в составе PC четырех, восьми и шестнадцати интерфейсов RS-232C. Для подключения устройств используется 9-контактный (DB9) или 25-контактный (DB25) разъем.

Интерфейс RS-232C содержит сигналы квитирования, обеспечивая асинхронный режим функционирования. При этом одно из устройств (обычно компьютер) выступает как DTE (Data Terminal Equipment - оконечное устройство), а другое - как DCE (Data Communication Equipment - устройство передачи данных), например, модем. Соответственно, если для DTE какой-то сигнал является входным, то для DCE этот сигнал будет выходным, и наоборот.

2.4.2. Интерфейс параллельного порта

Стандартный интерфейс параллельного порта получил свое первоначальное название по имени американской фирмы Centronics - производителя принтеров. Первые версии этого стандарта были ориентированы исключительно на принтеры, подразумевали передачу данных лишь в одну сторону (от компьютера к принтеру) и имели невысокую скорость передачи (150-300 Кбайт/с).

Такие скорости неприемлемы для современных печатающих устройств. Кроме того, для работы с некоторыми устройствами необходима двусторонняя передача данных. Поэтому некоторые фирмы (Xircom, Intel, Hewlett Packard, Microsoft) предложили несколько модификаций скоростных параллельных интерфейсов. На основе этих разработок в 1994 году Институтом инженеров по электронике и электротехнике был принят стандарт IEEE 1284, ныне повсеместно используемый в персональных компьютерах в качестве стандартного параллельного интерфейса.

Стандарт IEEE 1284 определяет работу параллельного интерфейса в трех режимах:

- Standard Parallel Port (SPP),
- Enhanced Parallel Port (EPP),
- Extended Capabilities Port (ECP).

Каждый из этих режимов предусматривает двустороннюю передачу данных между компьютером и периферийным устройством.

Режим SPP (Стандартный параллельный порт) используется для совместимости со старыми принтерами, не поддерживающими IEEE 1284.

В режиме EPP (Улучшенный параллельный порт) используется аппаратная реализация сигналов квитирования, благодаря чему возможно увеличение скорости передачи до 2 Мбайт/с.

Режим ECP (Порт расширенных возможностей) также использует аппаратное квитирование и адресацию устройств (до 128). Дополнительно ECP поддерживает распознавание ошибок, согласование скорости и режима передачи, буферизацию данных в очереди FIFO (с использованием DMA) и их компрессию по алгоритму RLE (Run Length Encoding), что позволяет достигать скорость до 4 Мбайт/с.

2.4.3. Инфракрасный интерфейс

В 1994 Ассоциацией инфракрасной передачи данных (Infra-Red Data Assotiation) была принята первая версия стандарта IrDA. Интерфейс IrDA позволяет соединяться с периферийным оборудованием без кабеля при помощи ИК-излучения с длиной волны 850-900 нм (номинально - 880 нм). Порт IrDA дает возможность устанавливать связь на коротком расстоянии до 1 метра в режиме "точка-точка". В цели интерфейса входили низкое ресурсопотребление и экономичность.

Порт IrDA основан на архитектуре коммуникационного порта и использует универсальный асинхронный приемо-передатчик UART (Universal Asynchronous Receiver Transmitter), позволяющий работать со скоростью передачи данных 2400-115200 бит/с. Данные передаются 10-битными символами: 8 бит данных, один стартовый бит в начале и один стоповый бит в конце посылки. Связь в IrDA полудуплексная, т.к. передаваемый ИК-луч неизбежно засвечивает приемный фотодиод.

В мобильных телефонах, notebook, мини-компьютерах, коммуникаторах и смартфонах **инфракрасный порт** позволяет обмениваться различной информацией: визитками, музыкой, картинками, файлами; печатать на принтерах документы и т.д. Но у инфракрасного порта есть свои недостатки: он работает только на расстоянии до 1 метра, скорость передачи крайне мала, флуоресцентные лампы и яркий солнечный свет мешают его работе.

2.4.4. Интерфейс Bluetooth

Вместо инфракрасного интерфейса теперь применяют беспроводной стандарт **Bluetooth**, который позволяет по радиоканалу передать музыку, картинки, файлы. Bluetooth имеет дальность передачи до 10 метров, не требуется прямая видимость (другое устройство может быть за стенкой). Название Bluetooth было дано в честь датского короля X-го века Гаральда II Блатана (Блатан, по-датски - Синий Зуб - Blue Tooth, по-английски), прославившегося своей способностью находить общий язык с князьями-вассалами.

2.4.5. Интерфейс USB

Спецификация периферийной шины USB была разработана лидерами компьютерной и телекоммуникационной промышленности (Compaq, DEC, IBM, Intel, Microsoft, NEC и Northern Telecom) для подключения компьютерной периферии вне корпуса ПК с автоматическим автоконфигурированием (Plug&Play). Первая версия стандарта появилась в 1996. Агрессивная политика Intel по внедрению этого интерфейса стимулирует постепенное исчезновение таких низкоскоростных интерфейсов, как RS 232C. Однако для высокоскоростных устройств с более строгими требованиями к производительности (например, доступ к удаленному накопителю или передача оцифрованного видео) конкурентом USB является интерфейс IEEE 1394.

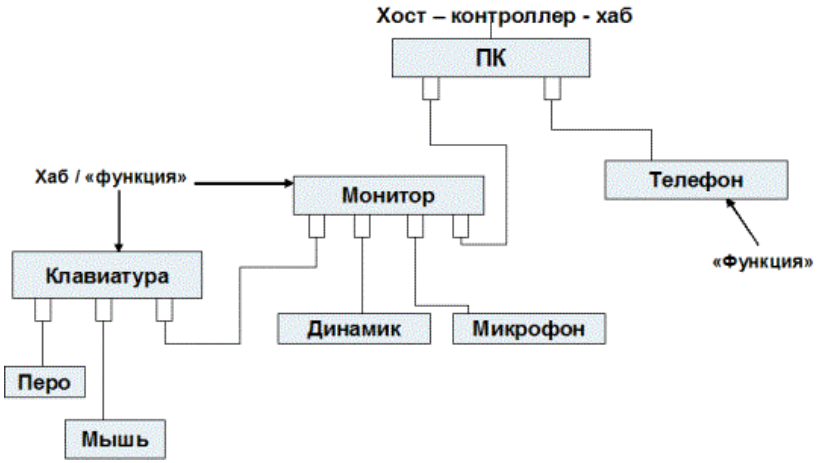
Интерфейс USB представляет собой последовательную, полудуплексную, двунаправленную шину со скоростью обмена:

- USB 1.1 - 1,5 Мбит/с или 12 Мбит/с;

- USB 2.0 - 480 Мбит/с.

Шина позволяет подключить к ПК до 127 физических устройств. Каждое физическое устройство может, в свою очередь, состоять из нескольких логических (например, клавиатура со встроенным манипулятором-трекболом).

Кабельная разводка USB начинается с узла (host). Хост обладает интегрированным корневым концентратором (root hub), который предоставляет несколько разъемов USB для подключения внешних устройств. Затем кабели идут к другим устройствам USB, которые также могут быть концентраторами, и функциональным компонентам (например, модем или акустическая система). Концентраторы часто встраиваются в мониторы и клавиатуры (которые являются типичными составными устройствами). Концентраторы могут содержать до семи "исходящих" портов.



Для передачи сигналов шина USB использует 4-проводной интерфейс. Одна пара проводников ("+5В" и "общий") предназначена для питания периферийных устройств с нагрузкой до 500 мА. Данные передаются по другой паре ("D+" "D").

Все концентраторы должны поддерживать на своих исходящих портах устройства обоих типов, не позволяя высокоскоростному трафику достигать низкоскоростных устройств. Высокопроизводительные устройства подключаются с помощью экранированного кабеля, длина которого не должна превышать 3 м. Если же устройство не формулирует особых требований к полосе пропускания, его можно подключить и неэкранированным кабелем (который может быть бо-

лее тонким и гибким). Максимальная длина кабеля для низкоскоростных устройств - 5 м.

Хост узнает о подключении или отключении устройства из сообщения от концентратора (эта процедура называется опросом шины - bus enumeration). Затем хост присваивает устройству уникальный адрес USB (1:127). После отключения устройства от шины USB его адрес становится доступным для других устройств.

Хост опрашивает все устройства и выдает им разрешения на передачу данных (рассылая для этого пакет-маркер - Token Packet). Таким образом, устройства лишены возможности непосредственного обмена данными - все данные проходят через хост. Это условие сильно мешало внедрению интерфейса USB на рынок портативных устройств. В результате в конце 2001 года было принято дополнение к стандарту USB 2.0 - спецификация USB OTG (On-The-Go), предназначенная для соединения периферийных USB-устройств друг с другом без необходимости подключения к хосту (например, цифровая камера и фотопринтер). Устройство, поддерживающее USB OTG, способно частично выполнять функции хоста и распознавать, когда оно подключено к полноценному хосту (на основе ПК), а когда - к другому периферийному устройству.

2.4.6. Интерфейс IEEE 1394 - FireWire

Группой компаний при активном участии Apple была разработана технология последовательной высокоскоростной шины, предназначенной для обмена цифровой информацией между компьютером и другими электронными устройствами. В 1995 эта технология была стандартизована IEEE (стандарт IEEE 1394). Компания Apple продвигает этот стандарт под торговой маркой FireWire, а компания Sony - под торговой маркой i-Link.

Интерфейс IEEE 1394 представляет собой дуплексную, последовательную, общую шину для периферийных устройств. Она предназначена для подключения компьютеров к таким бытовым электронным приборам, как записывающая и воспроизводящая видео- и аудиоаппаратура, а также используется в качестве интерфейса дисковых накопителей.

Первоначальный стандарт (1394a) поддерживает скорости передачи данных 100 Мбит/с, 200 Мбит/с и 400 Мбит/с. Последующие усовершенствования стандарта (1394b) обеспечивают поддержку скорости передачи данных 800 и 1600 Мбит/с (FireWire-800, FireWire-1600).

Устройства, которые передают данные на разных скоростях, могут быть одновременно подключены к кабелю (поскольку пары обменивающихся данными

устройств используют для этого одну и ту же скорость). Рекомендуемая максимальная длина кабеля между устройствами составляет 4,5 м. К кабелю общей длиной до 72 м может быть одновременно подключено до 63 устройств, называемых узлами (nodes). Для увеличения числа шин вплоть до максимального значения (1023) могут быть использованы мосты.

Каждое устройство обладает 64-разрядным адресом:

- 6 бит - идентификационный номер устройства на шине,
- 10 бит - идентификационный номер шины,
- 48 бит - используются для адресации памяти (каждое устройство может адресовать до 256 Тбайт памяти).

Шина предполагает наличие корневого узла, выполняющего некоторые функции управления. Корневой узел может быть выбран автоматически во время инициализации шины, либо его атрибут может быть принудительно присвоен конкретному узлу (скорее всего, ПК). Некорневые узлы являются или ветвями (если они поддерживают более чем одно активное соединение), или листьями (если они поддерживают только одно активное соединение).

Как правило, устройства имеют по 1-3 порта, причем одно устройство может быть включено в любое другое (с учетом ограничений на то, что между любыми двумя устройствами может быть не более 16 пролетов и они не могут быть соединены петлей). Допускается подключение в "горячем" режиме, поэтому устройства могут подключаться и отключаться в любой момент. При подключении устройств адреса назначаются автоматически, поэтому присваивать их вручную не придется.

IEEE 1394 поддерживает два режима передачи данных (каждый из которых использует пакеты переменной длины).

- Асинхронная передача используется для пересылки данных по конкретному адресу с подтверждением приема и обнаружением ошибок. Трафик, который не требует очень высоких скоростей передачи данных и не чувствителен ко времени доставки, вполне подходит для данного режима (например, для передачи некоторой управляющей информации).
- Изохронная передача предполагает пересылку данных через равные промежутки времени, причем подтверждения приема не используются. Этот режим предназначен для пересылки оцифрованной видео- и аудиоинформации.

Пакеты данных пересылаются порциям, которые имеют размер, кратный 32 битам, и называются квадлетами (guadlets). При этом пакеты начинаются, по меньшей мере, с двух квадлетов заголовка, после чего следует переменное число квадлетов полезной информации. Для заголовка и полезных данных контрольные суммы (CRC) указываются отдельно. Длина заголовков асинхронных пакетов составляет, как минимум, 4 квадлета. У изохронных пакетов может быть заголовок длиной 2 квадлета, поскольку единственным необходимым при этом адресом является номер канала.

IEEE 1394 выделяет следующие функции устройств:

- Хозяин цикла (cycle master) - выполняется корневым узлом, имеет наивысший приоритет доступа к шине, обеспечивает общую синхронизацию остальных устройств на шине, а также изохронных сеансов передачи данных.
- Диспетчер шины (bus manager) управляет питанием шины и выполняет некоторые функции оптимизации.
- Диспетчер изохронных ресурсов (isochronous resource manager) распределяет временные интервалы среди узлов, собирающихся стать передатчиками (talkers).

Все функции диспетчеризации могут выполняться одним и тем же либо различными устройствами. Хозяин цикла посылает синхронизирующее сообщение о начале цикла через каждые 125 мкс (как правило). Теоретически 80% цикла (100 мкс) резервируется для изохронного трафика, а остальная часть становится доступной для асинхронного трафика.

Для подключения к данному интерфейсу применяется 6-контактный соединитель. Используемый при этом кабель имеет круглую форму и содержит:

- экранированную витую пару A (TPA), в которой используется симметричное, разностное напряжение (для обеспечения требуемой помехоустойчивости), а данные передаются в обоих направлениях с помощью схемы кодирования;
- экранированную витую пару B (TPB), пересылающую стробирующий сигнал, который изменяет состояние всякий раз, когда два последовательных разряда данных (на другой паре) одинаковы (т.н. кодирование данных со стробированием - data-strobe encoding), и гарантирует изменение состояния в паре для передачи данных либо стробирующих сигналов по фронту каждого разряда;

- провода, обеспечивающие питание небольших устройств. При этом по проводу VP подается напряжение 8-40 В, обеспечивающее нагрузку до 1,5 А, а провод VG заземлен;
- общий экран, который изолирован от экранов пар и прикреплен к корпусам соединителей.

В IEEE 1394b допускается применять также простые UTP-кабели 5-й категории, но только на скоростях до 100 Мбит/с. Для достижения максимальных скоростей на максимальных расстояниях предусмотрено использование оптоволокна (пластмассового - для длины до 50 метров, и стеклянного - для длины до 100 метров).

2.4.7. Сопроцессоры

Сопроцессор - это специализированная интегральная схема, которая работает в содружестве с ЦП, но менее универсальна. В отличие от ЦП, сопроцессор не имеет счетчика команд. Сопроцессор предназначен для выполнения специфического набора функций, например: выполнение операций с вещественными числами - математический сопроцессор, подготовка графических изображений и трехмерных сцен - графический сопроцессор, цифровая обработка сигналов - сигнальный сопроцессор и др.

Использование сопроцессоров с различной функциональностью позволяет решать проблемы широкого круга:

- обработка экономической информации;
- моделирование;
- графические преобразования;
- промышленное управление;
- системы числового управления;
- роботы;
- навигация;
- сбор данных и др.
- Кэш-память

Кэш-память представляет собой быстродействующее ЗУ, размещенное на одном кристалле с ЦП или внешнее по отношению к ЦП. Кэш служит высокоскоростным буфером между ЦП и относительно медленной основной памятью. Идея кэш-памяти основана на прогнозировании наиболее вероятных обращений ЦП к оперативной памяти.

Если ЦП обратился к какому-либо объекту оперативной памяти, то с высокой вероятностью он вскоре снова обратится к этому объекту. Примером этой ситуации может быть код или данные в циклах.

Для согласования содержимого кэш-памяти и оперативной памяти используют три метода записи:

- Сквозная запись (write through) - одновременно с кэш-памятью обновляется оперативная память.
- Буферизованная сквозная запись (buffered write through) - информация задерживается в кэш-буфере перед записью в оперативную память и переписывается в оперативную память в те циклы, когда ЦП к ней не обращается.
- Обратная запись (write back) - используется бит изменения в поле тега, и строка переписывается в оперативную память только в том случае, если бит изменения равен 1.

В структуре кэш-памяти выделяют два типа блоков данных:

- память отображения данных (собственно сами данные, дублированные из оперативной памяти);
- память тегов (признаки, указывающие на расположение кэшированных данных в оперативной памяти).

Пространство памяти отображения данных в кэше разбивается на строки - блоки фиксированной длины (например, 32, 64 или 128 байт). Каждая строка кэша может содержать непрерывный выровненный блок байт из оперативной памяти. Какой именно блок оперативной памяти отображен на данную строку кэша, определяется тегом строки и алгоритмом отображения. По алгоритмам отображения оперативной памяти в кэш выделяют три типа кэш-памяти:

- полностью ассоциативный кэш;
- кэш прямого отображения;
- множественный ассоциативный кэш.

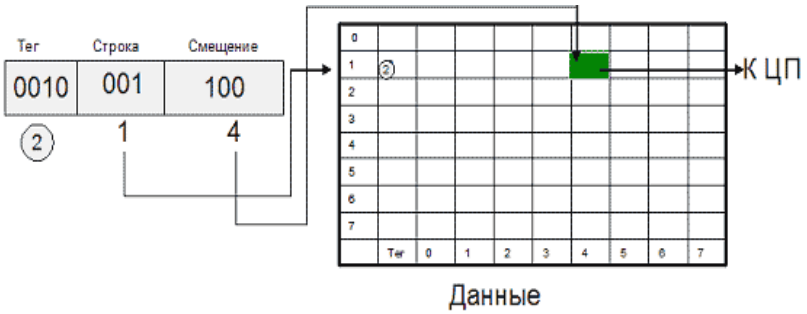
Для полностью ассоциативного кэша характерно, что кэш-контроллер может поместить любой блок оперативной памяти в любую строку кэш-памяти.

В этом случае физический адрес разбивается на две части: смещение в блоке (строке кэша) и номер блока. При помещении блока в кэш номер блока сохраняется в теге соответствующей строки. Когда ЦП обращается к кэшу за необходимым блоком, кэш-промах будет обнаружен только после сравнения тегов всех строк с номером блока.

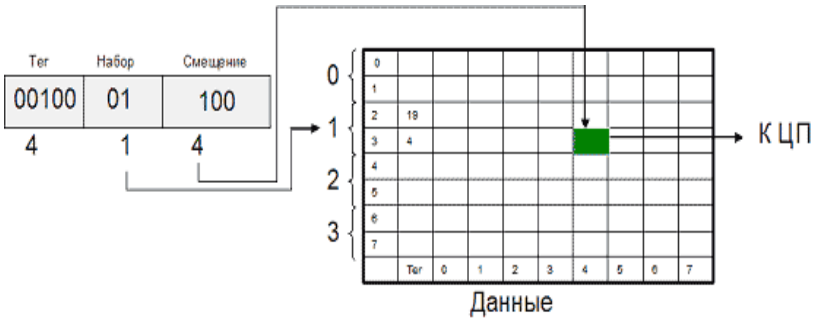
Одно из основных достоинств данного способа отображения - хорошая утилизация оперативной памяти, т.к. нет ограничений на то, какой блок может быть отображен на ту или иную строку кэш-памяти. К недостаткам следует отнести сложную аппаратную реализацию этого способа, требующую большого количества компонент схемотехники (в основном компараторов), что приводит к увеличению времени доступа к такому кэшу и увеличению его стоимости.

Кэш прямого отображения (или одновходовый ассоциативный кэш). В этом случае адрес памяти (номер блока) однозначно определяет строку кэша, в которую будет помещен данный блок. Физический адрес разбивается на три части: смещение в блоке (строке кэша), номер строки кэша и тег. Тот или иной блок будет всегда помещаться в строго определенную строку кэша, при необходимости заменяя собой хранящийся там другой блок. Когда ЦП обращается к кэшу за необходимым блоком, для определения удачного обращения или кэш-промаха достаточно проверить тег лишь одной строки.

Очевидными преимуществами данного алгоритма являются простота и дешевизна реализации. К недостаткам следует отнести низкую эффективность такого кэша из-за вероятных частых перезагрузок строк. Например, при обращении к каждой 64-й ячейке памяти в системе кэш-контроллер будет вынужден постоянно перегружать одну и ту же строку кэш-памяти, совершенно не задействовав остальные.



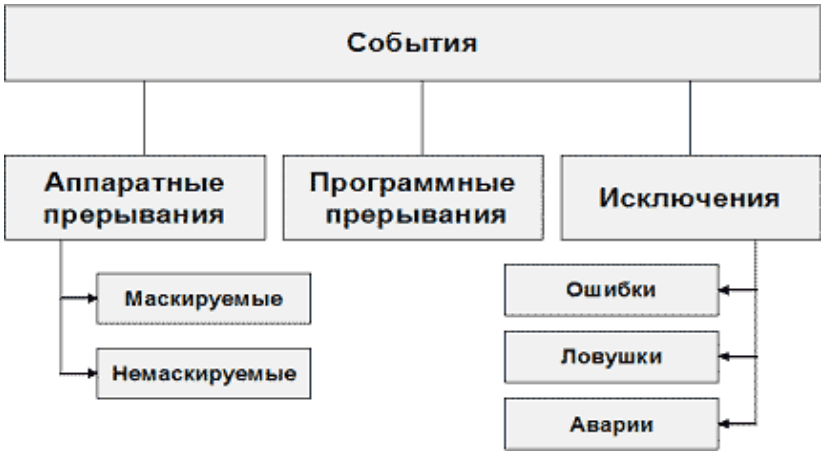
Множественный ассоциативный кэш (или частично-ассоциативный кэш). Это компромиссный вариант между первыми двумя алгоритмами.



При этом способе организации кэш-памяти строки объединяются в группы, в которые могут входить 2/4/8/: строк. В соответствии с количеством строк в таких группах различают 2-входовый, 4-входовый и т.п. ассоциативный кэш. При обращении к памяти физический адрес разбивается на три части: смещение в блоке (строке кэша), номер группы (набора) и тег. Блок памяти, адрес которого соответствует определенной группе, может быть размещен в любой строке этой группы, и в теге строки размещается соответствующее значение. Очевидно, что в рамках выбранной группы соблюдается принцип ассоциативности. С другой стороны, тот или иной блок может попасть только в строго определенную группу, что перекликается с принципом организации кэша прямого отображения. Для того чтобы процессор смог идентифицировать кэш-промах, ему надо будет проверить теги лишь одной группы (2/4/8/: строк).

2.4.8. Система прерываний и исключений

Прерывания и исключения - это события, которые указывают на возникновение в системе или в выполняемой в данный момент задаче определенных условий, требующих вмешательства процессора. Возникновение таких событий вынуждает процессор прервать выполнение текущей задачи и передать управление специальной процедуре либо задаче, называемой обработчиком прерывания или обработчиком исключения. Различные синхронные и асинхронные события в системе можно классифицировать следующим образом:



Прерывания обычно возникают в произвольный момент времени. Прерывания бывают аппаратные (или внешние) и программные.

Внешние прерывания генерируются по аппаратному сигналу, поступающему от периферийного оборудования, когда оно требует обслуживания. Процессор определяет необходимость обработки внешнего прерывания по наличию сигнала на линии прерывания При появлении сигнала на линии INTR# внешний контроллер прерываний должен предоставить процессору вектор (номер) прерывания.

Прерывания всегда обрабатываются на границе инструкций, т.е. при появлении сигнала прерывания процессор сначала завершит выполняемую в данный момент инструкцию (или итерацию при наличии префикса повторения), а только потом начнет обрабатывать прерывание. Помещаемый в стек обработчика адрес очередной инструкции позволяет корректно возобновить выполнение прерванной программы.

Исключения являются для процессора внутренними событиями и сигнализируют о каких-либо ошибочных условиях при выполнении той или иной инструкции. Источниками исключений являются три типа событий:

- генерируемые программой исключения, позволяющие программе контролировать определенные условия в заданных точках программы (проверка на переполнение, контрольная точка, проверка границ массива);

- исключения машинного контроля, возникающие в процессе контроля операций внутри чипа и транзакций на шине процессора;
- обнаруженные процессором ошибки в программе (деление на ноль, отсутствие страницы и т.п.)

Все прерывания и исключения имеют номер (иногда именуемый вектором) в диапазоне от 0 до 255.

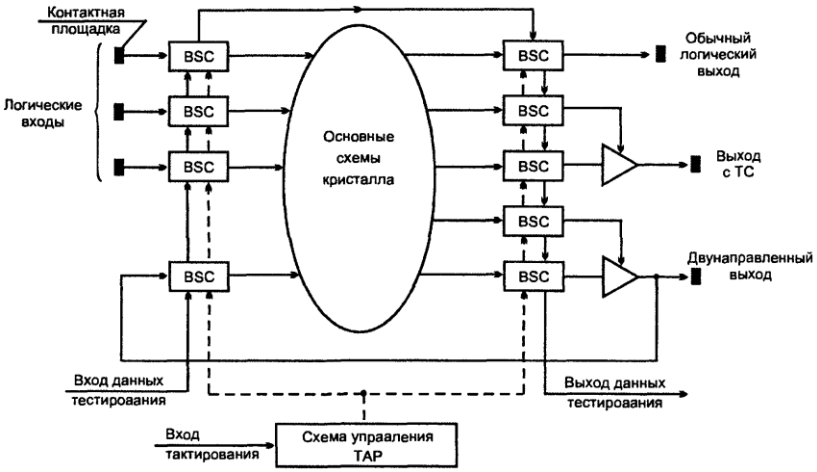
2.4.9. Интерфейс JTAG

Термином JTAG обозначают совокупность средств и операций, позволяющих проводить тестирование БИС/СБИС без физического доступа к каждому ее выводу. Аббревиатура JTAG возникла по наименованию разработчика - объединенной группы по тестам Joint Test Action Group. Термином "периферийное сканирование" (ПС) или по-английски Boundary Scan Testing (BST) называют тестирование по JTAG стандарту (IEEE Std 1149.1).

Такое тестирование возможно только для микросхем, внутри которых имеется набор специальных элементов - ячеек периферийного сканирования (ячеек ПС), в английской терминологии BSC (Boundary Scan Cells) и схем управления их работой.

Ячейки BSC размещены между каждым внешним выводом микросхемы и схемами кристалла, образующими само проверяемое устройство. Все большее число современных микросхем снабжается интерфейсом JTAG, т. е. возможностями периферийного сканирования.

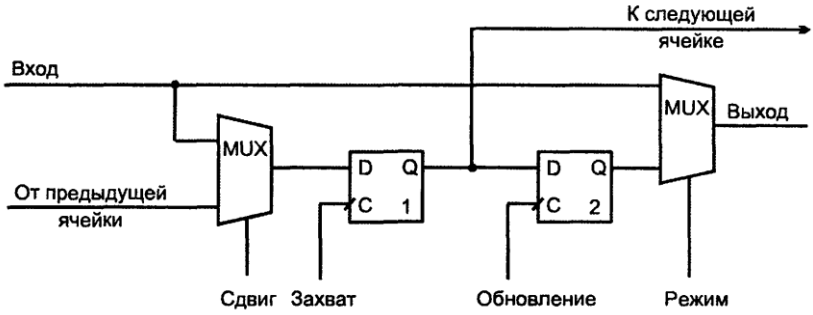
Основная концепция периферийного сканирования иллюстрируется рисунком.



Ячейки сканирования BSC могут работать в разных режимах. В рабочем режиме они просто пропускают сигналы через себя слева направо и не изменяют функционирования устройства. При этом для выходов обычного логического типа нужна одна BSC, для выходов с третьим состоянием - две (вторая для выработки сигнала управления буфером), для двунаправленных выводов - три. Входные сигналы проходят через ячейки BSC прямо к соответствующим точкам основных схем кристалла.

В режиме тестирования пропуск сигналов через ячейки прекращается, а сами они, соединяясь последовательно, образуют сдвигающий регистр, обладающий также некоторыми дополнительными функциями. В такой сдвигающий регистр со входа данных тестирования может быть введен тестовый код для подачи на входные точки основной схемы кристалла. Результат, который выработает основная схема, загружается в ячейки BSC на ее выходах и затем выводится последовательно для сравнения с ожидаемым правильным результатом вне устройства.

Схема BSC содержит два мультиплексора MUX и два D-триггера.



В зависимости от адресного входа "Режим" выходного мультиплексора, ячейка либо свободно пропускает сигнал со входа на выход, либо передает на выход состояние второго триггера. Адресный сигнал входного мультиплексора "Сдвиг" управляет подачей на первый триггер входного сигнала (от логических входов микросхемы) или же сигнала от предыдущей ячейки. Таким образом, по синхросигналу для первых триггеров и при передаче через входной мультиплексор сигнала "Вход" осуществляется параллельная загрузка этих триггеров во всех ячейках. При передаче через входной мультиплексор сигнала от предыдущей ячейки тактовый сигнал производит сдвиг на один разряд в регистре, образованном последовательным соединением ячеек.

По синхросигналу "Обновление" текущее содержимое регистра, составленного из цепочки первых триггеров, переписывается в статический регистр, составленный из вторых триггеров. Сдвиги в регистре на триггерах 1 не будут влиять на содержимое регистра на триггерах 2.

Периферийное сканирование позволяет проверять работу самих микросхем, монтажные межсоединения микросхем между собой на печатной плате, считывать сигналы на выводах микросхемы во время ее работы или управлять этими сигналами.

3. Архитектура CISC от Intel

3.1. Введение

CISC (Complete Instruction Set Computing) – компьютер с полным набором команд. CISC характеризуется следующими свойствами:

- нефиксированное значение длины команды;
- арифметические действия кодируются в одной команде;

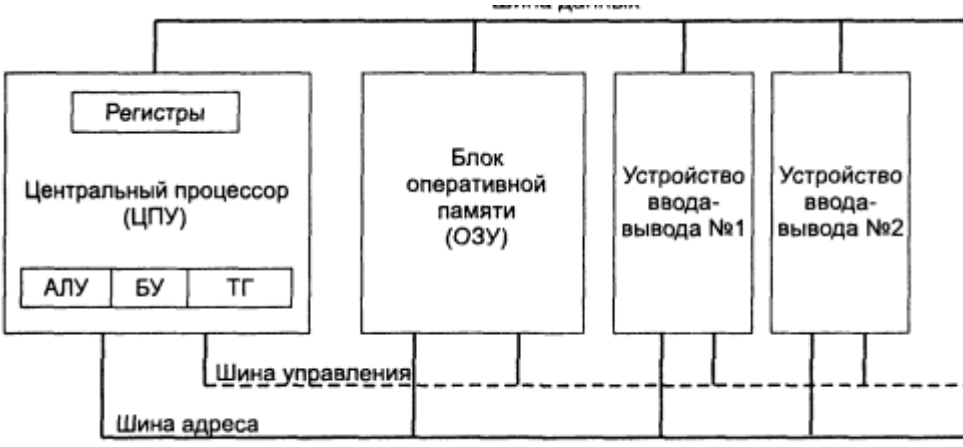
- небольшое число регистров, каждый из которых выполняет строго определённую функцию.

Наборы инструкций CISC для облегчения ручного написания программ на языках ассемблеров или прямо в машинных кодах, а также для упрощения реализации компиляторов, включали сложно выполняемые действия. Нередко в наборы включались инструкции для прямой поддержки конструкций языков высокого уровня.

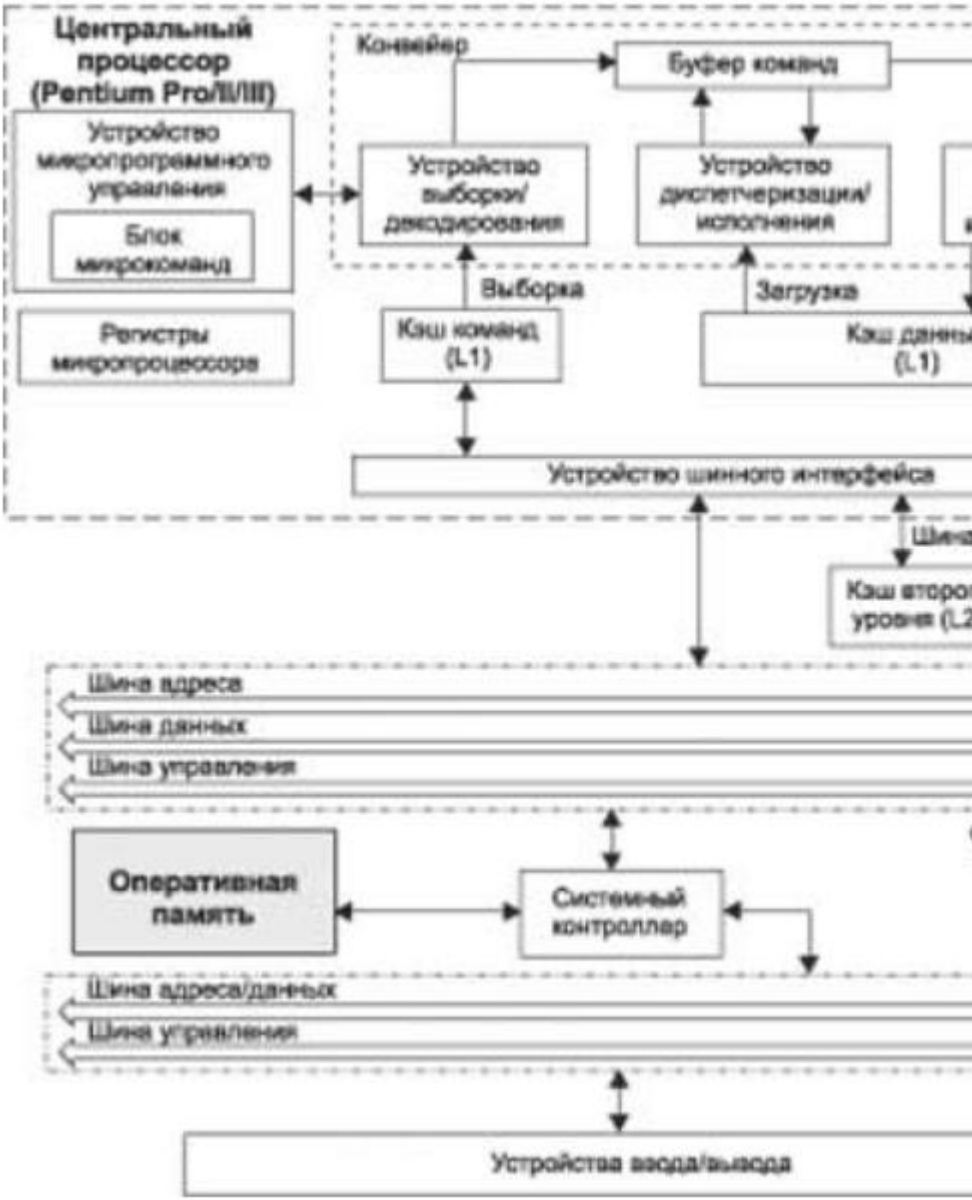
Другая особенность наборов CISC — большинство инструкций, как правило, допускали все возможные методы адресации (например, и операнды, и результат в арифметических операциях доступны не только в регистрах, но и через непосредственную адресацию, и прямо в памяти). Однако многие компиляторы не задействовали все возможности таких наборов инструкций.

Типичными представителями являются процессоры на основе x86-команд (исключая современные Intel Pentium 4, Pentium D, Core, AMD Athlon, Phenom, которые являются гибридными) и процессоры Motorola MC680x0.

Наиболее распространённая архитектура современных настольных, серверных и мобильных процессоров построена по архитектуре Intel x86 (или x86-64 в случае 64-разрядных процессоров).



Структурная схема типичной микропроцессорной системы

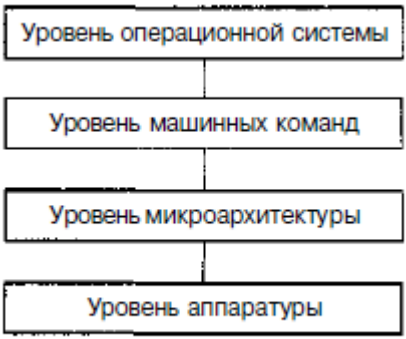

Центральный процессор (Pentium Pro/II/III)
Устройство микропрограммного управления
Блок микрокоманд
Регистры микропроцессора
Конвейер
Буфер команд
Устройство выборки/ декодирования
Устройство диспетчеризации/ исполнения
Выборка
Загрузка
Кэш команд (L1)
Кэш данных (L1)
Устройство шинного интерфейса
Шина
Кэш второго уровня (L2
Шина адреса
Шина данных
Шина управления
Оперативная память
Системный контроллер
Шина адреса/данных
Шина управления
Устройства ввода/вывода

Формально все x86-процессоры являлись CISC процессорами, однако новые процессоры, начиная с Intel 486DX, являются CISC процессорами с RISC ядром. Они непосредственно перед исполнением преобразуют CISC инструкции процессоров x86 в более простой набор инструкций RISC ядра.

В процессор встраивается аппаратный транслятор, превращающий команды x86 в команды RISC ядра. При этом одна команда x86 может порождать несколько RISC команд.

3.2. Микроархитектура Intel

Понятие микроархитектуры впервые было определено Intel для процессоров семейства Pentium Pro. Его введение объяснялось необходимостью правильного позиционирования новых процессоров среди существующих. Внешняя программная модель (логическая) 32-разрядных процессоров изменялась только в сторону развития, в то время как их исполнительная (физическая) часть могла быть совершено разной. Понятие микроархитектуры ориентировано на описание особенностей исполнительной части процессоров, то есть того, какими способами и какими средствами процессор выполняет обработку машинного кода.



Представление компьютера в виде уровней

На сегодняшний день в рамках IA-32 существует две микроархитектуры процессоров Intel: P6 и NetBurst.

3.2.1. Микроархитектура P6

Микроархитектуру P6 поддерживают такие процессоры Intel, как Pentium Pro, Pentium II (Xeon), Celeron, Pentium III (Xeon). Эта микроархитектура является, по определению Intel, трехходовой (three-way) суперскалярной конвейерной архи-

тектурой. Термин трехходовая означает поддержку технологий параллельного вычисления, позволяющих процессору одновременно (за один такт) обрабатывать до трех инструкций.

Проблема оптимальной обработки потока машинных команд является ключевой при разработке любого процессора. Поэтому для большей ясности необходимо показать эту проблему в развитии. В компьютере фон-неймановской архитектуры существуют две основные стадии исполнения команды — выборка очередной команды из памяти и собственно ее исполнение.

В первых процессорах Intel все блоки процессора работали последовательно, начиная с этапа выборки очередной команды из памяти и заканчивая этапом завершения ее обработки процессором. Напоминание об этом осталось в названии регистра IP/EIP — (Instruction Pointer — указатель инструкции). До появления процессоров Intel с конвейерной архитектурой данный регистр непосредственно указывал на очередную команду, подлежащую выполнению.

Процессоры Intel относятся к группе CISC-процессоров, в которых для выполнения одной команды может требоваться от единиц до нескольких десятков процессорных тактов. При такой обработке команд увеличение производительности может быть достигнуто только повышением частоты генерации машинных тактов. Простое увеличение частоты работы процессора не имеет смысла, так как есть физически обусловленная верхняя граница, до которой ее можно поднимать. По этому пути разработчики Intel шли до процессора i80386 включительно.

В ходе исполнения команды есть и другое узкое место — выборка команды из памяти. Это затратная по времени операция. Частичное решение проблемы было найдено еще на заре развития компьютерной техники в виде буфера упреждающей выборки. Развитием этой и реализацией других идей стал конвейер — специальное устройство, существующее на уровне архитектуры исполнительной части компьютера. Благодаря конвейеру исполнение команды разбивается на несколько стадий, каждая из которых реализует некоторую элементарную операцию общего процесса обработки команды. Впервые для процессоров Intel конвейер был реализован в архитектуре процессора i80486. Конвейер i80486 имеет пять ступеней, которые соответствуют перечисленным далее стадиям обработки машинной команды.

1. Выборка команды из кэш-памяти или из оперативной памяти.
2. Декодирование команды.

3. Генерация адреса, в ходе которой определяются адреса операндов в памяти и выполняется выборка операндов.
4. Выполнение операции с помощью АЛУ.
5. Запись результата (место записи результата зависит от алгоритма работы конкретной машинной команды).

В чем преимущество такого подхода? Очередная команда после ее выборки попадает в блок декодирования. Таким образом блок выборки освобождается и может выбрать следующую команду. В результате на конвейере могут находиться в различной стадии выполнения пять команд. Скорость вычисления в результате существенно возрастает.

В процессорах Pentium конвейерная архитектура была усовершенствована и получила название суперскалярной. В отличие от скалярной архитектуры 480486 (с одним конвейером), первые модели процессоров Pentium имели два конвейера.

В идеале такой суперскалярный процессор должен выполнять две команды за машинный такт. Но не все так просто. Реально два конвейера Pentium не были функционально равнозначными. В связи с этим они даже имели разные названия — u-конвейер (главный) и v-конвейер (второстепенный). Главный конвейер был полнофункциональным и мог выполнять любые машинные команды. Функциональность второстепенного конвейера была ограничена основными целочисленными командами и одной командой с плавающей точкой (FXCH). Внутренняя структура обоих конвейеров такая же, как у 480486 с одним общим блоком выборки команд.

Для того чтобы два разных по функциональным возможностям конвейера могли обеспечить предельную эффективность (две выполненных команды за такт работы процессора), необходимо было группировать команды из входного потока в совместимые пары. Важно заметить, что исходная последовательность команд входного потока была неизменной. Если процессору не удавалось собрать совместимую пару, то выполнялась одна команда на u-конвейере. Оставшуюся команду процессор пытался «спарить» со следующей командой входного потока.

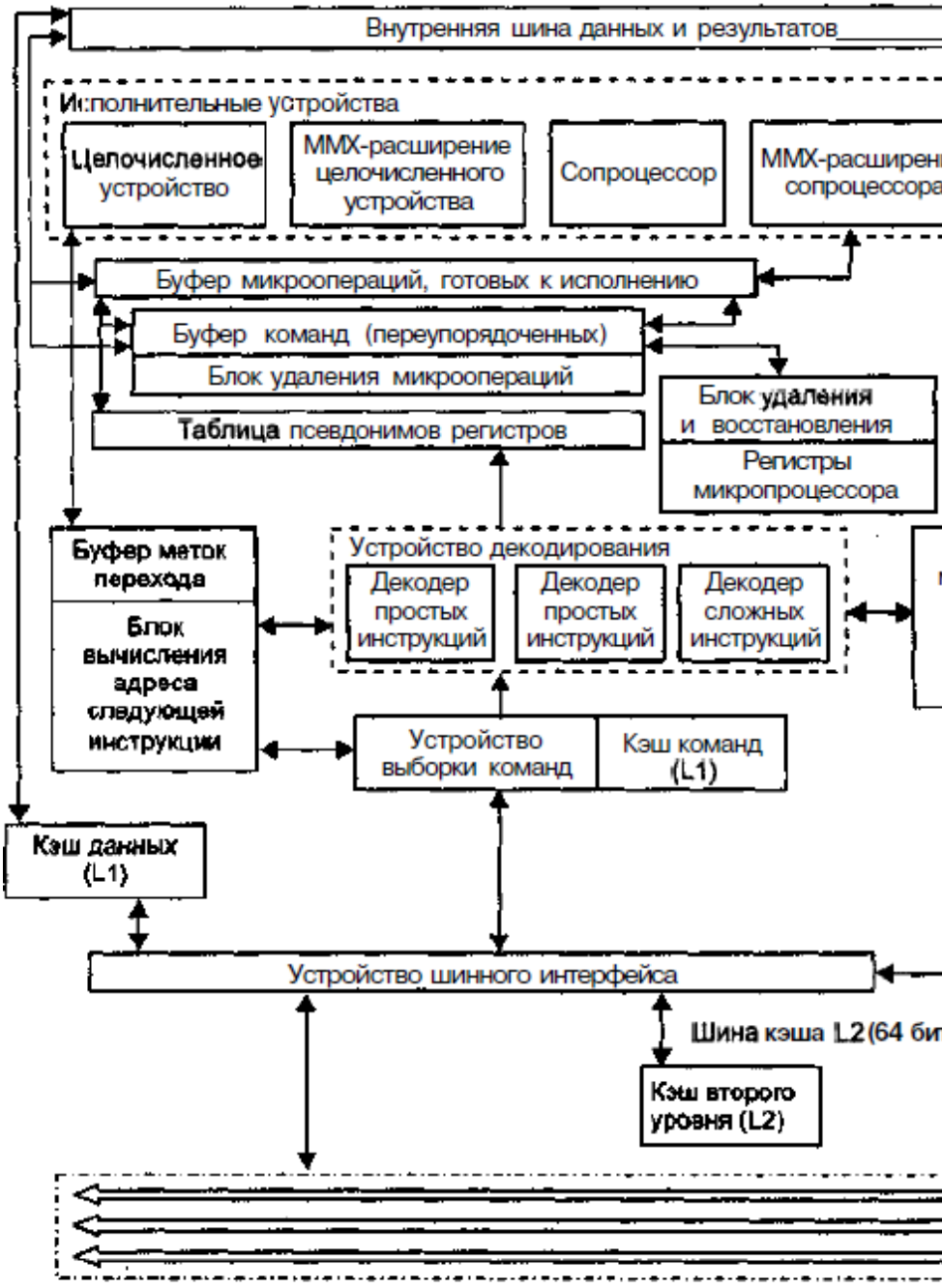
Вернемся к процессорам микроархитектуры P6. Они имеют другую структуру конвейера. Собственно конвейера в понимании 180486 и первых Pentium уже нет. Конвейеризация заключается в том, что весь процесс обработки команд разбит на 12 стадий, которые исполняются различными блоками процессора. Сколько именно команд обрабатывается процессором, сказать трудно. Термин

трехходовой означает лишь то, что для исполнения из входного потока выбираются до трех команд.

Известен верхний предел — в процессоре в каждый момент времени могут находиться до 30 команд в различной стадии исполнения. Детали этого процесса скрыты за понятием динамическое исполнение с нарушением исходного порядка следования машинных команд (out of order), что означает исполнение команд в порядке, определяемом исполнительным устройством процессора, а не исходной последовательностью команд. В основу технологии динамического исполнения положены три концепции:

- **Предсказание правильного адреса перехода**. Основная задача механизма предсказания — исключить перезагрузку конвейера. Под переходом понимается запланированное алгоритмом изменение последовательного характера выполнения программы. Как показывает статистика, типичная программа на каждые 6-8 команд содержит 1 команду перехода. Последствия обработки перехода предсказать несложно: при наличии конвейера через каждые 6-8 команд его нужно очищать и заполнять заново в соответствии с адресом перехода. Все преимущества конвейеризации теряются. Поэтому в архитектуру Pentium в состав устройства выборки/декодирования (см. главу 1) был введен блок предсказания переходов. Вероятность правильного предсказания составляет около 80 %.
- **Динамический анализ потока данных**. Анализ проводится с целью определения зависимостей команд программы от данных и регистров процессора с последующей оптимизацией выполнения потока команд. Главный критерий здесь — максимально полная загрузка процессора. Для реализации данного критерия допускается нарушение исходного порядка следования команд. Сбоя при этом не происходит, так как внешне логика работы программы не нарушается. Подобная внутренняя неупорядоченность исполнения команд позволяет держать процессор загруженным даже тогда, когда данные в кэш-памяти второго уровня отсутствуют и необходимо тратить время на обращение за ними в оперативную память.
- **Спекулятивное исполнение** — способность процессора исполнять машинные команды на участках программы с условными переходами и циклами до того, как эти переходы будут разрешены алгоритмом программы. Если переход предсказан правильно, то процессор к этому моменту уже имеет исполненный код, в противном случае весь конвейер нужно очищать, загружать и исполнять код новой ветви программы, что очень накладно.

Рассмотрим порядок функционирования исполнительного устройства микроархитектуры Pб и реализацию с его помощью описанных ранее технологий. Это рассмотрение не является строгим, кое-где для лучшего понимания оно упрощено. Для иллюстрации будем использовать схему, представленную на рисунке.



Структурная схема процессора семейства P6

Из схемы видно, что структурно процессор микроархитектуры P6 состоит из нескольких подсистем.

- **Подсистема памяти** состоит из системной шины, кэша второго уровня L2, устройства шинного^интерфейса, кэша первого уровня L1 (инструкций и данных), устройства связи с памятью и буфера переупорядочивания запросов к памяти.
- **Устройство выборки/декодирования** состоит из устройства выборки команд, блока предсказания переходов, в который входят блоки меток перехода и вычисления адреса следующей инструкции, устройства декодирования, устройства микропрограммного управления и таблицы псевдонимов регистров.
- Буфер команд.
- **Устройство диспетчеризации/исполнения** содержит буфер микроопераций, готовых к исполнению, и пять исполнительных устройств (два — для исполнения целочисленных операций, два — для исполнения операций с плавающей точкой, а также устройство связи с памятью). Необходимо заметить, что здесь допущена вольность в трактовке назначения исполнительных устройств: выделены устройства для выполнения обычных команд (целочисленных и с плавающей точкой) и MMX-команд (также целочисленных и с плавающей точкой). Реальное деление несколько иное. Такое допущение сделано исключительно с учебной целью — для более осознанного перехода от архитектуры к системе команд ассемблера.
- Блок удаления и восстановления.

Подсистема памяти. Кэширование — способ увеличения быстродействия системы за счет хранения часто используемых данных и кодов в так называемой кэш-памяти, находящейся внутри процессора (кэш-память первого уровня) либо в непосредственной близости от него (кэш-память второго уровня). Для бесперебойной работы процессора в микроархитектуре Pб используется два уровня кэш-памяти1. Кэш-память первого уровня состоит из кэшей команд и данных размером по 8 Кбайт, расположенных внутри процессора в непосредственной близости к его исполнительной части. Кэш-память второго уровня является внешней по отношению к процессору (но в едином конструктиве с ним), имеет значительно б ольший размер (256 Кбайт, 512 Кбайт или 1 Мбайт) и соединена с ядром процессора посредством 64-разрядной шины. Разделение кэш-памяти на две части (для кода и данных) обеспечивает бесперебойную поставку машинных инструкций и элементов данных в исполнительное устройство процессора. Исходные данные для кэш-памяти первого уровня предоставляет кэш-

память второго уровня. Заметьте, что информация из нее поступает на устройство шинного интерфейса и далее в соответствующую кэш-память первого уровня по 64-разрядной шине. При этом благодаря более быстрому обновлению содержимого кэш-памяти первого уровня обеспечивается высокий темп работы процессора.

Устройство шинного интерфейса обращается к оперативной памяти системы через внешнюю системную шину. Эта 64-разрядная шина ориентирована на обработку запросов, то есть каждый шинный запрос обрабатывается отдельно и требует обратной реакции. Пока устройство шинного интерфейса ожидает ответа на один апрос шины, возможно формирование многочисленных дополнительных запросов. Все они обслуживаются в порядке поступления. Считываемые по запросу данные помещаются в кэш второго уровня. То есть процессор посредством устройства шинного интерфейса читает команды и данные из кэша второго уровня. Устройство шинного интерфейса взаимодействует с кэшем второго уровня через 64-разрядную шину кэша, которая также ориентирована на обработку запросов и функционирует на тактовой частоте процессора. Доступ к кэшу первого уровня осуществляется через внутренние шины на тактовой частоте процессора. Синхронная работа с системной памятью кэш-памяти обоих уровней осуществляется благодаря специальному протоколу MESI.

Запросы от команд на получение операндов из памяти в исполнительном устройстве процессора обслуживаются посредством устройства связи с памятью и буфера переупорядочивания запросов к памяти. Эти два устройства специально включены в схему для того, чтобы обеспечить бесперебойное снабжение исполняемых команд необходимыми данными. Особо стоит подчеркнуть роль буфера переупорядочивания запросов к памяти. Он отслеживает все запросы к операндам в памяти и выполняет функции планирующего устройства. Если нужные для очередной операции данные в кэш-памяти первого уровня (L1) отсутствуют, то буфер переупорядочивания запросов к памяти автоматически передает информацию о неудачном обращении к данным кэшу второго уровня (L2). Если и в кэше L2 нужных данных не оказалось, то буфер переупорядочивания запросов к памяти заставляет устройство шинного интерфейса сформировать запрос к оперативной памяти компьютера.

Устройство выборки/декодирования. Оно извлекает одну 32-байтную строку кэша команд (L1) за такт и передает в декодер, который преобразует ее в последовательность микроопераций. Поток микроопераций (пока он еще соответствует последовательности исходных команд) поступает в буфер команд. Устройство выборки команд вычисляет указатель на следующую команду, подле-

жащую выборке, на основании информации трех источников: буфера меток перехода, состояния прерывания/исключения и сообщения от исполнительного целочисленного устройства об ошибке в предсказании метки перехода. Важная часть этого процесса —предсказание метки перехода, которое выполняется по специальному алгоритму.

В его основе лежит работа с буфером меток перехода, который содержит информацию о последних 256 переходах. Если очередная команда, выбираемая из памяти, является командой перехода, то содержащийся в ней адрес перехода сравнивается с адресами, уже находящимися в буфере меток перехода. Если этот адрес уже есть в буфере меток переходов, то он станет адресом следующей команды, с которой устройство выборки будет извлекать очередную команду. Если искомого адреса перехода в буфере нет, то выборка команд из памяти будет продолжена до момента исполнения команды перехода исполнительным устройством. В результате ее исполнения становится ясно, было ли правильным решение об адресе начала выборки следующих команд после выборки команды перехода. Если предсказанный переход оказывается неверным, то конвейер сбрасывается и загружается заново в соответствии с адресом перехода. Цель предсказания переходов — в том, чтобы устройство исполнения постоянно было занято полезной работой и сброс конвейера производился как можно реже.

Устройство выборки команд выбирает команды для исполнения и помещает их в устройство декодирования. Устройство декодирования состоит из трех параллельно работающих декодеров (два простых и один сложный). Именно эти декодеры воплощают в жизнь понятие исполнения с нарушением исходного порядка следования команд (out of order) и являются теми самыми тремя входами (three way) в исполнительное устройство процессора.

Декодеры преобразуют команды процессора в микрооперации. Микрооперации представляют собой примитивные команды, которые выполняются пятью исполнительными устройствами процессора, работающими параллельно. Многие машинные команды преобразуются в одиночные микрооперации (это делает простой декодер), а некоторые машинные команды — в последовательность от двух и более (оптимально — четырех) микроопераций (это делает сложный декодер). Информация о последовательности микроопераций для реализации конкретной машинной команды содержится в устройстве микропрограммного управления.

Кроме команд, декодеры обрабатывают также префиксы команд. Декодер команд может формировать до шести микроопераций за такт — по одной от про-

стых декодеров и до четырех от сложного декодера. Для достижения наибольшей производительности работы декодеров необходимо, чтобы на их вход поступали команды, которые декодируются шестью микрооперациями в последовательности 4+1+1. Если время работы программы критично, то имеет смысл провести ее оптимизацию, заключающуюся в переупорядочивании исходного набора команд таким образом, чтобы группы команд формировали последовательности микроопераций по схеме 4+1+1.

После того как команды разбиты на микрооперации, порядок их выполнения трудно предсказать. При этом могут возникнуть проблемы с таким критичным ресурсом, как регистры. Суть здесь в том, что если в двух соседних фрагментах программы данные помещались в одинаковые регистры, откуда они, возможно, записывались в некоторые области памяти, а после переупорядочивания эти фрагменты перемешались, то как разобраться в том, какие регистры и где использовались. Эта проблема носит название проблемы ложных взаимозависимостей и решается с помощью механизма переименования регистров. Основу этого механизма составляет набор из 40 внутренних универсальных регистров, которые и задействуются в реальных вычислениях исполнительным устройством абсолютно прозрачно для программ. Универсальные регистры могут работать как с целыми числами, так и со значениями с плавающей точкой. Информация о действительных именах регистров процессора и их внутренних именах (номерах универсальных регистров) помещается в таблицу псевдонимов регистров.

В заключение процесса декодирования устройство управления таблицей псевдонимов регистров добавляет к микрооперациям биты состояния и флаги, чтобы подготовить их к неупорядоченному выполнению, после чего посылает получившиеся микрооперации в буфер переупорядоченных команд. Нужно заметить, что новый порядок их следования не соответствует порядку следования соответствующих команд в исходной программе. Буфер переупорядоченных команд представляет собой массив ассоциативной памяти, физически выполненный в виде 40 регистров и представляющий собой кольцевую структуру, элементы которой содержат два типа микроопераций: ожидающие своей очереди на исполнение и уже частично выполненные, но не до конца из-за их переупорядочивания и зависимости от других частично или полностью не выполненных микроопераций. Устройство диспетчеризации/исполнения может выбирать микрооперации из этого буфера в любом порядке.

Устройство диспетчеризации/исполнения. Оно планирует и исполняет неупорядоченную последовательность микроопераций из буфера переупорядо-

ченных команд. Но оно не занимается непосредственной выборкой микроопераций из буфера переупорядоченных команд, так как в нем могут содержаться и не готовые к исполнению микрооперации. Этим занимается устройство, управляющее специальным буфером, который условно назовем буфером микроопераций, готовых к исполнению. Оно постоянно сканирует буфер переупорядоченных команд в поисках микроопераций, готовых к исполнению (фактически это означает доступность всех операндов), после чего посылает их соответствующим исполнительным устройствам, если они не заняты. Результаты исполнения микроопераций возвращаются в буфер переупорядоченных команд и сохраняются там наряду с другими микрооперациями до тех пор, пока не будут удалены устройством удаления и восстановления.

Подобная схема планирования и исполнения программ реализует классический принцип неупорядоченного выполнения, при котором микрооперации посылаются исполнительным устройствам вне зависимости от их расположения в исходном алгоритме. В случае, если к выполнению одновременно готовы две или более микрооперации одного типа (например, целочисленные), то они выполняются в соответствии с принципом FIFO (First In, First Out — первым пришел, первым ушел), то есть в порядке поступления в буфер переупорядоченных команд.

Непосредственно исполнительное устройство состоит из пяти блоков, каждый из которых обрабатывает свой тип микроопераций: два целочисленных устройства, два устройства для вычислений с плавающей точкой и одно устройство связи с памятью. Такимобразом, за один машинный такт одновременно исполняется пять микроопераций.

Два целочисленных исполнительных устройства могут параллельно обрабатывать две целочисленные микрооперации. Одно из этих целочисленных исполнительных устройств специально предназначено для работы с микрооперациями переходов. Оно способно обнаружить непредсказанный переход и сообщить об этом устройству выборки команд, чтобы перезапустить конвейер. Такая операция реализована следующим образом. Декодер команд отмечает каждую микрооперацию перехода и адрес перехода. Когда целочисленное исполнительное устройство выполняет микрооперацию перехода, то оно определяет, был предсказан переход или нет. Если переход предсказан правильно, то микрооперация отмечается пригодной для использования, и выполнение продолжается по предсказанной ветви. Если переход предсказан неправильно, то целочисленное исполнительное устройство изменяет состояние всех последующих микроопераций с тем, чтобы удалить их из буфера переупорядоченных

команд. После этого целочисленное устройство помещает метку перехода в буфер меток перехода, который, в свою очередь, совместно с устройством выборки команд перезапускает конвейер относительно нового исполнительного адреса.

Устройство связи с памятью. Оно управляет загрузкой и сохранением данных для микроопераций. Для их загрузки в исполнительное устройство достаточно определить только адрес памяти, поэтому такое действие кодируется одной микрооперацией. Для сохранения данных необходимо определять и адрес, и записываемые данные, поэтому это действие кодируется двумя микрооперациями. Та часть устройства связи с памятью, которая управляет сохранением данных, имеет два блока, позволяющие ему обрабатывать адрес и данные для микрооперации параллельно. Это позволяет устройству связи с памятью выполнить загрузку и сохранение данных для микроопераций параллельно в одном такте.

Исполнительные устройства с плавающей точкой аналогичны устройствам в более ранних моделях процессора Pentium. Было добавлено только несколько новых команд с плавающей точкой для организации условных переходов и перемещений.

Последний блок в этой схеме выполнения команд исходной программы — блок удаления и восстановления, задачей которого является возврат вычислительного процесса в рамки, определенные исходной последовательностью команд. Для этого он постоянно сканирует буфер переупорядоченных команд на предмет обнаружения полностью выполненных микроопераций, не имеющих связи с другими микрооперациями. Такие микрооперации удаляются из буфера переупорядоченных команд и восстанавливаются в порядке, соответствующем порядку следования команд исходной программы с учетом прерываний, исключений, точек прерывания и переходов. Блок удаления и восстановления может удалить три микрооперации за один машинный такт. При восстановлении порядка следования команд блок удаления и восстановления записывает результаты в реальные регистры процессора и в оперативную память.

Таким образом, система динамического исполнения команд позволяет организовать прохождение команд программы через исполнительное устройство процессора эффективнее, чем это было в конвейере процессора 180486 и первых процессоров Pentium.

3.2.2. Микроархитектура NetBurst

Микроархитектура NetBurst, реализованная в процессоре Pentium IV, является развитием идей микроархитектуры P6, поэтому рассматривается довольно конспективно. Судя по названию (net — сеть, burst — прорыв), микроархитектура NetBurst призвана обеспечить некий «сетевой прорыв». Очевидно, что этим разработчики хотели подчеркнуть те новые особенности процессора Pentium IV, которые позволяют организовать более быструю и эффективную работу приложений в современных сетевых и мультимедийных информационных средах. Отметим наиболее важные свойства новой микроархитектуры.

- Применена гарвардская структура с разделением потоков команд и данных.
- Быстрая исполнительная часть процессора. АЛУ процессора работает на удвоенной частоте процессора. За каждый такт процессора выполняются две основные целочисленные команды. Обеспечена более высокая пропускная способность потока команд через исполнительную часть процессора и уменьшены различные задержки.
- Используется гиперконвейерная технология (Hyper-PIPelINed Technology) выполнения команд, при которой число ступеней конвейера достигает 31 (в Pentium III - 11 ступеней). Таким образом, одновременно в процессе выполнения на разных стадиях реализации может находиться свыше 30 команд. Цель увеличения длины конвейера — упрощение задач, реализуемых каждой из его ступеней, и, как следствие, упрощение соответствующей аппаратной логики.
- Используется динамическое выполнение команд (dynamic execution), построенное на трех базовых концепциях: предсказание переходов (branch prediction), динамический анализ потока данных (dynamic data flow analysis) и спекулятивное выполнение (OUT-oforder execution). Аналогичный механизм, названный Dynamic Execution, используется в МП Pentium III, однако в INTel Pentium 4 он улучшен.
- Выполнение арифметических и логических операций происходит с удвоенной тактовой частотой процессора, что позволяет за один такт получить результаты для двух команд.
- Новая подсистема кэширования. Отсутствует кэш команд первого уровня. Вместо него введен кэш трасс. Трассами называются последовательности микроопераций, в которые были декодированы ранее выбранные команды. Кэш трасс способен хранить до 12 Кбайт микроопераций и доставлять ис-

полнительному ядру до 3 микроопераций за такт. Кэш второго уровня работает на полной частоте ядра процессора.

- Кеш-память 2-го уровня емкостью 256 Кбайт размещается непосредственно на кристалле процессора, что позволяет сократить время выборки по сравнению с Pentuim III, где эта кэш-память располагается на отдельном кристалле в общем корпусе с процессором.

Микроархитектура NetBurst поддерживает еще одну новую технологию — HyperThreading. Данная технология позволяет на базе одного физического процессора Pentium IV моделировать несколько логических, каждый из которых имеет собственное архитектурное пространство IA-32. Под архитектурным пространством IA-32 понимается совокупность регистров данных, сегментных регистров, системных регистров и регистров MSR. Каждый логический процессор имеет также собственный контроллер прерываний API C.

3.2.3. Микроархитектура Pentium 4

Микропроцессор Pentium 4 является завершающей моделью 32-разрядных микропроцессоров фирмы Intel с архитектурой IA-32. Основные особенности этого процессора:

- ·новая микроархитектура процессора NetBurst (пакетно-сетевая);
- ·новая системная шина FSB.

Микроархитектура процессора определяет реализацию его внутренней структуры, принципы выполнения поступающих команд, способы размещения и обработки данных.

Структурная схема процессора Pentium IV показана на рисунке.

Структура микропроцессора Pentium 4

Команды и данные поступают в микропроцессор через блок системного интерфейса.

Любой процессор архитектуры x86 обязательно оснащен процессорной шиной. Эта шина служит каналом связи между процессором и всеми остальными устройствами в компьютере: памятью, видеокартой, жестким диском и так далее. Так, классическая схема организации внешнего интерфейса процессора предполагает, что параллельная мультиплексированная процессорная шина, кото-

рую принято называть FSB (Front Side Bus), соединяет процессор (иногда два процессора или даже больше) и контроллер, обеспечивающий доступ к оперативной памяти и внешним устройствам. Этот контроллер обычно входит в состав северного моста набора системной логики (чипсета).

Для ускорения обмена с памятью в Pentium 4 используется новая реализация системной шины, обеспечивающая обмен с эквивалентной частотой 400 МГц. Такая скорость достигается путем применения нового типа сверхбыстродействующей двухканальной памяти типа RDRAM и специальной микросхемы MCH (Memory Controller Hub), реализующей 4 канала передачи данных. При тактовой частоте каждого канала 100 МГц обеспечивается общая частота обмена, эквивалентная 400 МГц. Шина включает 64-разрядную двунаправленную шину данных, дающую пропускную способность в 3,2 Гбайт/с, и 36-разрядную шину адреса (33 адресных линии A35-A3 и 8 линий выбора байтов BE7-BE0), что позволяет адресовать физическую память емкостью до 64 Гбайт.

Полученная по системной шине информация сохраняется в кэш-памяти 2-го уровня (L2) емкостью 256 Кбайт, общей для команд и данных, которая размещается непосредственно на кристалле МП. Ширина шины, по которой идет обмен данными между кэш-памятью L2 и процессором, составляет 256 бит (32 байта), а ее тактовая частота совпадает с тактовой частотой ядра процессора.

Гарвардская внутренняя структура реализуется на уровне кэш-памяти 1-го уровня (L1) путем разделения потоков команд и данных. Кэш-память данных 1-го уровня имеет емкость 8 Кбайт. Вместо кэш-памяти команд 1-го уровня в Pentium 4 используется кэш-память для декодированных команд (микрокоманд). Execution TRace Cache - это название и одновременно способ реализации L1-кэша инструкций в архитектуре NetBurst. Смысловое содержание этого термина можно перевести как "кэш трассировки выполняемых микрокоманд". В Execution TRace Cache хранятся микрокоманды (?ops), которые были получены в результате декодирования входного потока инструкций исполняемого кода и готовы для передачи на выполнение конвейеру. Емкость Execution TRace Cache составляет 12 Кбайт.

После заполнения кэш-памяти микрокоманд практически любая команда будет храниться в ней в декодированном виде. Поэтому при поступлении очередной команды блок трассировки выбирает из этой кэшпамяти необходимые микрокоманды, обеспечивающие ее выполнение.

Если в потоке команд оказывается команда условного перехода, то включается механизм предсказания ветвления, который формирует адрес следующей вы-

бираемой команды до того, как будет определено условие выполнения перехода.

После формирования потоков микрокоманд производится выделение регистров, необходимых для выполнения декодированных команд. Эта процедура реализуется блоком распределения регистров. Он выделяет для каждого указанного в команде логического регистра (регистра цлочисленных операндов EAX, EBX и т. д., регистра операндов с плавающей точкой ST0-ST7 или регистра блоков MMX, SSE) один из 128 физических регистров, входящих в состав блоков регистров замещения (БРЗ) целочисленного блока микропроцессора и блока обработки чисел с плавающей точкой. Эта процедура позволяет минимизировать конфликты в конвейерах и выполнять команды, использующие одни и те же логические регистры, одновременно или с изменением их последовательности. Ступени распределения/переименования конвейера могут выпустить три микрокоманды за такт на следующую ступень конвейера.

Выбранные микрокоманды размещаются в очереди микрокоманд. В ней содержатся микрокоманды, реализующие выполнение до 120 поступивших и декодированных команд, которые затем направляются в исполнительные устройства. Отметим, что в процессорах Pentium III в очереди находятся микрокоманды для 40 поступивших команд. Значительное увеличение числа команд, стоящих в очереди, позволяет более эффективно организовать поток их исполнения, изменяя последовательность выполнения команд и выделяя команды, которые могут выполняться параллельно. Эти функции реализует блок распределения микрокоманд. Он выбирает микрокоманды из очереди не в порядке их поступления, а по мере готовности соответствующих операндов и исполнительных устройств. В результате команды, поступившие позже, могут быть выполнены до ранее выбранных команд. При этом реализуется одновременное выполнение нескольких микрокоманд (команд) в параллельно работающих исполнительных устройствах. Таким образом, естественный порядок следования команд (микрокоманд) нарушается, чтобы обеспечить более полную загрузку параллельно включенных исполнительных устройств и повысить производительность процессора.

Адреса операндов, выбираемых из памяти, вычисляются блоком формирования адреса (БФА), который реализует интерфейс с кэш-памятью данных 1-го уровня. В соответствии с заданными в декодированных командах способами адресации формируются 48 адресов для загрузки операндов из памяти в регистр БРЗ и 24 адреса для записи из регистра в память (в Pentium III формируются 16 адресов для загрузки регистров и 12 адресов для записи в память). При этом

БФА формирует адреса операндов для команд, которые еще не поступили на выполнение. При обращении к памяти БФА одновременно выдает адреса двух операндов: один для загрузки операнда в заданный регистр БРЗ, второй - для пересылки результата из БРЗ в память. Таким образом реализуется процедура предварительного чтения данных для последующей их обработки в исполнительных блоках (спекулятивная выборка).

Аналогичным образом организуется параллельная работа блоков SSE, FPU, MMX, которые используют отдельный набор регистров и блок формирования адресов операндов.

При выборке операнда из памяти производится обращение к кэшпамяти данных (L1), которая имеет отдельные порты для чтения и записи. За один такт производится выборка операндов для двух команд.

При формировании адресов обеспечивается обращение к заданному сегменту памяти. Каждый сегмент может делиться на страницы. Для сокращения времени трансляции используется **буфер ассоциативной трансляции страничного адреса** TLB, который хранит базовые адреса наиболее часто используемых страниц.

Микрокоманды поступают в исполнительное ядро из блока распределения по 4 портам в 8 исполнительных блоков. Эти порты выполняют функцию шлюзов к функциональным устройствам. Для обработки целочисленных данных и выполнения логических операций в Pentium 4 используются 4 однотипных арифметико-логических устройства (ALU). Обработка чисел с плавающей запятой проходит в FPU. Блоки MMX и SSE предназначены для выполнения команд этих типов.

За один такт через порты может пройти до 6 микрокоманд. Это больше, чем может выполнить препроцессор (3 микрокоманды за такт), что дает некоторую свободу в случае резкого увеличения количества готовых к исполнению микрокоманд. Суперскалярная архитектура микропроцессора реализуется путем организации исполнительного ядра МП в виде ряда параллельно работающих блоков.

Арифметико-логические блоки ALU производят обработку **целочисленных** операндов, которые поступают из заданных регистров БРЗ. В эти же регистры заносится и результат операции. При этом проверяются условия ветвления для команд условных переходов и выдаются сигналы перезагрузки конвейера команд в случае неправильно предсказанного ветвления. Рабочая тактовая частота модулей ALU в два раза выше тактовой частоты процессора. Это достига-

ется за счет срабатывания как по переднему, так и по заднему фронтам задающего тактового сигнала. Таким образом, каждый ALU-модуль способен выполнить до двух целочисленных операций за один рабочий такт процессора.

Эффективность конвейера резко снижается из-за необходимости его перезагрузки при выполнении условных ветвлений, когда требуется произвести очистку всех предыдущих ступеней и выбрать команду из другой ветви программы. Чтобы сократить потери времени, связанные с перезагрузкой конвейера, используется улучшенный **блок предсказания ветвлений**. Его основной частью является ассоциативная память, называемая буфером адресов ветвлений BTB, в которой хранятся 4092 адреса ранее выполненных переходов. Отметим, что в BTB процессора Pentium III хранятся адреса только 512 переходов. Кроме того, BTB содержит биты, хранящие предысторию ветвления, которые указывают, выполнялся ли переход при предыдущих выборках данной команды. При поступлении очередной команды условного перехода указанный в ней адрес сравнивается с содержимым BTB. Если этот адрес не содержится в BTB, то есть ранее не производились переходы по данному адресу, то предсказывается отсутствие ветвления. В этом случае продолжается выборка и декодирование команд, следующих за командой перехода. При совпадении указанного в команде адреса перехода с каким-либо из адресов, хранящихся в BTB, производится анализ предыстории. В процессе анализа определяется чаще всего реализуемое направление ветвления, а также выявляются чередующиеся переходы. Если предсказывается выполнение ветвления, то выбирается и загружается в конвейер команда, размещенная по предсказанному адресу. Более совершенный механизм предсказания переходов в МП Pentium 4 обеспечивает уменьшение количества ошибочно предсказанных переходов в среднем на 33 % по сравнению с Pentium III. Таким образом, резко уменьшается число перезагрузок конвейера при неправильном предсказании ветвления.

В Pentium 4 также интегрирован набор из 144 новых SIMD-инструкций, обеспечивающих одновременное выполнение одной операции над несколькими операндами. Рассмотрим особенности использования этой схемы обработки данных подробнее.

3.2.4. Микроархитектура Intel Core

В Intel делают особую ставку на микроархитектуру Core, поскольку в самое ближайшее время на неё перейдут не только PC-совместимые настольные ПК и ноутбуки, но и компьютеры Apple, и даже мощные серверы, рассчитанные на операционную систему следующего поколения Windows Vista. Неслучайно при разработке этой универсальной архитектуры ставились задачи максимального

увеличения производительности при максимальном снижении энергопотребления - эти требования сегодня предъявляются не только к ноутбукам, но и к настольным машинам, и к серверам.

По формальным признакам Core ближе всего к архитектуре мобильных Pentium M (Banias), однако при этом целый ряд технологий позаимствован и из Pentium 4. В каком-то смысле, перефразируя Ленина, Core можно охарактеризовать как "шаг назад, два шага вперёд": за основу новой архитектуры, как, впрочем, и архитектуры Pentium M, была взята микроархитектура P6, которая за шесть лет забытья не только не потеряла своей актуальности, но и доказала свою перспективность. Разумеется, Core - далеко не механическая копия P6; скорее - это эволюционное развитие отвергнутой некогда архитектуры.

Как известно, разработкой Core (как и Pentium M) занималось израильское отделение Intel, и уже с самого начала работы перед инженерами была поставлена цель: ориентироваться на многоядерные микросхемы. При этом важно было добиться существенного снижения энергопотребления без ущерба для производительности. К счастью, проблемы с размещением многоядерных процессоров в пределах печатных плат привычных размеров не возникало: переход на более тонкие технологические процессы упрощает задачу размещения на одном кристалле нескольких ядер, поскольку, несмотря на рост числа транзисторов, сами эти транзисторы становятся всё меньше и меньше.

Кстати, выпуск многоядерных процессоров обеспечит дальнейшее процветание культа "закона Мура", говорящего о ежегодном удвоении числа транзисторов в микропроцессорах. Всё дело в масштабируемости, которой характеризуется архитектура Core. Если в Pentium 4 производительность можно было повышать, прежде всего, путём увеличения тактовой частоты, то в случае с Core гораздо проще добавить дополнительные ядра, а потом уже - проводить периодические повышения тактовые частот.

Сегодня всё больше микропроцессоров отходят от схемы внеочередного исполнения команд (OOOE) в пользу поочерёдного, появляется всё больше моделей с архитектурой VLIW (с очень длинными инструкциями, предусматривающих несколько параллельно выполняющихся операций), которым требуется многопоточность и оптимизированные компиляторы. Тем не менее, разработчики Core уделили особое внимание именно схеме OOOE, при которой обеспечивается стопроцентно аппаратная оптимизация выполняемого кода и потоков на уровне кристалла. Инженеры Intel взяли все прекрасно себя зарекомендовавшие элементы современных процессоров и объединили их, одновременно обеспечив расширенные вычислительные ресурсы.

Разумеется, существуют ограничения на число параллельно выполняемых инструкций, поэтому, чем больше вычислительные ресурсы, тем больше доступных для исполнения тактов могут остаться невостребованными из-за ограничений в параллелизме на уровне команд (ILP). Нерациональное использование больших ресурсов возможно и из-за латентности памяти, снижающей общую скорость работы системы.

В микроархитектуре Core предусмотрены целенаправленные решения именно этих двух проблем - ограничений в параллелизме инструкций и латентности памяти, - призванные обеспечить максимальную загрузку ядра. В интерфейсном блоке применяются технология Micro-ops fusion, при которой несколько инструкций после декодирования объединяются и отправляются для обработки в конвейер, и технология Macro-fusion, при которой ещё до декодирования несколько команд "сливаются" вместе и обрабатываются как единая инструкция. Кроме того, в интерфейсном блоке имеется модуль предсказаний переходов-ветвлений (BPU), ускоряющий обработку инструкций до их передачи на исполнение. Расширенная шина позволяет большему числу инструкций передаваться на исполнительные блоки за каждый такт. Кроме того, в архитектуре Core было ликвидировано известное "узкое место" блока SSE, присутствовавшее в предыдущих микроархитектурах, что обеспечивает заметно более высокую производительность новых чипов в векторных вычислениях по сравнению с предшественниками.

Одной из характерных особенностей архитектуры P6 была структура портов запуска (issue port), которые в Intel называют "dispatch ports", т.е. "порты диспетчеризации". Эта структура сохранена и в микроархитектуре Core, однако существуют и важные различия в портах запуска и буфере ("станции") резервирования (reservation station или RS) двух этих архитектур.

Чтобы понять происхождение этой структуры портов запуска, стоит обратиться к структурной схеме процессора Pentium Pro - первого чипа на базе архитектуры P6. Два порта из пяти отведены арифметике, а три оставшихся - отвечают за доступ к памяти. Буфер резервирования ядра P6 способен передавать на исполнительные блоки по одной инструкции через каждый порт за один такт, то есть, всего до пяти инструкций за такт.

С развитием архитектуры P6 в процессорах Pentium II и Pentium III дорабатывалось и ядро: были добавлены исполнительные блоки для целочисленных векторных вычислений и векторных расчётов с плавающей запятой. Эти блоки были добавлены на два арифметических порта, что привело к их некоторой перегруженности. Именно из-за этого возникло описанное выше "узкое место",

из-за которого возможно снижение производительности векторных вычислений в условиях недостаточной пропускной способности портов. Аналогичной архитектурой обладает и процессор Pentium M на ядре Banias.

Исполнительное ядро процессора Core. Помимо существенно расширенного буфера резервации (до 32 записей), в ядре Core реализована новая структура портов: здесь не пять портов, как в ядрах P6, не четыре порта, как в NetBurst, а уже шесть портов. В отличие от предшественников, в Core арифметическим и логическим инструкциям отданы не два, а три порта, что и позволяет избежать описанного выше "узкого места".

У Core есть три 64-разрядных исполнительных блока для целочисленных вычислений, каждый из которых может исполнять однотактовые 64-битные скалярные целочисленные операции. По всей видимости, здесь, как и в ядре P6 предусмотрен один сложный 64-разрядный блок (CIU) и два простых блока (SIU), выполняющих элементарные действия вроде сложения. Один из SIU делит порт 2 с модулем исполнения переходов (BEU): они способны работать параллельно над выполнением объёдиненных в процессе Macro-fision инструкций.

Способность осуществлять 64-разрядные целочисленные вычисления за один такт впервые появились в процессорах Intel x86 line, и эта возможность ставит чипы с архитектурой Core даже выше серверного процессора IBM PowerPC 970, способного выполнять такие вычисления только за два такта. Более того, поскольку блоки арифметической логики (ALU) расположены на отдельных портах, теоретически чипы Core способны выполнять за один цикл сразу три 64-разрядные целочисленные операции.

В Core используются два исполнительных блока для вычислений с плавающей запятой, способные осуществлять как скалярные, так и векторные арифметические операции. Блок, расположенный на порту 1, отвечает за сложение и другие простые операции в форматах: скалярных - с одинарной (32-бит) и двойной (64-бит) точностью; векторных - с 4х одинарной и 2х двойной точностью. Исполнительный блок на порту 2 осуществляет операции умножения и деления в этих же векторных и скалярных форматах. Два блока векторных вычислений делят те же порты, что и два аналогичных блока скалярных вычислений, поэтому их следует считать двумя едиными линиями, выполняющие как скалярные, так и векторные операции.

Одно из важнейших улучшений в Core - блоки векторных или SIMD-вычислений (т.е. с одним потоком инструкций и несколькими потоками данных). Новая архи-

тектура обеспечивает полноценную 128-битную обработку во всех векторных блоках. Когда Intel впервые добавила поддержку 128-битных векторных вычислений в процессоры семейства Pentium при помощи потоковых SIMD-расширений (SSE), результаты оказались не слишком впечатляющими из-за отсутствия возможности работы с трёхоперандными инструкциями, а также из-за ограничений 64-битной внутренней разрядности обработки данных для арифметики с плавающей запятой и инструкций MMX. Для выполнения 128-битных инструкций в P6 приходилось сначала делить эту инструкцию на две, а затем передавать эту пару на соответствующий исполнительный блок. В результате все 128-битные операции могли выполняться не быстрее, чем за два такта. Этот недостаток унаследовали и Pentium 4, и Pentium M.

В новой архитектуре Core благодаря 128-битной ширине внутренних шин обеспечивается однотактовое исполнение 128-разрядных векторных инструкций. При этом требуется лишь одна микрооперация для трансляции и декодирования каждой 128-битной инструкции. К сожалению, двухоперандное ограничение осталось и в Core, однако серьёзных проблем оно уже не вызывает. Если объединить описанные доработки с увеличением числа исполнительных блоков и расширенной пропускной способностью внутренних шин, можно представить, насколько значительно выросла теоретическая производительность векторных операций.

Конвейеры. Пока доступно не слишком много подробной информации об организации вычислительного конвейера в Core, однако, уже известно, что он состоит из 14 ступеней. Для сравнения, столько же ступеней у процессора IBM PowerPC 970, у Pentium 4 на ядре Prescott - 30 ступеней, а у чипов с архитектурой P6 - 12 ступеней. Число ступеней свидетельствует, с одной стороны, о сохранении сильных черт архитектуры P6, а с другой - об отказе от "гонки мегагерц", свойственной архитектуре NetBurst. Точной информации от Intel о назначении двух дополнительных ступеней пока нет, и мы не станем выдвигать собственных версий.

Поскольку постпроцессор ядра Core тоже имеет гораздо более широкую шину, чем у предшественников, буфер переупорядочивания инструкций (ROB) также расширен до 96 записей. Для сравнения, у Pentium M Banias ROB состоял из 40 записей. Так называемое "окно команд", состоящее из буфера переупорядочивания (ROB) и буфера резервирования (RS), было не только физически расширено, но и "виртуально". Технологии Macro-fusion and micro-ops fusion позволяют отслеживать большее число инструкций меньшими средствами. Именно поэтому можно сказать, что функционально окно команд Core шире, чем просто сум-

ма записей ROB и RS. А наполнение такого окна команд необходимым потоком новых инструкций - весьма нетривиальная задача, с которой, впрочем, вполне справились инженеры Intel.

В интерфейсном блоке установлен новый декодирующий модуль, позволяющий увеличить число инструкций, которые могут быть конвертированы в микрооперации за один цикл. Декодер, использовавшийся в ядре P6, состоял их двух "быстрых" декодирующих блоков и одного сложного ("медленного") декодирующего блока. "Быстрые" блоки транслируют инструкции x86 в одну микрооперацию (micro-op), из которых и состоит подавляющее большинство инструкций. "Быстрые" декодеры за один такт могут отправлять одну микрооперацию в соответствующий буфер. Сложный декодер отвечает за трансляцию инструкций в две-четыре микрооперации. Таких инструкций немного и они редко используются. Три декодирующих модуля ядра P6 могут оправлять за один такт до шести микроопераций в буфер микроопераций, а декодирующий блок в целом может посылать в буфер переупорядочивания до шести микроопераций за один такт.

Перед конструкторами Core стояла задача повысить скорость декодирования настолько, чтобы она позволяла использовать возросшие вычислительные мощности ядра, Прежде всего, в декодирующий блок был добавлен ещё один "быстрый" модуль, что позволило увеличить число отправляемых в буфер микроопераций до семи и число отправляемых в буфер переупорядочивания - до четырёх. Кроме того, в Core "быстрыми" модулями могут обрабатываться больше типов инструкций, в частности, инструкции памяти и SSE, благодаря чему разработчики приблизились к цели получить одну микрооперацию из каждой инструкции x86.

Среди новых технологий, реализованных в архитектуре Core - уже упомянутая Macro-fusion, позволяющая объединять некоторые типы инструкций x86 перед декодированием, что даёт возможность отправлять их на один декодер для трансляции в одну микрооперацию. Такими инструкциями могут быть, в частности, инструкции сравнения, тестирования и переходов. Любой из четырёх декодеров способен за один такт генерировать не более одной микрооперации с использованием технологии Macro-fusion. Один блок арифметической логики, тем самым, способен выполнять, как минимум, две объединённых инструкции определённого типа одновременно. К тому же, при этом освобождаются исполнительные блоки для других типов инструкций. Таким образом, эта технология позволяет "виртуально" расширить окно команд, выполняя в действительности больше инструкций, чем их формальное количество.

Ещё одна технология, Micro-ops fusion, впервые появившаяся в Pentium M, решает почти те же самые задачи, что и Macro-fusion, но другими средствами: "быстрый" декодирующий модуль получает одну инструкцию x86, которая должна транслироваться в две микрооперации, и выдаёт смешанную пару, которая воспринимается в буфере переупорядочивания как одна запись. После перехода в буфер резервирования две микрооперации обрабатываются отдельно: либо параллельно через два отдельных порта, либо последовательно через один порт, в зависимости от конкретной ситуации. Как правило, такими микрооперациями являются load и store.

Технология Micro-ops fusion, как и Macro-fusion, "виртуально" расширяет окно команд и делает архитектуру Core более энергоэффективной, поскольку большее число операций выполняется меньшими аппаратными средствами.

Модуль предсказания переходов в ядре Core был существенно расширен в целях обеспечения повышенной производительности и энергоэффективности. Модуль состоит из тех же трёх частей, что и аналогичный модуль в процессорах Pentium M (Banias): прямого предсказателя ветвлений (global и bi-modal), непрямого (indirect) и определителя циклов (loop detector). Предсказатели работают на основе информации о недавно исполненных переходах, причём прямой путь, заложенный в самой команде, обеспечивает почти стопроцентную точность предсказаний, в то время как при непрямом пути данные о наиболее часто используемых адресах извлекаются из регистра, что обеспечивает чуть меньшую, но всё равно весьма высокую точность.

Переходы завершения цикла могут быть выполнены лишь однажды - по завершению цикла, поэтому в буфере переходов не сохраняется достаточно статистических данных для того чтобы с точностью предсказать завершение ветвления. Для сведения к минимуму вероятности возникновения подобных ситуаций применяется определитель циклов, отслеживающий исполнение каждого ветвления с целью определить, какое из них удовлетворяет условиям завершения цикла. Соответствующая статистика накапливается, поэтому точность предсказания подобных переходов приближается к ста процентам.

Устранение неоднозначности в памяти. Оригинальная технология устранения неоднозначности в памяти (memory disambiguation) призвана сводить к минимуму недостатки процессоров OOOE (то есть, со схемой внеочередного исполнения команд). Дело в том, что такие процессоры не могут производить дальнейшие вычисления до тех пор, пока не будут возвращены в основную память или в файл регистра данные о результатах исполнения предыдущих инструкций.

Даже если последующая операция никак не зависит от предыдущих, процессор вынужден ожидать сохранения результатов их выполнения.

По некоторым данным, порядка 97 процентов инструкций в окне команд никак не связаны с предшествующими и могут исполняться независимо от них. Тем не менее, в архитектурах P6 и NetBurst предполагалось, что эти связи существуют, и это предположение вело к неоправданному снижению производительности. Технология устранения неоднозначности как раз и пытается определять подобные ложные связи, что позволяет повысить производительность работы чипа.

Решение проблемы заключается в том, чтобы дать процессору "понять" зависимости между исполняемыми инструкциями. Для этого и применяется технология устранения неоднозначности: при помощи особых алгоритмов предпринимаются попытки предсказания последовательности загрузки последующих инструкций ещё до сохранения предыдущих. В худшем случае операции выполняются, как и раньше, после сохранения, а при удачном исходе предсказания - одновременно с сохранением данных о результатах выполнения предыдущих операций. Разумеется, подобные алгоритмы применимы не во всех ста процентах случаев, но в целом их использование позволит заметно увеличить производительность процессора.

3.2.5. Микроархитектура Intel Core Duo

Core 2 — шестое поколение микропроцессоров архитектуры x86-64 корпорации Intel, основанное на процессорной архитектуре Intel Core. Это потомок микроархитектуры Intel P6, на которой, начиная с процессора Pentium Pro, построено большинство микропроцессоров Intel, исключая процессоры с архитектурой NetBurst. Введя новый бренд, от названий Pentium и Celeron Intel не отказалась, в 2007 году переведя их также на микроархитектуру Core, и на данный момент доступны процессоры Pentium Dual-Core (не путать с Pentium D) и Core Celeron (400-я серия). Но теперь воссоединились мобильные и настольные серии продуктов (разделившиеся на Pentium M и Pentium 4 в 2003 году).

Первые процессоры Core 2 официально представлены 27 июля 2006 года. Также как и их предшественники, процессоры Intel Core, они делятся на модели Solo (одноядерные), Duo (двухъядерные), Quad (четырёхъядерные) и Extreme (двух- или четырёхъядерные с повышенной частотой и разблокированным множителем). Процессоры получили следующие кодовые названия — «Conroe» (двухъядерные процессоры для настольного сегмента), «Merom» (для портативных ПК), «Kentsfield» (четырёхъядерный Conroe) и «Penryn» (Merom, выполненный по 45 нанометровому техпроцессу). Хотя процессоры «Woodcrest» так-

же основаны на архитектуре Core, они выпускаются под маркой Xeon.[1]. С декабря 2006 года все процессоры Core 2 Duo производятся на пластинах диаметром 300 миллиметров на заводе Fab 12 в Аризоне, США и на заводе Fab 24-2 в County Kildare, Ирландия.

В отличие от процессоров архитектуры NetBurst (Pentium 4 и Pentium D), в архитектуре Core 2 ставка делается не на повышение тактовой частоты, а на улучшение других параметров процессоров, таких как кэш, эффективность и количество ядер. Рассеиваемая мощность этих процессоров значительно ниже, чем у настольной линейки Pentium. С параметром TDP, равным 65 Вт, процессор Core 2 имеет наименьшую рассеиваемую мощность из всех доступных в продаже настольных чипов, в том числе на ядрах Prescott (в системе кодовых имён Intel) с TDP, равным 130 Вт, и на ядрах San Diego's (в системе кодовых имён AMD) с TDP, равным 89 Вт.

3.2.6. Микроархитектура Intel Nehalem

Архитектура Nehalem была представлена в 2008 как архитектура, которая может адаптироваться для нужд всех трех основных рынков: мобильного, настольного и серверного.

Архитектура Nehalem названа именем одного из индейских племен и отличается рядом принципиальных новшеств:

- Все ядра размещены на одном кристалле.
- На самом процессоре расположен двухканальный или трехканальный контроллер оперативной памяти DDR3.
- В чип интегрирована системная шина QPI или DMI, заменившая FSB.
- МП содержит общую для всех ядер кеш - память третьего уровня.
- В МП может быть встроено графическое ядро.

Энергопотребление стало на 30% меньше благодаря использованию в Nehalem набору инструкций SSE 4.2, вновь стала использоваться технология Hyper-Threading, позволяющая представить одно физическое ядро как два виртуальных. В первых процессорах Nehalem использовалась технология 45-нм, а с 2010 стали применять технологию 32-нм. Процессоры устанавливаются в материнские платы, имеющие рпзъем **LGA1156** или **LGA1366**.

Nehalem изначально была разработана модульной — набор базовых "кирпичей", которые можно собирать, чтобы создавать разные версии архитектуры.

3.2.7. Микроархитектура Intel Sandy Bridge

Sandy Bridge — микроархитектура процессоров, разработанная фирмой Intel. Она основана на 32-нм технологическом процессе, анонсирована 3.01.2011. Процессоры этой архитектуры будут продаваться на рынке под торговыми марками Core i7/i5/i3/Pentium/Celeron.

Первый дизайн ядер на основе этой архитектуры представляет собой сочетание центрального процессора CPU с частотой до 3,5 ГГц (обладающего 2—4 ядрами) и всокопроизводительного графического процессора GPU с частотой до 1,35 ГГц. Также в чип интегрирован северный мост набора системной логики (контроллер PCI Express 2.0 и двухканальный контроллер памяти стандарта DDR3 SDRAM с частотой до 1333 МГц). Каждое ядро имеет по 256 КБ кеша второго уровня и до 8 МБ объединенного кеша третьего уровня. Процессор находится на одной кремниевой подложке площадью 216 кв.мм. Энергопотребление данного дизайна не выходит за пределы 95 Вт для топовых моделей. До конца 2011 года Intel планирует перевести процессоры всех ценовых сегментов на архитектуру Sandy Bridge.

Новая микроархитектура несёт поддержку новых SIMD (инструкций для работы с векторными вычислениями Advanced Vector Extensions, **AVX**), которые дополнят расширения SSE (новый набор, оставаясь обратно совместимым с SSE, увеличивает разрядность регистров в два раза — до 256 бит, а также даёт в распоряжение программистов дополнительные трёх- и четырёхоперандные команды). При этом Intel обещает, что использование AVX будет способно поднять скорость работы некоторых алгоритмов на величину, достигающую 90 %.

3.2.8. Архитектура Haswell

Революционные изменения произойдут в 2012 году с появлением процессора с кодовым именем Haswell. Хотя базовое количество ядер в CPU останется равным 8, нас ожидает кардинальная переделка архитектуры кэш-памяти и внедрение принципиально новых механизмов энергосбережения. Также предполагается, что Haswell получит интегрированный векторный сопроцессор. Пересмотр ожидает и набор базовых инструкций, к которым добавятся команды семейства FMA (Fused Multiply-Add), позволяющие работать с четырьмя операндами и совершать одновременное умножение и сложение.

Haswell — кодовое название процессорной микроархитектуры, разрабатываемой Intel, которая планируется как преемница Sandy Bridge. Разработана для

22-нанометровой производственной технологии. Релиз планируется в 2012 году. Первые процессоры на данной архитектуре ожидается в 1 квартале 2013 года.

3.2.9. Микроархитектура Intel Itanium

В чем заключается революционность Itanium?

Сама Intel по праву называет Itanium «самой значительной новой разработкой Intel в области микропроцессорной архитектуры с момента выпуска процессора i386 в 1985 году». Революционность состоит в отказе от давно морально устаревшей системы команд x86 (она в ходу с 1978 года) и в радикальном переходе к новой архитектуре, свободной от «пережитков прошлого». Революционной в Itanium является не только 64-разрядность (у 32-разрядных процессоров теоретическое ограничение объема адресуемой оперативной памяти составляет 4 Гбайт, а у 64-битных — несколько терабайт), но и явный параллелизм EPIC (Explicity Parallel Instruction Computing). По сути, архитектура IA-64 (именно так называется технология, по которой проектируются процессоры Itanium, McKinley и последующие) впитала в себя все лучшие идеи:

- VLIW (Very Long Instruction Word, архитектура с длинными командами),
- устранение ветвлений,
- улучшенный механизм предварительной подачи данных и пр.

Среди характеристик первого процессора архитектуры IA-64 можно также отметить увеличенное адресное пространство, обнаружение и исправление ошибок. В принципе, о силиконовой составляющей процессора Itanium давно уже известно многое. Думаю, что к моменту выхода эта информация будет разниться только в деталях: станет известна точная частота (Intel может, к примеру, выдать не обещанные 800, а все 1000 МГц) и будет определена ценовая политика. Картридж Itanium предназначен для установки в Slot M — комбинированный процессорный разъем, сочетающий достоинства как Socket, так и Slot. Сигнальная матрично-штырьковая часть разведена с силовой частью, по которой подается питание, с тем чтобы исключить помехи. На обратной стороне процессорного картриджа расположена массивная теплоотводная пластина, позволяющая равномерно распределять по всей поверхности процессора ватты, излучаемые в воздух. К тому же сильно нагревающиеся блоки процессора тоже размещены равномерно. Согласно предварительным данным Itanium поддерживает частоту шины памяти 266 МГц. Архитектура Itanium подразумевает использование 2 или 4 Мбайт кэш-памяти третьего уровня. Статическая кэш-память новой конструкции размещена на одной плате с ядром процессора и работает с ним на одинаковой тактовой частоте.

3.2.10. Микроархитектура Intel IA-64

Это модификация Intel IA-32 под 64-разрядные процессоры.

3.3. Программная модель IA-32

Любая выполняющаяся программа получает в свое распоряжение определенный набор ресурсов процессора. Эти ресурсы необходимы для обработки и хранения в памяти команд и данных программы, а также информации о текущем состоянии программы и процессора. Программную модель процессора в архитектуре IA-32 процессоров Intel составляет следующий набор ресурсов (рис. 2.5):

- пространство адресуемой памяти до 232 - 1 байт (4 Гбайт), для Pentium III/IV — до 236 - 1 байт (64 Гбайт);
- 8 регистров целочисленного устройства для хранения данных общего назначения;
- 6 сегментных регистров;
- набор регистров состояния и управления;
- 8 регистров устройства вычислений с плавающей точкой (сопроцессора);
- 8 регистров целочисленного MMX-расширения, отображенных на регистры сопроцессора (впервые появились в архитектуре процессора Pentium MMX);
- 8 регистров XMM-расширения с плавающей точкой (впервые появились в архитектуре процессора Pentium III);
- программный стек — специальная информационная структура, работа с которой предусмотрена на уровне машинных команд (более подробно она будет обсуждена позже).

Это основной набор ресурсов. Кроме того, к ресурсам, поддерживаемым архитектурой IA-32, необходимо отнести порты ввода-вывода, счетчики мониторинга производительности.

3.3.1. Адресация памяти в IA_32

В семействе процессоров IA-32 выбор метода обращения к памяти определяется режимом работы процессора. Возможны 3 режима:

- **Реальный** – в этом режиме адрес формируется аналогично i8086, т.е. при формировании адреса используются 16-ти разрядные смещения и 16-ти разрядные сегментные адреса, которые хранятся в сегментных регистрах. При их сложении по приведенной выше схеме получаются 20-ти разрядные

физические адреса, поэтому в этом режиме доступен только первый мегабайт оперативной памяти. Реальный режим работы процессора используется в операционной системе MS DOS, которая устарела. В настоящее время режим практически не используется.

- **Защищенный** – в этом режиме используется 32-х разрядная адресация, предусматривающая несколько вариантов защиты, откуда и появилось название этого режима;
- **Виртуальный** – в этом режиме процессор моделирует псевдоодновременную работу нескольких виртуальных процессоров i8086. В настоящее время режим устарел и практически не используется.
- **Системного управления** (System Management Mode - SMM) — это новый режим работы процессора, впервые появившийся в процессоре Pentium. Он обеспечивает операционную систему механизмом для выполнения машинно-зависимых функций, таких как перевод компьютера в режим пониженного энергопотребления или выполнения действий по защите системы. Для перехода в данный режим процессор должен получить специальный сигнал SMI от усовершенствованного программируемого контроллера прерываний (Advanced Programmable Interrupt Controller - APIC), при этом сохраняется состояние вычислительной среды процессора. Функционирование процессора в этом режиме подобно его работе в режиме реальных адресов. Возврат из этого режима производится специальной командой процессора.

Требование сохранить возможность выполнения программ, использующих 16-ти разрядную адресацию, привело к тому, что схема 32-х разрядной адресации является многокомпонентной.

В этом режиме по-прежнему используется сегментная организация памяти, но размер сегмента уже не ограничивается 64 Кб, а теоретически может достигать 4 Гб. 32-х разрядный адрес базы сегмента хранится не в виде сегментного адреса в сегментном регистре, как при 16-ти разрядной адресации, а полностью в специальных внутренних регистрах процессора – дескрипторах. Номер дескриптора заносится в 14 бит сегментного регистра, который в этом режиме называется селектором. Один бит селектора из этих 14-ти отвечает за выбор таблицы локальных или глобальных дескрипторов.

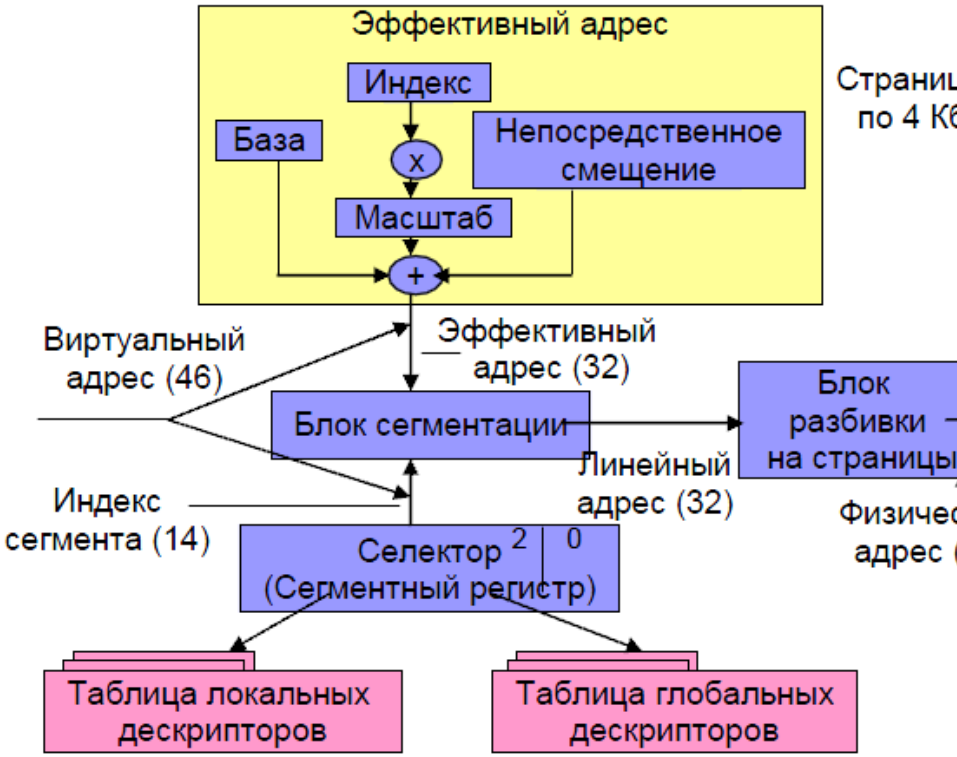


Таблица локальных дескрипторов содержит дескрипторы сегментов приложения, а таблица глобальных – дескрипторы сегментов программ операционной системы. Оставшиеся два бита селектора содержат код уровня привилегий сегмента, который проверяется при обращениях из других программ. Таким образом, реализуется защита сегментов.

14 бит селектора и 32 бита эффективного или исполнительного адреса, формируемого на основе машинной команды, объединяются в 46-ти разрядный виртуальный адрес.

Сумма 32-х разрядного базового адреса сегмента и 32-х разрядного эффективного адреса образует 32-х разрядный линейный адрес. Физический же адрес определяется по таблице страниц на основе линейного.

Соответственно различают несколько адресных пространств:

- виртуальное – 64 Тб;
- линейное – 4 Гб;
- физическое – 4 Гб.

При создании приложений Windows в основном используется модель памяти **Flat** «плоская». Эта модель подразумевает, что каждому приложению отводится линейное адресное пространство объемом 2 Гб, а остальные 2 Гб предоставляются операционной системе. Базовый адрес в дескрипторах всех сегментов приложения устанавливается равным 0. В езультате все сегменты приложения «перекрываются». Программа, данные и стек размещаются в разных местах памяти за счет азличных смещений. Разделение памяти между приложениями осуществляется операционной системой, которая размещает страницы приложений с одинаковыми линейными адресами в разных местах оперативной памяти. Следовательно и защита сегментов при этой модели не работает.

Чтобы предотвратить взаимное влияние выполняющихся программ друг на друга им выделяются изолированные участки памяти (т.е. код и данные программ находятся во взаимно несмежных сегментах). В защищенном режиме работают такие ОС, как MS Windows и Linux.

В типичной программе, написанной для защищенного режима есть 3 сегмента: кода, данных и стека, информация о которых хранится в трех перечисленных ниже сегментных регистрах.

- В регистре CS хранится указатель на дескриптор сегмента кода программы.
- В регистре DS хранится указатель на дескриптор сегмента данных программы.
- В регистре SS хранится указатель на дескриптор сегмента стека программы.

3.3.2. Наборы регистров

Регистрами называются области высокоскоростной памяти, расположенные внутри процессора в непосредственной близости от его исполнительного ядра. Доступ к ним осуществляется несравнимо быстрее, чем к ячейкам оперативной памяти. Соответственно, машинные команды с операндами в регистрах выполняются максимально быстро, поэтому в программах на языке ассемблера регистры используются очень интенсивно. К сожалению, архитектура IA-32 предоставляет в распоряжение программиста не слишком много регистров, поэтому

они являются критически важным ресурсом и за их содержимым приходится следить очень внимательно.

Большинство регистров имеют определенное функциональное назначение. С точки зрения программиста их можно разделить на две большие группы.

Первую группу образуют пользовательские регистры, к которым относятся:

- **Регистры общего назначения (РОН)** - EAX/AX/AH/AL, EBX/BX/BH/BL, EDX/DX/DH/DL, ECX/CX/CH/CL, EBP/BP, ESI/SI, EDI/DI, ESP/SP предназначены для хранения данных и адресов, программист может их использовать (с определенными ограничениями) для реализации своих алгоритмов.
- **Сегментные регистры** - CS, DS, SS, ES, FS, GS используются для хранения адресов сегментов в памяти.
- **Регистры сопроцессора** - ST(0), ST(1), ST(2), ST(3), ST(4), ST(5), ST(6), ST(7) предназначены для написания программ, использующих тип данных с плавающей точкой.
- **Целочисленные регистры MMX-расширения** - MMXO, MMXl, MMX2, MMX3, MMX4, MMX5, MMX6, MMX7;
- **Регистры XMM-расширения с плавающей точкой** - XMMO, XMM1, XMM2, XMM3, XMM4,XMM5,XMM6,XMM7;
- **Регистры состояния и управления** (регистр флагов E FLAGS/FLAGS и регистр указатель команды EIP/IP) содержат информацию о состоянии процессора исполняемой программы и позволяют изменить это состояние.

Во вторую группу входят системные регистры, то есть регистры, предназначенные для поддержания различных режимов работы, сервисных функций, а также регистры, специфичные для определенной модели процессора. Перечислим системные регистры, поддерживаемые IA-32:

- **Управляющие регистры** – CR0.. .CR4. Они определяют режим работы процессора и характеристики текущей исполняемой задачи.
- **Регистры управления памятью** - GDTR, IDTR, LDTR и TR используются в защищенном режиме работы процессора для локализации управляющих структур этого режима.
- **Отладочные регистры** DR0.. .DR7 предназначены для мониторинга и управления различными аспектами отладки;
- **Регистры типов областей памяти** MTRR используются для аппаратного управления кэшированием в целях назначения соответствующих свойств областям памяти.

- **Машинно-зависимые регистры** MSR используются для управления процессором, контроля за его производительностью, получения информации обошибках.

Почему в обозначениях многих из регистров общего назначения присутствует наклонная разделительная черта? Это не разные регистры — это части одного большого 32-разрядного регистра, но их можно использовать в программе как отдельные объекты. Зачем так сделано? Чтобы обеспечить работоспособность программ, написанных для прежних 16-разрядных моделей процессоров фирмы Intel начиная с i8086.

Многие из приведенных регистров предназначены для работы с определенны ми вычислительными подсистемами процессора: сопроцессором, MMX-расширениями. Поэтому их целесообразно рассматривать в соответствующем контексте.

Так как первая часть учебника посвящена вопросам программирования целочисленной подсистемы процессора, то в данной главе описываются регистры, обеспечивающие ее функционирование. В дальнейшем при обсуждении новых вычислительных подсистем процессора будут рассмотрены и другие регистры из перечисленных ранее.

3.4. Целочисленный процессор

При работе в защищенном режиме процессоры семейства IA-32 могут адресовать до 4 Гбайт оперативной памяти. Такой диапазон адресов определяется разрядностью внутренних регистров процессора. Поскольку регистры 32-разрядные, в них могут храниться значения от 0 до 2^{32} -1.

Регистрами называют участки высокоскоростной памяти, расположенные внутри ЦП и предназначенные для оперативного хранения данных и быстрого доступа к ним со стороны внутренних компонентов процессора. Например, при выполнении оптимизации циклов программы по скорости, переменные, к которым выполняется доступ внутри цикла, располагают в регистрах процессора, а не в памяти.

Структура основных программных регистров (Program execution registers) процессора семейства IA-32 и их названия, определенные специалистами фирмы Intel.

Структура основных программных регистров процессора семейства IA-32

Существуют:

- 8 регистров общего назначения,
- регистр состояния процессора (или регистр флагов EFLAGS),
- регистр указателя команд (EIP).
- 6 сегментных регистров,
- Управляющие регистры

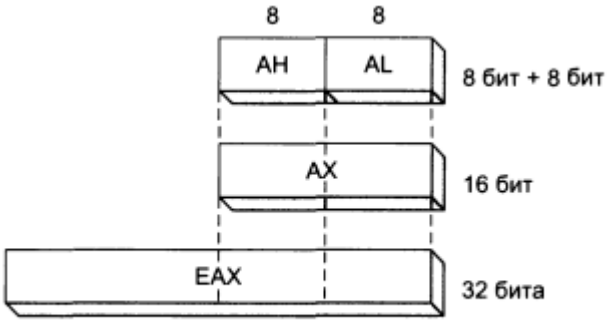
3.4.1. Регистры общего назначения (РОН)

Регистры EAX, EBX, ECX, EDX называют рабочими регистрами. Так как эти регистры физически находятся в микропроцессоре внутри арифметико-логического устройства (АЛУ), то их еще называют регистрами АЛУ:

- **eax/ax/ah/al (Accumulator register)** — аккумулятор. Применяется для хранения промежуточных данных. В некоторых командах использование этого регистра обязательно.
- **ebx/bx/bh/bl (Base register)** — базовый регистр. Применяется для хранения базового адреса некоторого объекта в памяти.

- **ecx/cx/ch/cl (Count register)** — регистр-счетчик. Применяется в командах, производящих некоторые повторяющиеся действия. Его использование зачастую неявно и скрыто в алгоритме работы соответствующей команды. К примеру, команда организации цикла **loop** кроме передачи управления команде, находящейся по некоторому адресу, анализирует и уменьшает на единицу значение регистра ecx/cx;
- **edx/dx/dh/dl (Data register)** — регистр данных. Так же, как и регистр eax/ax/ah/al, он хранит промежуточные данные. В некоторых командах его использование обязательно; для некоторых команд это происходит неявно.

РОНы используются в основном для выполнения арифметических операций и пересылки данных. Как показано на рисунке, к каждому РОНу можно обратиться как к 32-разрядному или как к 16-разрядному регистру.



Особенности обращения к регистрам общего назначения

К некоторым 16-разрядным регистрам можно обращаться как к двум 8-разрядным регистрам. Например, регистр EAX является 32-разрядным, однако его младшие 16-разрядов находятся в регистре AX. Старшие 8-разрядов регистра AX находятся в регистре AH, а младшие 8-разрядов — в регистре AL.

В таблицах показаны особенности обращения к другим регистрам общего назначения, которые мы условно назвали основными и дополнительными

Обращение к основным регистрам общего назначения

32-разрядный регистр	*16-разрядный регистр*	*8-разрядный регистр (старший байт)*	*8-разр (мла*
EAX	AX	AH	
EBX	BX	BH	
ECX	CX	CH	
EDX	DX	DH	

Обращение к дополнительным регистрам общего наз

32-разрядный регистр	*16-разрядный регистр*
ESI	SI
EDI	DI
EBP	BP
ESP	SP

Особенности использования регистров. При выполнении команд процессором часть регистров общего назначения имеют особое значение.

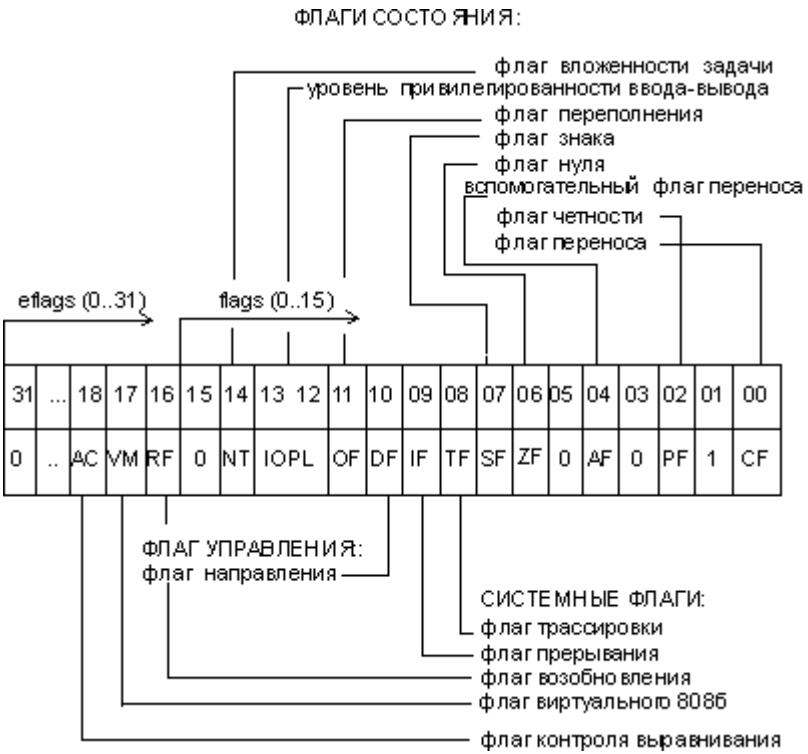
- Содержимое регистра EAX автоматически используется при выполнении команд умножения и деления. Поскольку этот регистр обычно связан с выполнением арифметических команд, его часто называют расширенным регистром аккумулятора (extended accumulator).
- Регистр ECX автоматически используется процессором в качестве счетчика цикла.

Регистры EDI, ESI - индексные регистры, используются для поддержки так называемых цепочечных операций, то есть операций, производящих последовательную обработку цепочек элементов, каждый из которых может иметь длину 32, 16 или 8 бит:

- **esi/si (Source Index register)** — индекс источника. Этот регистр в цепочечных операциях содержит текущий адрес элемента в цепочке-источнике;
- **edi/di (Destination Index register)** — индекс приемника (получателя). Этот регистр в цепочечных операциях содержит текущий адрес в цепочке-приемнике.

3.4.2. Регистры флагов EFLAGS

eflags/flags (flags register) — регистр флагов. Разрядность eflags/flags — 32/16 бит. Отдельные биты данного регистра имеют определенное функциональное назначение и называются флагами. Младшая часть этого регистра полностью аналогична регистру flags для i8086. Содержимое регистра eflags.



Каждый бит этого регистра отвечает либо за особенности выполнения некоторых команд ЦП, либо отражает результат выполнения команд блоком АЛУ процессора. Для анализа битов этого регистра предусмотрены специальные команды процессора.

- Управляющие флаги. Состояние битов регистра EFLAGS, соответствующих управляющим флагам, программист может изменить с помощью специаль-

ных команд процессора. Эти флаги управляют процессом выполнения некоторых команд ЦП. В качестве примера можно привести флаги управления направлением пересьшки данных (Direction) и прерыванием (Interrupt).

- Флаги состояния. Эти флаги отражают результат выполнения арифметической или логической команды ЦП. Их название, описание и сокращенное обозначение приведены ниже^
- Флаг переноса (Carryflag, или CF) устанавливается в случае, если при выполнении беззнаковой арифметической операции получается число, разрядность которого превышает разрядность выделенногодля него поля результата.
- Флаг переполнения {Overflow flag, или OF) устанавливается в случае, если при выполнении арифметической операции со знаком получается число, разрядность которого превышает разрядность выделенногодля него поля результата.
- Флаг знака (Sign f!ag, или SF) устанавливается, если при выполнении арифметической или логической операции получается отрицательное число (т.е. старший бит результата равен 1).
- Флаг нуля {Zero f!ag, или ZF) устанавливается, если при выполнении арифметической или логической операции получается число, равное нулю (т.е. все биты результата равны 0).
- Флаг служебного переноса {Auxiliary Carry, или AF) устанавливается, если при выполнении арифметической операции с 8-разрядным операндом происходит перенос из третьего бита в четвертый.
- Флаг четности {Parity flag, или PF) устанавливается в случае, если в результате выполнения арифметической или логической операции получается число, содержащее четное количеств единичных битов.

Исходя из особенностей использования, флаги регистра eflags/flags можно разделить на три группы:

8 флагов состояния. Эти флаги могут изменяться после выполнения машинных команд. Флаги состояния регистра eflags отражают особенности результата исполнения арифметических или логических операций. Это дает возможность анализировать состояние вычислительного процесса и реагировать на него с помощью команд условных переходов и вызовов подпрограмм.

Мнемоника	Флаг	№ в eflags	Содержание и назначение

cf	Флаг переноса (Carry Flag)	0	• 1 — арифметическая операция произвела перенос из старшего бита результата. Старшим является 7, 15 или 31-й бит в зависимости от размерности операнда; • 0 — переноса не было.
pf	Флаг паритета (Parity Flag)	2	• 1 — 8 младших разрядов (этот флаг — только для 8 младших разрядов операнда любого размера) результата содержат четное число единиц; • 0 — 8 младших разрядов результата содержат нечетное число единиц.
af	Вспомогательный флаг переноса (Auxiliary carry Flag)	4	Только для команд, работающих с BCD-числами. Фиксирует факт заема из младшей тетрады результата: • 1 — в результате операции сложения был произведен перенос из разряда 3 в старший разряд или при вычитании был заем в разряд 3 младшей тетрады из значения в старшей тетраде; • 0 — переносов и заемов в(из) 3 разряд(а) младшей тетрады результата не было.
zf	Флаг нуля (Zero Flag)	6	• 1 — результат нулевой; • 0 — результат ненулевой.
sf	Флаг знака (Sign Flag)	7	Отражает состояние старшего бита результата (биты 7, 15 или 31 для 8, 16 или 32-разрядных операндов соответственно): • 1 — старший бит результата равен 1; • 0 — старший бит результата равен 0.
of	Флаг	11	Флаг of используется для фиксирования

	переполнения (Overflow Flag)		факта потери значащего бита при арифметических операциях: • 1 — в результате операции происходит перенос (заем) в(из) старшего, знакового бита результата (биты 7, 15 или 31 для 8, 16 или 32-разрядных операндов соответственно); • 0 — в результате операции не происходит переноса (заема) в(из) старшего, знакового бита результата.
iopl	Уровень привилегий ввода-вывода (Input/Output Privilege Level)	12, 13	Используется в защищенном режиме работы микропроцессора для контроля доступа к командам ввода-вывода в зависимости от привилегированности задач.
nt	флажок вложенности задачи (Nested Task)	14	Используется в защищенном режиме работы микропроцессора для фиксации того факта, что одна задача вложена в другую.

1 флаг управления. Обозначается df (Directory Flag). Он находится в 10-м бите регистра eflags и используется цепочечными командами. Значение флага df определяет направление поэлементной обработки в этих операциях: от начала строки к концу (df = 0) либо наоборот, от конца строки к ее началу (df = 1). Для работы с флагом df существуют специальные команды: cld (снять флаг df) и std (установить флаг df). Применение этих команд позволяет привести флаг df в соответствие с алгоритмом и обеспечить автоматическое увеличение или уменьшение счетчиков при выполнении операций со строками;

5 системных флагов, управляющих вводом/выводом, маскируемыми прерываниями, отладкой, переключением между задачами и виртуальным режимом 8086. Прикладным программам не рекомендуется модифицировать без необходимости эти флаги, так как в большинстве случаев это приведет к прерыванию работы программы.

Мнемоника	Флаг	№ в eflags	Содержание и назначение
tf	Флаг трассировки (Trace Flag)	8	Предназначен для организации пошаговой работы микропроцессора. • 1 — микропроцессор генерирует прерывание с номером 1 после выполнения каждой машинной команды. Может использоваться при отладке программ, в частности отладчиками; • 0 — обычная работа
if	Флаг прерывания (Interrupt enable Flag)	9	Предназначен для разрешения или запрещения (маскирования) аппаратных прерываний (прерываний по входу INTR). • 1 — аппаратные прерывания разрешены; • 0 — аппаратные прерывания запрещены
rf	Флаг возобновления (Resume Flag)	16	Используется при обработке прерываний от регистров отладки.
vm	Флаг виртуального (Virtual 8086 Mode)	17	Признак работы микропроцессора в режиме виртуального 8086. • 1 — процессор работает в режиме виртуального 8086; • 0 — процессор работает в реальном или защищенном режиме
ac	Флаг контроля выравнивания (Alignment Check)	18	Предназначен для разрешения контроля выравнивания при обращениях к памяти. Используется совместно с битом am в системном регистре cr0. К примеру, Pentium разрешает размещать команды и данные с любого адреса. Если требуется контролировать выравнивание данных и команд по адресам кратным 2 или 4, то установка данных битов приведет к тому, что все обращения

			по некратным адресам будут возбуждать исключительную ситуацию.

eip/ip (Instraction Pointer register) — регистр-указатель команд. Регистр eip/ip имеет разрядность 32/16 бит и содержит смещение следующей подлежащей выполнению команды относительно содержимого сегментного регистра cs в текущем сегменте команд. 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. Возникновение прерываний также приводит к модификации регистра eip/ip.

3.4.3. Регистр указателя команд

В регистре EIP, который также называют регистром указателя команд, хранится адрес следующей выполняемой команды. В процессоре есть несколько команд, которые влияют на содержимое этого регистра. Изменение адреса, хранящегося в регистре EIP, вызывает передачу управления на новый участок программы.

3.4.4. Сегментные регистры

В программной модели микропроцессора имеется 6 сегментных регистров: CS, SS, DS, ES, GS, FS.

Их существование обусловлено спецификой организации и использования оперативной памяти микропроцессорами Intel. Она заключается в том, что микропроцессор аппаратно поддерживает структурную организацию программы в виде 3 частей, называемых сегментами. Соответственно, такая организация памяти называется сегментной.

Для того чтобы указать на сегменты, к которым программа имеет доступ в конкретный момент времени, и предназначены сегментные регистры. Фактически в этих регистрах содержатся адреса памяти, с которых начинаются соответствующие сегменты. Логика обработки машинной команды построена так, что при выборке команды, доступе к данным программы или к стеку неявно используются адреса во вполне определенных сегментных регистрах. Микропроцессор поддерживает следующие типы сегментов:

Эти регистры используются в качестве базовых при обращении к заранее распределенным областям оперативной памяти, которые называются сегментами. Существует 3 типа сегментов и, соответственно, сегментных регистров:

- кода (CS), в них хранятся только команды процессора, т.е. машинный код программы;
- данных (DS, ES, FS и GS). В них хранятся области памяти, выделяемые под переменные программы и под данные;
- стека {SS), в них хранится системная область памяти, называемая стеком, в которой распределяются локальные (временные) переменные программы и параметры, передаваемые функциям при их вызове.

Сегмент кода. Содержит команды программы. Для доступа к этому сегменту служит регистр CS (code segment register) — сегментный регистр кода. Он содержит адрес начала сегмента с машинными командами, к которому имеет доступ микропроцессор (то есть эти команды загружаются в конвейер микропроцессора).

Суть сегментной адресации заключается в следующем. Обращение к памяти осуществляется исключительно с помощью сегментов - логических образований, накладываемых на те или иные участки физической памяти. Исполнительный адрес любой ячейки памяти вычисляется процессором путем сложения начального адреса сегмента, в котором располагается эта ячейка, со смещением к ней (в байтах) от начала сегмента. Это смещение иногда называют относительным адресом. Образование физического адреса из сегментного адреса и смещения:

Начальный адрес сегмента без четырех младших битов, т.е. деленный на 16, помещается в один из сегментных регистров и называется сегментным адресом. Сам же начальный адрес хранится в специальном внутреннем регистре

процессора, называемом теневым регистром. Для каждого сегментного регистра имеется свой теневой регистр; начальный адрес сегмента загружается в него процессором в тот момент, когда программа заносит в соответствующий сегментный регистр новое значение сегментного адреса.

Сегмент данных. Содержит обрабатываемые программой данные. Для доступа к этому сегменту служит регистр ds (data segment register) — сегментный регистр данных, который хранит адрес начала сегмента данных текущей программы.

Сегмент стека. Этот сегмент представляет собой область памяти, называемую стеком. Работу со стеком микропроцессор организует по принципу LIFO (Last In First Out – последним пришел, первым ушел). Для доступа к этому сегменту служит регистр SS (stack segment register) — сегментный регистр стека, содержащий **адрес начала сегмента стека**. Для работы со стеком в системе команд микропроцессора есть специальные команды, а в программной модели микропроцессора для этого существуют специальные регистры:

- С помощью регистра **ESP/SP (Stack Pointer register**) происходит обращение к данным, хранящимся в стеке. Этот регистр обычно никогда не используется для выполнения обычных арифметических операций и команд пересылки данных. Его часто иазывают расширенным **регистром указателя стека** (extended stack pointer).
- Регистр **EBP/BP (Base Pointer register)** обычно используется для **произвольной адресации** в стеке параметров и локальных переменных. Регистр ESP - указатель стека, автоматически модифицируется командами PUSH, POP, RET, CALL. Явно используется реже.

Дополнительный сегмент данных. Неявно алгоритмы выполнения большинства машинных команд предполагают, что обрабатываемые ими данные расположены в сегменте данных, адрес которого находится в сегментном регистре ds.

Если программе недостаточно одного сегмента данных, то она имеет возможность использовать еще 3 дополнительных сегмента данных. Но в отличие от основного сегмента данных, адрес которого содержится в сегментном регистре ds, при использовании дополнительных сегментов данных их адреса требуется указывать явно с помощью специальных префиксов переопределения сегментов в команде. Адреса дополнительных сегментов данных должны содержаться в регистрах es, gs, fs (extension data segment registers).

3.4.5. Управляющие регистры

Регистр CR0.

- 0-й бит, разрешение защиты (PE). Переводит процессор в защищенный режим.
- 1-й бит, мониторинг сопроцессора (MP). Вызывает исключение 7 по каждой команде WAIT.
- 2-й бит, эмуляция сопроцессора (EM). Вызывает исключение 7 по каждой команде сопроцессора.
- 3-й бит, бит переключения задач (TS). Позволяет определить, относится данный контекст сопроцессора к текущей задаче или нет. Вызывает исключение 7 при выполнении следующей команды сопроцессора.
- 4-й бит, индикатор поддержки инструкций сопроцессора (ET).
- 5-й бит, разрешение стандартного механизма сообщений об ошибке сопроцессора (NE).
- 5-15-й бит, не используются.
- 16-й бит, разрешение защиты от записи на уровне привилегий супервизора (WP).
- 17-й бит, не используется.
- 18-й бит, разрешение контроля выравнивания (AM).
- 19-28-й бит, не используются.
- 29-й бит, запрет сквозной записи кэша и циклов аннулирования (NW).
- 30-й бит, запрет заполнения кэша (CD).
- 31-й бит, включение механизма страничной переадресации.

Регистр CR1 пока не используется.

Регистр CR2 хранит 32-битный линейный адрес, по которому был получен последний отказ страницы памяти.

Регистр CR3 - в старших 20 битах хранится физический базовый адрес таблицы каталога страниц. 3-й бит, кэширование страниц со сквозной записью (PWT). 4-й бит, запрет кэширование страницы (PCD).

Регистр CR4

- 0-й бит, разрешение использования виртуального флага прерываний в режиме V8086 (VME).
- 1-й бит, разрешение использования виртуального флага прерываний в защищенном режиме (PVI).

- 2-й бит, превращение инструкции RDTSC в привилегированную (TSD).
- 3-й бит, разрешение точек останова по обращению к портам ввода-вывода (DE).
- 4-й бит, включает режим адресации с 4-мегабайтными страницами (PSE).
- 5-й бит, включает 36-битное физическое адресное пространство (PAE).
- 6-й бит, разрешение исключения MC (MCE).
- 7-й бит, разрешение глобальной страницы (PGE).
- 8-й бит, разрешает выполнение команды RDPMC (PMC).
- 9-й бит, разрешает команды быстрого сохранения/восстановления состояния сопроцессора (FSR).

3.4.6. Системные адресные регистры

Само название этих регистров говорит о том, что они выполняют специфические функции в системе. Использование системных регистров жестко регламентировано. Именно они обеспечивают работу защищенного режима. Их также можно рассматривать как часть архитектуры микропроцессора, которая намеренно оставлена видимой для того, чтобы квалифицированный системный программист мог выполнить самые низкоуровневые операции. Большинство из системных регистров программно доступны. Не все из них понадобятся в нашем дальнейшем изложении, но, тем не менее, я коротко рассмотрел их с тем, чтобы возбудить у читателя интерес к дальнейшему исследованию архитектуры микропроцессора.

Системные регистры можно разделить на три группы:

- 4 регистра системных адресов;
- 4 регистра управления;
- 8 регистров отладки.

Регистры системных адресов. Эти регистры еще называют регистрами управления памятью. Они предназначены для защиты программ и данных в мультизадачном режиме работы микропроцессора. При работе в защищенном режиме микропроцессора адресное пространство делится на:

- глобальное — общее для всех задач;
- локальное — отдельное для каждой задачи.

Этим разделением и объясняется присутствие в архитектуре микропроцессора следующих системных регистров:

- регистра таблицы глобальных дескрипторов gdtr (Global Descriptor Table Register) имеющего размер 48 бит и содержащего 32-битовый (биты 16—

47) базовый адрес глобальной дескрипторной таблицы GDT и 16-битовое (биты 0—15) значение предела, представляющее собой размер в байтах таблицы GDT;

- регистра таблицы локальных дескрипторов ldtr (Local Descriptor Table Register) имеющего размер 16 бит и содержащего так называемый селектор дескриптора локальной дескрипторной таблицы LDT. Этот селектор является указателем в таблице GDT, который и описывает сегмент, содержащий локальную дескрипторную таблицу LDT;
- регистра таблицы дескрипторов прерываний idtr (Interrupt Descriptor Table Register) имеющего размер 48 бит и содержащего 32-битовый (биты 16–47) базовый адрес дескрипторной таблицы прерываний IDT и 16-битовое (биты 0—15) значение предела, представляющее собой размер в байтах таблицы IDT;
- 16-битового регистра задачи tr (Task Register), который подобно регистру ldtr, содержит селектор, то есть указатель на дескриптор в таблице GDT. Этот дескриптор описывает текущий сегмент состояния задачи (TSS — Task Segment Status). Этот сегмент создается для каждой задачи в системе, имеет жестко регламентированную структуру и содержит контекст (текущее состояние) задачи. Основное назначение сегментов TSS — сохранять текущее состояние задачи в момент переключения на другую задачу.

Регистры управления. В группу регистров управления входят 4 регистра: cr0, cr1, cr2, cr3. Эти регистры предназначены для общего управления системой. Регистры управления доступны только программам с уровнем привилегий 0.

Хотя микропроцессор имеет 4 регистра управления, доступными являются только 3 из них — исключается cr1, функции которого пока не определены (он зарезервирован для будущего использования).

Регистр cr0 содержит системные флаги, управляющие режимами работы микропроцессора и отражающие его состояние глобально, независимо от конкретных выполняющихся задач. Назначение системных флагов:

- pe (Protect Enable), бит 0 — разрешение защищенного режима работы.Состояние этого флага показывает, в каком из двух режимов — реальном (pe=0) или защищенном (pe=1) — работает микропроцессор в данный момент времени.
- mp (Math Present), бит 1 — наличие сопроцессора. Всегда 1.
- ts (Task Switched), бит 3 — переключение задач. Процессор автоматически устанавливает этот бит при переключении на выполнение другой задачи.

- am (Aligment Mask), бит 18 — маска выравнивания. Этот бит разрешает (am = 1) или запрещает (am = 0) контроль выравнивания.
- cd (Cache Disable), бит 30, — запрещение кэш-памяти.С помощью этого бита можно запретить (cd = 1) или разрешить (cd = 0) использование внутренней кэш-памяти (кэш-памяти первого уровня).
- pg (PaGing), бит 31, — разрешение (pg = 1) или запрещение (pg = 0) страничного преобразования.
  Флаг используется при страничной модели организации памяти.

Регистр cr2 используется при страничной организации оперативной памяти для регистрации ситуации, когда текущая команда обратилась по адресу, содержащемуся в странице памяти, отсутствующей в данный момент времени в памяти. В такой ситуации в микропроцессоре возникает исключительная ситуация с номером 14, и линейный 32-битный адрес команды, вызвавшей это исключение, записывается в регистр cr2. Имея эту информацию, обработчик исключения 14 определяет нужную страницу, осуществляет ее подкачку в память и возобновляет нормальную работу программы;

Регистр cr3 также используется при страничной организации памяти. Это так называемый регистр каталога страниц первого уровня. Он содержит 20-битный физический базовый адрес каталога страниц текущей задачи. Этот каталог содержит 1024 32-битных дескриптора, каждый из которых содержит адрес таблицы страниц второго уровня. В свою очередь каждая из таблиц страниц второго уровня содержит 1024 32-битных дескриптора, адресующих страничные кадры в памяти. Размер страничного кадра — 4 Кбайт.

Регистры отладки. Это очень интересная группа регистров, предназначенных для аппаратной отладки. Средства аппаратной отладки впервые появились в микропроцессоре i486. Аппаратно микропроцессор содержит восемь регистров отладки, но реально из них используются только 6. Регистры отладки DR0...DR3 - хранят 32-битные линейные адреса точек останова. DR6 (равносильно DR4) - отражает состояние контрольных точекDR7 (равносильно DR5) - управляет установкой контрольных точек.

Регистры dr0, dr1, dr2, dr3 имеют разрядность 32 бит и предназначены для задания линейных адресов четырех точек прерывания. Используемый при этом механизм следующий: любой формируемый текущей программой адрес сравнивается с адресами в регистрах dr0...dr3, и при совпадении генерируется исключение отладки с номером 1.

Регистр dr6 называется регистром состояния отладки. Биты этого регистра устанавливаются в соответствии с причинами, которые вызвали возникновение последнего исключения с номером 1.

- b0 — если этот бит установлен в 1, то последнее исключение (прерывание) возникло в результате достижения контрольной точки, определенной в регистре dr0;
- b1 — аналогично b0, но для контрольной точки в регистре dr1;
- b2 — аналогично b0, но для контрольной точки в регистре dr2;
- b3 — аналогично b0, но для контрольной точки в регистре dr3;
- bd (бит 13) — служит для защиты регистров отладки;
- bs (бит 14) — устанавливается в 1, если исключение 1 было вызвано состоянием флага tf = 1 в регистре eflags;
- bt (бит 15) устанавливается в 1, если исключение 1 было вызвано переключением на задачу с установленным битом ловушки в TSS t = 1.

Все остальные биты в этом регистре заполняются нулями. Обработчик исключения 1 по содержимому dr6 должен определить причину, по которой произошло исключение, и выполнить необходимые действия.

Регистр dr7 называется регистром управления отладкой. В нем для каждого из 4 регистров контрольных точек отладки имеются поля, с помощью которых можно уточнить следующие условия, при которых следует сгенерировать прерывание:

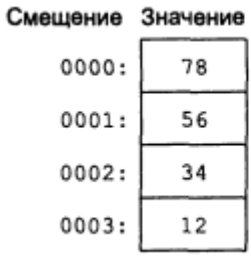
- Место регистрации контрольной точки — только в текущей задаче или в любой задаче. Эти биты занимают младшие восемь бит регистра dr7 (по два бита на каждую контрольную точку (фактически точку прерывания), задаваемую регистрами dr0, dr1, dr2, dr3 соответственно). Первый бит из каждой пары — это так называемое локальное разрешение; его установка говорит о том, что точка прерывания действует если она находится в пределах адресного пространства текущей задачи. Второй бит в каждой паре определяет глобальное разрешение, которое говорит о том, что данная контрольная точка действует в пределах адресных пространств всех задач, находящихся в системе.
- Тип доступа, по которому инициируется прерывание: только при выборке команды, при записи или при записи/чтении данных. Биты, определяющие подобную природу возникновения прерывания, локализуются в старшей части данного регистра.

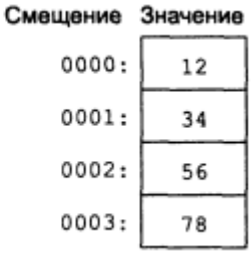
3.4.7. Прямой и обратный порядок следования байтов

В процессорах фирмы Intel при выборке и хранении данных в памяти используется так называемый прямой порядок следования байтов (little endian order). Это означает, что младший байт переменной хранится в памяти по меньшему адресу. Оставшиеся байты переменной хранятся в последующих ячейках памяти в порядке возрастания их старшинства.

В качестве примера рассмотрим двойное слово, значение которого равно 1234 5678h. Предположим, что оно хранится в памяти со смещением 0. Тогда значение 7 8h будет храниться в первом байте со смещением 0, 56h- во втором байте со смещением 1, 34h — в третьем байте со смещением 2, 12h — в четвертом байте со смещением 3, как показано на рисунке слева..

В некоторых типах процессоров используется обратный {big endian order) порядок следования байтов. При этом старший байт переменной хранится по младшему адресу, как показано на рисунке справа.





3.4.8. Виды адресации операндов в памяти

Прямая адресация — это простейший вид адресации операнда в памяти, так как эффективный адрес содержится в самой команде и для его формирования не используется никаких дополнительных источников или регистров. Эффективный адрес берется непосредственно из поля смещения машинной команды, которое может иметь размер 8, 16, 32 бита. Это значение однозначно определяет байт, слово или двойное слово в сегменте данных. Прямая адресация может быть двух типов.

- **Относительная прямая адресация** используется в командах условных переходов для указания относительного адреса перехода. Относительность такого пе- рехода заключается в том, что в поле смещения машинной

команды содержится 8-, 16- или 32-разрядное значение, которое в результате работы команды будет складываться с содержимым регистра указателя команд IP/EIP. В результате такого сложения получается адрес, по которому и осуществляется переход.

- **Абсолютная прямая адресация** — в этом случае эффективный адрес является частью машинной команды, но формируется этот адрес только из значения поля смещения в команде. Для формирования физического адреса операнда в памяти процессор складывает это поле со сдвинутым на четыре бита значением сегментного регистра. Однако такая адресация применяется редко — обычно ячейкам в программе присваиваются символические имена. В процессе трансляции ассемблер вычисляет и подставляет значения смещений этих имен в поле смещения формируемой им машинной команды (см. главу 3). В итоге получается, что машинная команда прямо адресует свой операнд, имея, фактически, в одном из своих полей значение эффективного адреса.

Остальные виды адресации относятся к косвенным. Слово косвенный в названии этих видов адресации означает, что в самой команде может находиться лишь часть эффективного адреса, а остальные его компоненты находятся в регистрах, на которые указывают своим содержимым байт mod r/m и, возможно, байт sib. Косвенная адресация имеет следующие разновидности:

- **Косвенная базовая** (или регистровая) адресация. Эффективный адрес операнда может находиться в любом из регистров общего назначения, кроме SP/ESP и BP/EBP (это специальные регистры для работы с сегментом стека). Синтаксически в команде этот режим адресации выражается заключением имени регистра в квадратные скобки. К примеру, команда mov ax,[ecx] помещает в регистр AX содержимое слова по адресу сегмента данных со смещением, хранящимся в регистре ECX. Так как содержимое регистра легко изменить в ходе работы программы, данный способ адресации позволяет динамически назначить адрес операнда для некоторой машинной команды. Это очень полезно, например, для организации циклических вычислений и для работы с различными структурами данныхтипа таблиц или массивов.
- **Косвенная базовая адресация со смещением** является дополнением предыдущего вида адресации и предназначена для доступа к данным с известным смещением относительно некоторого базового адреса. Этот вид адресации удобно использовать для доступа к элементам структур данных, когда смещение элементов известно заранее на стадии разработки программы, а базовый (начальный) адрес структуры должен вычисляться ди-

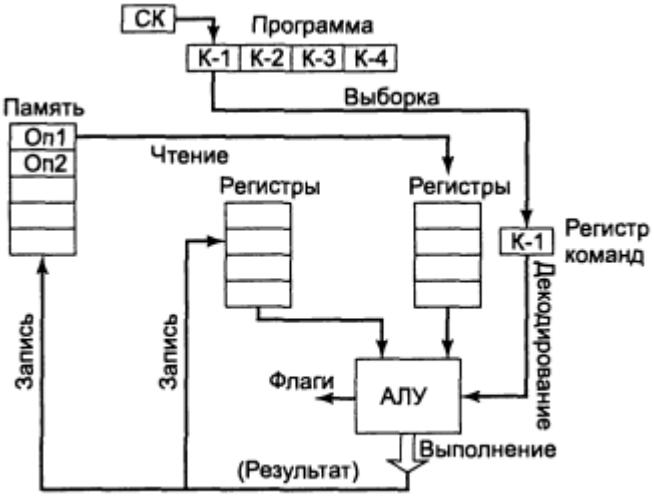
намически на стадии выполнения программы. Модификация содержимого базового регистра позволяет обращаться к одноименным элементам различных экземпляров однотипных структур данных. К примеру, команда mov ax,[edx+3h] пересылает в регистр AX слово из области памяти по адресу, определяемому содержимым EDX + 3h. Команда mov ax,mas[dx] пересылает в регистр AX слово по адресу, определяемому содержимым DX плюс значение идентификатора mas (не забывайте, что транслятор присваивает каждому идентификатору значение, равное смещению этого идентификатора относительно начала сегмента данных).

- Ко**свенная индексная адресация со смещением** очень похожа на косвенную базовую адресацию со смещением. Здесь также для формирования эффективного адреса используется один из регистров общего назначения. Но индексная адресация обладает одной интересной особенностью, которая очень удобна для работы с массивами. Она связана с возможностью так называемого масштабирования содержимого индексного регистра.
- **Косвенная базовая индексная адресация**. Ээффективный адрес формируется как сумма содержимого двух регистров общего назначения: базового и индексного. В качестве этих регистров могут применяться любые регистры общего назначения, при этом часто содержимое индексного регистра масштабируется.
- **Косвенная базовая индексная адресация со смещением**. является дополнением косвенной индексной адресации. Эффективный адрес формируется как сумма трех составляющих: содержимого базового регистра, содержимого индексного регистра и значения поля смещения в команде.

3.4.9. Цикл выполнения команды

Схема цикла выполнения команды ниже:

- По номеру команды из счетчика СК комагда считывается из памяти программ в легистр команд.
- Затем она декодируется и формируются сигналы управления для исполнительного арифметико-логического устройства (АЛУ).
- АЛУ при необходимости выбирает данные из РОН и выполняет команду.
- Результат записывается в регистры или в память данных.
- При выполнении команды АЛУ устанавливает флаги.
- Возможно чтение данных из памяти в РОН.



Упрощенная схема цикла выполнения команды

3.4.10. Распределение адресного пространства

Не следует думать, что термины "адресное пространство" и "оперативная память" эквивалентны. Адресное пространство - это просто набор адресов, которые умеет формировать процессор; совсем не обязательно все эти адреса отвечают реально существующим ячейкам памяти. В зависимости от модификации компьютера и состава его периферийного оборудования, распределение адресного пространства может несколько различаться. Тем не менее, размещение основных компонентов системы довольно строго унифицировано.

Сегментная структура программ. Обращение к памяти осуществляется исключительно посредством сегментов - логических образований, накладываемых на любые участки физического адресного пространства. Начальный адрес сегмента, деленный на 16, т.е. без младшей шестнадцатеричной цифры, заносится в один из сегментных регистров; после этого мы получаем доступ к участку памяти, начинающегося с заданного сегментного адреса.

Каким образом понятие сегментов памяти отражается на структуре программы? Следует заметить, что структура программы определяется, с одной стороны, архитектурой процессора (если обращение к памяти возможно только с помощью сегментов, то и программа, видимо, должна состоять из сегментов), а с

другой - особенностями той операционной системы, под управлением которой эта программа будет выполняться. Наконец, на структуру программы влияют также и правила работы выбранного транслятора - разные трансляторы предъявляют несколько различающиеся требования к исходному тексту программы.

Имеются 3 сегмента:

- сегмент команд с именем code,
- сегмент данных с именем data,
- сегмент стека с именем stk.

Описание каждого сегмента начинается с ключевого слова **segment**, предваряемого некоторым именем, и заканчивается ключевым словом **end**, перед которым указывается то же имя, чтобы транслятор знал, какой именно сегмент мы хотим закончить. Имена сегментов выбираются вполне произвольно.

Порядок описания сегментов в программе, как правило, не имеет значения. Важно понимать, что в оперативную память компьютера сегменты попадут в том же порядке, в каком они описаны в программе (если специальными средствами ассемблера не задать иной порядок загрузки сегментов в память).

Сегменты вводятся в программу с помощью директив ассемблера segment и ends. К директивам ассемблера относятся обозначения начала и конца сегментов segment и ends; ключевые слова, описывающие тип используемых данных (db, dup); специальные описатели сегментов вроде stack и т. д. Директивы служат для передачи транслятору служебной информации, которой он пользуется в процессе трансляции программы. Однако в состав выполнимой программы, состоящей из машинных кодов, эти строки не попадут, так как процессору, выполняющему программу, они не нужны. Другими словами, операторы типа segment и ends не транслируются в машинные коды, а используются лишь самим ассемблером на этапе трансляции программы.

При загрузке программы сегменты размещаются в памяти, как показано на рисунке.

3.4.11. Образ программы в памяти.

Образ программы в памяти начинается с сегмента префикса программы (Program Segment Prefics, PSP), образуемого и заполняемого системой. PSP всегда имеет размер 256 байт; он содержит таблицы и поля данных, используемые системой в процессе выполнения программы. Вслед за PSP располагаются сегменты программы в том порядке, как они объявлены в программе.

Сегментные регистры автоматически инициализируются следующим образом:

- ES и DS указывают на начало PSP (что дает возможность, сохранив их содержимое, обращаться затем в программе к PSP),
- CS - на начало сегмента команд,
- SS - на начало сегмента стека.

В указатель команд IP загружается относительный адрес точки входа в программу (из операнда директивы end), а в указатель стека SP - величина, равная объявленному размеру стека, в результате чего указатель стека указывает на конец стека (точнее, на первое слово за его пределами).

Таким образом, после загрузки программы в память адресуемыми оказываются все сегменты, кроме сегмента данных. Инициализация регистра DS в первых строках программы позволяет сделать адресуемым и этот сегмент.

Важнейшая особенность архитектуры процессоров Intel - адрес любой ячейки памяти состоит из двух слов, одно из которых определяет расположение в памяти соответствующего сегмента, а другое - смещение в пределах этого сегмента.

Сегмент всегда начинается с адреса, кратного 16, т.е. на границе 16-байтового блока памяти (параграфа). Сегментный адрес можно рассматривать, как номер параграфа, с которого начинается данный сегмент. Размер сегмента определяется объемом содержащихся в нем данных, но никогда не может превышать величину 64 Кбайт, что определяется максимально возможной величиной смещения.

Сегментный адрес сегмента команд хранится в регистре CS, а смещение к адресуемому байту - в указателе команд IP. Как уже отмечалось, после загрузки программы в IP заносится смещение первой команды программы; процессор, считав ее из памяти, увеличивает содержимое IP точно на длину этой команды (команды процессоров Intel могут иметь длину от 1 до 6 байт), в результате чего IP указывает на вторую команду программы. Выполнив первую команду, процессор считывает из памяти вторую, опять увеличивая значение IP. В результате в IP всегда находится смещение очередной команды, т. е. команды, следующей за выполняемой. Описанный алгоритм нарушается только при выполнении команд переходов, вызовов подпрограмм и обслуживания прерываний.

Сегментный адрес сегмента данных обычно хранится в регистре DS, а смещение может находится в одном из регистров общего назначения, например, в BX или SI.

3.4.12. Стек

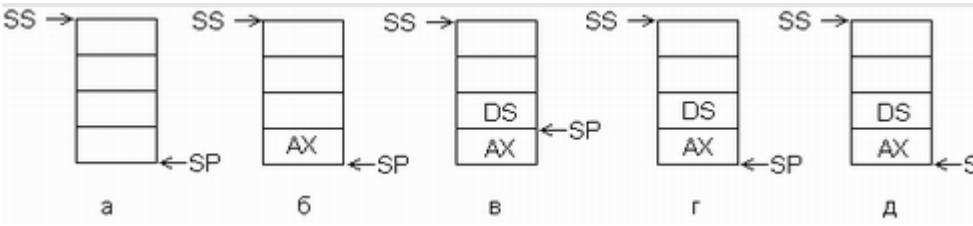
Стеком называют область программы для временного хранения произвольных данных. Разумеется, данные можно сохранять и в сегменте данных, однако в этом случае для каждого сохраняемого на время данного надо заводить отдельную именованную ячейку памяти, что увеличивает размер программы и количество используемых имен. Удобство стека заключается в том, что его область используется многократно, причем сохранение в стеке данных и выборка их оттуда выполняется с помощью эффективных команд push и pop без указания каких-либо имен.

Стек традиционно используется, например, для сохранения содержимого регистров, используемых программой, перед вызовом подпрограммы, которая, в свою очередь, будет использовать регистры процессора "в своих личных це-

лях". Исходное содержимое регистров извлекается из стека после возврата из подпрограммы. Другой распространенный прием - передача подпрограмме требуемых ею параметров через стек. Подпрограмма, зная, в каком порядке помещены в стек параметры, может забрать их оттуда и использовать при своем выполнении.

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.

Элементы стека располагаются в области памяти, отведенной под стек**, начиная со дна стека** (т.е. с его максимального адреса) по последовательно уменьшающимся адресам. Адрес верхнего, доступного элемента хранится в регистре-указателе стека SP. Как и любая другая область памяти программы, стек должен входить в какой-то сегмент или образовывать отдельный сегмент. В любом случае сегментный адрес этого сегмента помещается в сегментный регистр стека SS. Таким образом, пара регистров SS:SP описывают адрес доступной ячейки стека: в SS хранится сегментный адрес стека, а в SP - смещение последнего сохраненного в стеке данного (рис. а). Обратите внимание на то, что в исходном состоянии указатель стека SP указывает на ячейку, лежащую под дном стека и не входящую в него.



Состояния стека:

- а - исходное состояние,
- б - после загрузки одного элемента (в данном примере - содержимого регистра AX),
- в - после загрузки второго элемента (содержимого регистра DS),
- г - после выгрузки одного элемента,
- д - после выгрузки двух элементов и возврата в исходное состояние.

Загрузка в стек осуществляется специальной командой работы со стеком push (протолкнуть). Эта команда сначала уменьшает на 2 содержимое указателя стека, а затем помещает операнд по адресу в SP. Если, например, мы хотим временно сохранить в стеке содержимое регистра AX, следует выполнить команду

```
push AX
```

Стек переходит в состояние, показанное на рис. б. Видно, что указатель стека смещается на два байта вверх (в сторону меньших адресов) и по этому адресу записывается указанный в команде проталкивания операнд. Следующая команда загрузки в стек, например,

```
push DS
```

переведет стек в состояние, показанное на рис. в. В стеке будут теперь храниться два элемента, причем доступным будет только верхний, на который указывает указатель стека SP. Если спустя какое-то время нам понадобилось восстановить исходное содержимое сохраненных в стеке регистров, мы должны выполнить команды выгрузки из стека pop (вытолкнуть):

```
pop DS
pop AX
```

Состояние стека после выполнения первой команды показано на рис. г, а после второй - на рис. д. Для правильного восстановления содержимого регистров выгрузка из стека должна выполняться в порядке, строго противоположном загрузке - сначала выгружается элемент, загруженный последним, затем предыдущий элемент и т.д.

Совсем не обязательно при восстановлении данных помещать их туда, где они были перед сохранением. Например, можно поместить в стек содержимое DS, а извлечь его оттуда в другой сегментный регистр - ES;

```
push DS
pop ES ; Теперь ES=DS, а стек пуст
```

Это распространенный прием для перенесения содержимого одного регистра в другой, особенно, если второй регистр - сегментный.

Обратите внимание на то, что после выгрузки сохраненных в стеке данных они физически не стерлись, а остались в области стека на своих местах. Правда, при "стандартной" работе со стеком они оказываются недоступными. Действительно, поскольку указатель стека SP указывает под дно стека, стек считается

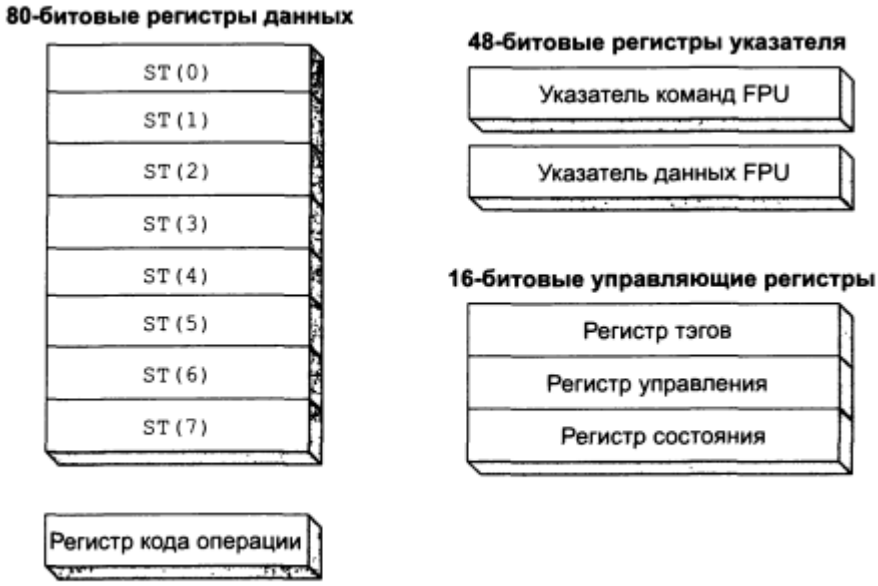
пустым; очередная команда push поместит новое данное на место сохраненного ранее содержимого AX, затерев его. Однако пока стек физически не затерт, сохраненными и уже выбранными из него данными можно пользоваться, если помнить, в каком порядке они расположены в стеке. Этот прием часто используется при работе с подпрограммами.

Какого размера должен быть стек? Это зависит от того, насколько интенсивно он используется в программе. Если, например, планируется хранить в стеке массив объемом 10 000 байт, то стек должен быть не меньше этого размера. При этом надо иметь в виду, что в ряде случаев стек автоматически используется системой, в частности, при выполнении команды прерывания int 21h. По этой команде сначала процессор помещает в стек адрес возврата, а затем операционная система отправляет туда же содержимое регистров и другую информацию, относящуюся к прерванной программе. Поэтому, даже если программа совсем не использует стек, он все же должен присутствовать в программе и иметь размер не менее нескольких десятков слов, например, 128 слов.

3.5. Математический сопроцессор

Семейство процессоров IA-32 содержит так называемый модуль операций с ПТ {FPU - Floating-Point Unit), который используется исключительно для быстрого выполнения этого типа операций. В процессорах Intel386 этот блок был реализован в виде отдельной микросхемы математического сопроцессора, которая обозначалась как Intel387. Однако, начиная с процессоров Intel486, математический сопроцессор стал находиться на одном кристалле с основным процессором.

В модуле FPU содержится 8 внутренних регистров для хранения данных с ПТ, которые называются ST(0), ST(1), ST(2), ST(3), ST(4), ST(5), ST(6) и ST(7). Они образуют стек сопроцессора Остальные регистры, выполняющие функции управления и хранящие указатели, показаны на рисунке.



Регистры математического сопроцессора

Другие регистры. В этом разделе мы просто упомянем о двух других наборах регистров, которые используютсядля поддержки мультимедийных приложений.

- Восемь 64-разрядных регистров, использующихся в так называемых MMX-командах.
- Восемь 128-разрядных XMM-регистров, использующихся при выполнении потоковой обработки данных (SIMD-операций), т.е. когда с помощью одной машинной команды можно выполнить одну и ту же операцию над несколькими данными (Single-Instruction, Multiple-Data, или SIMD).

3.6. MMX-технология

MMX = Multi-Media eXtension – мульти-медиа расширение.

Технология MMX - итог совместной работы создателей архитектуры микропроцессоров Intel и программистов. При ее разработке был исследован широкий круг программ аудиовизуальной обработки информации: обработка изображений, MPEG-видео, синтеза музыки, сжатия речи и ее распознавания, поддержка

видеоконференций, компьютерные игровые программы и т. д. В результате этого анализа были выявлены **основные особенности таких программ:**

- использование данных целого типа небольшой разрядности, например, 8-разрядные графические пиксели и 16-разрядная оцифровка звука;
- короткие циклы с высокими коэффициентами повторяемости;
- большое количество операций умножения и суммирования, в том числе из-за широкого использования быстрого преобразования Фурье;
- применение алгоритмов, требующих интенсивных вычислений;
- широкое использование операций с высоким уровнем параллелизма.

Было отмечено, что в мультимедийных приложениях 80% времени выполнения программы приходится на 10-20% программного кода. Малая разрядность данных требует дополнительных действий при их обработке на 32-разрядном микропроцессоре, не позволяя в то же время использовать всю мощь 32-разрядной архитектуры.

Простым и наглядным примером такого рода обработки может служить изменение значений всех пикселей видеопамяти на определенную величину. Пусть емкость видеопамяти составляет 1 Мбайт, а каждый пиксель кодируется 1 байтом. Тогда для выполнения указанного действия потребуется выполнить примерно 1 млн операций по прибавлению константы к однобайтовому операнду, который выбирается из памяти. Одновременное выполнение таких действий над 4 операндами, что сократило бы количество операций в 4 раза, невозможно в классической архитектуре IA-32 из-за отсутствия соответствующих команд в системе команд и форматов используемых данных.

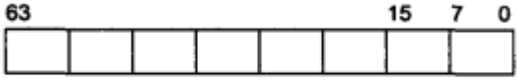
На устранение этих противоречий и были направлены основные усилия разработчиков технологии MMX. Процессор Pentium MMX, в котором впервые была реализована новая технология, был представлен фирмой INTel в январе 1997 года. Он позволил на 10-20 % повысить производительность на стандартных тестах, а для специализированных мультимедийных приложений - на 50 %.

Главной особенностью MMX-технологии является новый принцип обработки информации - обработка по схеме SIMD (Single Instruction Multiple Data) - один поток команд, много потоков данных. Этот вид обработки подразумевает, что с помощью одной команды одна и та же операция выполняется сразу над несколькими операндами, например, производится суммирование нескольких пар слагаемых. Такой подход требует поддержки как со стороны системы команд и форматов данных, так и на аппаратном уровне.

Расширение MMX ориентировано в основном на использование в мультимедийных приложениях. Основная идея MMX заключается в одновременной обработке нескольких элементов данных за одну инструкцию. Для этого используются 64-битные регистры MMX, в которых размещается несколько данных меньшего размера.

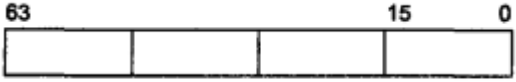
Расширение MMX использует новые типы упакованных данных:

Packed byte - упакованные байты (восемь байт).



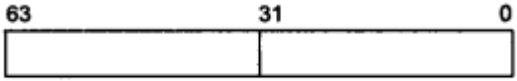
8-байтовый формат представления данных

Packed word - упакованные слова (четыре слова).



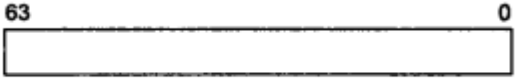
Формат представления данных в виде четырех слов

Packed doubleword - упакованные двойные слова (два двойных слова).



Формат представления данных в виде двух двойных слов

Packed quadword - учетверенное слово.

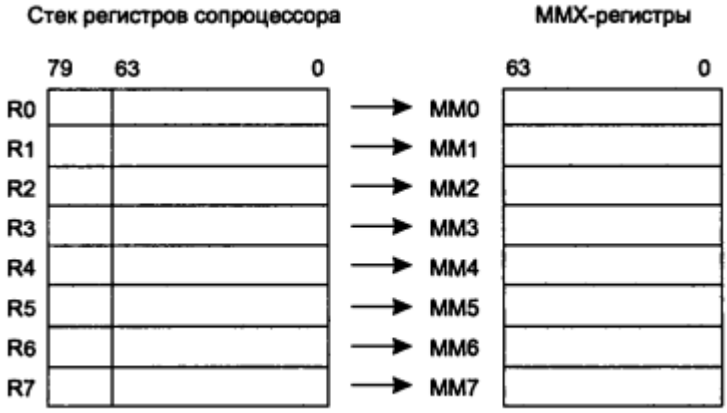


Формат представления данных в виде учетверенного слова

Расширение MMX включает 8 регистров общего пользования (MM0-MM7). Размер регистров составляет 64 бита. Физически эти регистры пользуются младшими битами рабочих регистров сопроцессора.

Команды MMX "портят" регистр состояния и регистр тэгов. По этой причине совместное использование команд MMX и команд сопроцессора может вызвать

определенные трудности. Другими словами, перед каждым использованием команд MMX Вам придется сохранять контекст сопроцессора, а это может весьма замедлить работу программы. Важно отметить также, что команды MMX работают непосредственно с регистрами сопроцессора, а не с указателями на элементы стека.



Соответствие регистров сопроцессора и MMX-расширения

Важное правило, которое следует соблюдать при совместном использовании математического сопроцессора и MMX-расширения: последней выполняемой командой MMX-расширения должна быть команда EMMS. Дело в том, что все MMX-команды выполняются в том же режиме, что и команды сопроцессора с плавающей точкой, что вызывает изменения содержимого регистра состояния (swr) сопроцессора. Команда EMMS обеспечивает корректный переход процессора от выполнения фрагмента программного кода с MMX-командами к обработке обычных команд с плавающей точкой. При этом emms устанавливает значение 1 во всех разрядах регистра состояния. Если фрагмент программы, в,котором есть MMX-команды, не заканчивается командой emms, то все последующие операции с плавающей точкой будут давать некорректные результаты, о чем сигнализирует исключение Stack overflow.

В систему команд MMX были включены 57 новых инструкций. Их использование было призвано, во-первых, уменьшить время выполнения мультимедийных приложений, а во-вторых, минимизировать **конфликты в конвейере**, который становился все более многоступенчатым, что приводило к существенным поте-

рям в производительности из-за конфликтов. Проиллюстрируем это на примере нескольких команд.

Команда PADDSB "Сложение со знаком с насыщением" выполняет сложение одновременно 8 пар однобайтовых операндов. Кроме того, если при выполнении сложения произошло переполнение, то результатом операции будет максимально возможное в этом формате число. Это избавляет программиста от необходимости использования после выполнения каждого сложения команд условных переходов, анализирующих признак переполнения результата, что, в свою очередь, благотворно сказывается на работе конвейера.

8	8	8	8	8	8	8	8
B7	B6	B5	B4	B3	B2	B1	B0
D7	D6	D5	D4	D3	D2	D1	D0
B7+D7	B6+D6	B5+D5	B4+D4	B3+D3	B2+D2	B1+D1	B0+D0

Команда PMADDWD "Умножение с накоплением" эффективна при выполнении вычислений, характерных для обработки звуковой и графической информации. Она одновременно перемножает четыре операнда формата "слово" (16 разрядов), попарно складывает результаты умножений двух младших и двух старших байт и получает два 32-разрядных результата.

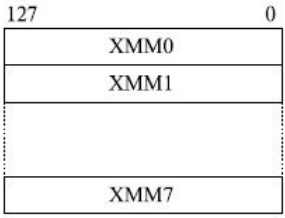
16	16	16	16	
B3	B2	B1	B0	операнд 1
D3	D2	D1	D0	операнд 2
B3 × D3 + B2 × D2		B1 × D1 + B0 × D0		результат

Команда PMAXSB выполняет нахождение максимума одновременно для восьми 8-разрядных операндов. Она позволяет не только выполнить операцию пакетами по 8 байт, но и избежать ветвлений, а следовательно, и "штрафов" за их неправильное предсказание.

8	8	8	8	8	8	8	8	
-2	3	4	-5	1	3	5	-6	операнд 1
3	-1	2	4	-2	1	6	-7	операнд 2
3	3	4	4	1	3	6	-6	результат

3.7. XMM технология

Технология MMX получила свое развитие в микропроцессоре Pentium III с появлением специального аппаратного блока SSE (Streaming SIMD Extension - потоковое SIMD-расширение) обработки информации по схеме SIMD. Новая технология получила название XMM (eXtended Multi-Media). Блок SSE содержит отдельный регистровый файл из восьми 128-разрядных регистров XMM, что позволяет обрабатывать по схеме SIMD числа с плавающей запятой (четыре 32-разрядных числа).



Числа с плавающей запятой имеют следующий формат:

- знак: 1 разряд;
- порядок (смещенный): 8 разрядов;
- мантисса: 23 разряда.

Расширено и количество форматов чисел с фиксированной точкой, обрабатываемых в XMM по схеме SIMD:

- 16 операндов x 8 разрядов;
- 8 операндов x 16 разрядов;
- 4 операнда x 32 разряда;
- 2 операнда x 64 разряда.

Для обработки чисел новых форматов в систему команд дополнительно введены 70 новых команд.

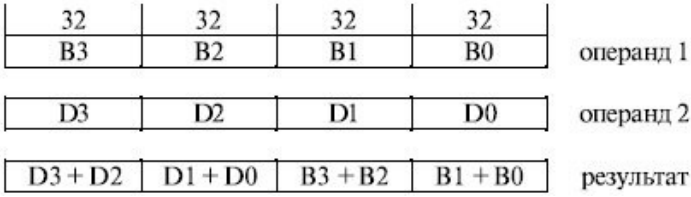
Блок SSE2, включенный в микропроцессор Pentium 4, реализует 144 новые команды. Из этих 144 инструкций 68 расширяют возможности старых SIMD-инструкций по работе с целыми числами, а 76 являются совершенно новыми. Среди последних - инструкции, позволяющие оперировать со 128-разрядными числами (как целыми, так и вещественными с двойной точностью).

Операции SSE2 позволили существенно повысить эффективность применения микропроцессора при реализации трехмерной графики и современных интернет-приложений, обеспечении сжатия и кодирования аудио- и видеоданных и ряда других применений. В результате производительность процессора Pentium 4 при выполнении таких операций стала вдвое выше, чем Pentium III.

Отметим несколько новых по сравнению с MMX инструкций, вошедших в состав команд SSE/SSE2.

Команда ADDSUBPS выполняет сложение второго и четвертого элементов с одинарной точностью с одновременным вычитанием первого и третьего элементов. Эта инструкция полезна при работе с комплексными числами в случае использования соответствующего типа переменных.

Команда HADDPS осуществляет горизонтальное сложение элементов с одинарной точностью. Первый результат является суммой первого и второго элементов первого (исходного) операнда; второй результат - суммой третьего и четвертого элементов первого операнда; третий результат - суммой первого и второго элементов второго операнда (операнда назначения) и, наконец, четвертый результат - суммой третьего и четвертого элементов второго операнда.



Новые возможности в этом направлении обработки информации были обеспечены в технологии SSE3, внедренной в ядре Prescott процессора Pentium 4 добавлением набора из 13 инструкций, и в технологии SSE4 в микропроцессорах семейства INTel Core 2 Duo.

3.8. Система команд

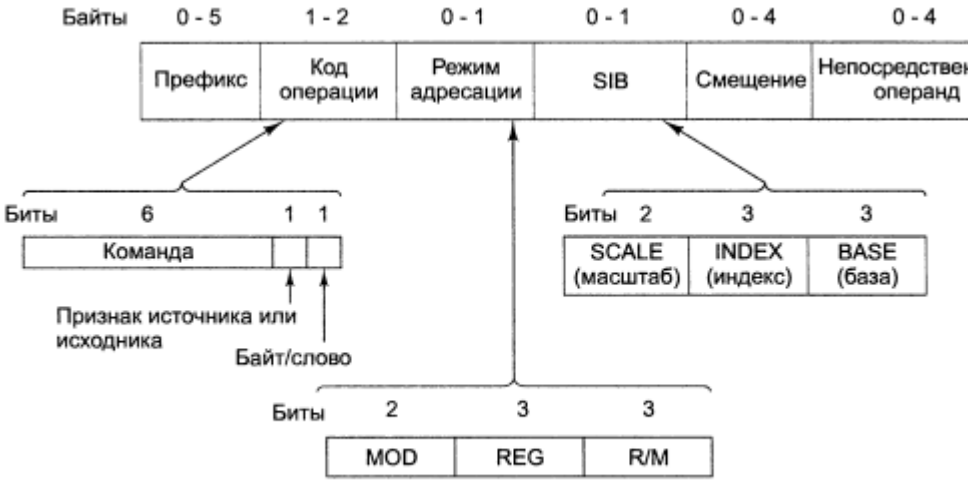
3.8.1. Формат команды

Машинная команда представляет собой закодированное по определенным правилам указание микропроцессору на выполнение некоторой операции или действия. Под каждую команду в системной памяти процессора прошиты подпрограммы их выполнения. Ассемблер, анализируя исходник проекта, в сегменте кода, заменяет предложения исходника на эти подпрограммы.

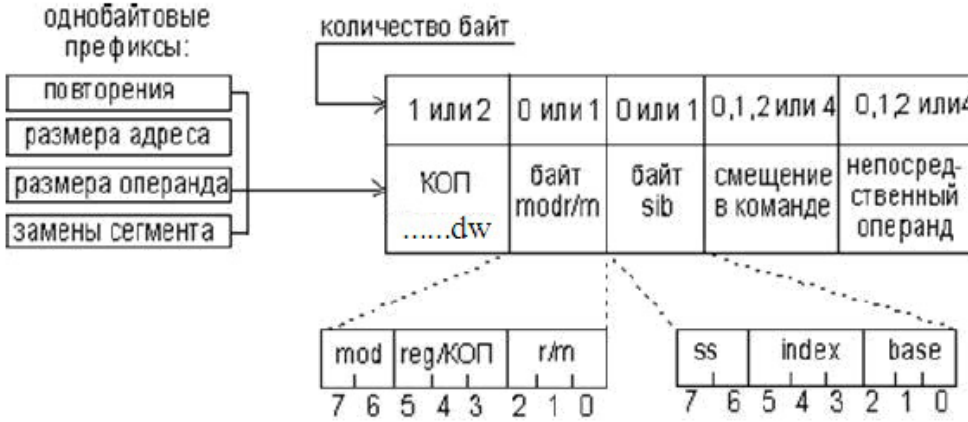
Каждая команда содержит элементы, определяющие:

- Код операции (КОП) описывает, что делать. Значение в поле КОП определяет в блоке микропрограммного управления подпрограмму, реализующую действия для данной команды.
- Операнды описывают объекты, с которыми нужно что-то делать. Операнды в команде могут и не задаваться, а подразумеваться по умолчанию.
- Типы операндов описывают, как делать, обычно задаются неявно.

Правила кодирования команд называются форматом команд. Команды процессоров архитектуры IA-32 считаются сложными. Максимальная длина машинной команды IA-32 составляет 15 байт. Реальная команда может содержать гораздо меньшее количество полей, вплоть до одного — только КОП. Приведенный на рисунке формат машинной команды является самым общим.



Форматы команд процессора Pentium 4



Структура машинной команды IA-32

Опишем назначения полей машинной команды.

Префиксы. Необязательные элементы машинной команды, каждый из которых состоит из одного байта или может отсутствовать. В памяти префиксы предшествуют команде. Назначение префиксов — модифицировать операцию, выпол-

няемую командой. Прикладная программа может использовать следующие типы префиксов:

- Префикс замены сегмента. В явной форме указывает, какой сегментный регистр используется в данной команде для адресации стека или данных. Префикс отменяет выбор сегментного регистра по умолчанию. Префиксы замены сегмента имеют следующие значения:
- 2eh –замена сегмента CS.
- 36h –замена сегмента SS.
- 3eh –замена сегмента DS.
- 26h –замена сегмента ES.
- 64h –замена сегмента FS.
- 65h –замена сегмента GS.
- Префикс разрядности адреса уточняет разрядность адреса (32 или 16-разрядный). Каждой команде, в которой используется адресный операнд, ставится в соответствие разрядность адреса этого операнда. Этот адрес может иметь разрядность 16 или 32 бит. Это смещение называется эффективный адрес. Если разрядность адреса 32 бит, это означает, что команда содержит 32-разрядное смещение, оно соответствует 32-разрядному смещению адресного операнда относительно начала сегмента и по его значению формируется 32-битное смещение в сегменте. С помощью префикса разрядности адреса можно изменить действующее по умолчанию значение разрядности адреса. Это изменение будет касаться только той команды, которой предшествует префикс.
- Префикс разрядности операнда аналогичен префиксу разрядности адреса, но указывает на разрядность операндов (32 или 16-разрядные), с которыми работает команда. В соответствии с какими правилами устанавливаются значения атрибутов разрядности адреса и операндов по умолчанию?
- Префикс повторения используется с цепочечными командами (командами обработки строк). Этот префикс “зацикливает” команду для обработки всех элементов цепочки. Система команд поддерживает два типа таких префиксов: безусловные (rep — 0f3h), заставляющие повторяться цепочечную команду некоторое количество раз; условные (repe/repz — 0f3h, repne/repnz — 0f2h), которые при зацикливании проверяют некоторые флаги, и в результате проверки возможен досрочный выход из цикла

Код операции. Обязательный элемент, описывающий операцию, выполняемую командой. Многим командам соответствует несколько кодов операций, каждый из которых определяет нюансы выполнения операции.

Поле кода операции не имеет однозначной структуры. В зависимости от конкретных команд оно может иметь в своем составе от 1 до 3 элементов. Один из этих трех элементов является непосредственно кодом операции или ее частью, остальные уточняют детали операции. Дополнительные биты поля КОП:

- Поле reg (3-бита) определяют регистр, используемый в команде.
- Бит d задает направление передачи данных: при 0 из регистра reg, при 1 в регистр reg,
- Бит s задает необходимость расширения 8-битового непосредственного операнда ло 16 или 32 бита
- Бит w определяет размер данных, которыми оперирует команда: байт, слово, двойное слово: при 0 — 8 битов; при 1 — 16 битов для 16-разрядного размера операндов или 32 бита для 32-разрядного размера операндов.

Однобайтовые КОП.

7	6	5	4	3	2	1	0
КОП							
КОП							w
КОП						d	w
КОП						s	w
КОП					reg		
КОП				w	reg		

Последующие поля машинной команды определяют местоположение операндов, участвующих в операции, и особенности их использования. Рассмотрение этих полей связано со способами задания операндов в машинной команде и потому будет выполнено позже.

Байт режима адресации modr/m. Иногда называется постбайтом, Значения этого байта определяет используемую форму адреса операндов. Операнды могут находиться в памяти в одном или двух регистрах. Если операнд находится в памяти, то байт **modr/m** определяет компоненты (смещение, базовый и индексный регистры), используемые для вычисления его эффективного адреса. Байт **modr/m** состоит из трех полей:

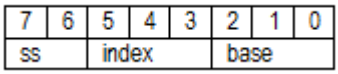
- **Поле mod** определяет количество байт, занимаемых в команде адресом операнда, поле смещение в команде).Поле mod используется совместно с полем r/m, которое указывает способ модификации адреса операнда смещение в команде.

К примеру, если mod = 00, это означает, что поле смещение в команде отсутствует, и адрес операнда определяется содержимым базового и (или) индексного регистра. Какие именно регистры будут использоваться для вычисления эффективного адреса, определяется значением этого байта. Если mod = 01, это означает, что поле смещение в команде присутствует, занимает один байт и модифицируется содержимым базового и (или) индексного регистра. Если mod = 10, это означает, что поле смещение в команде присутствует, занимает два или четыре байта (в зависимости от действующего по умолчанию или определяемого префиксом размера адреса) и модифицируется содержимым базового и (или) индексного регистра. Если mod = 11, это означает, что операндов в памяти нет: они находятся в регистрах. Это же значение байта mod используется в случае, когда в команде применяется непосредственный операнд.

- **Поле reg/КОП** определяет либо регистр, находящийся в команде на месте первого операнда, либо возможное расширение кода операции;
- **Поле r/m** используется совместно с полем mod и определяет либо регистр, находящийся в команде на месте первого операнда (если mod = 11), либо используемые для вычисления эффективного адреса (совместно с полем смещение в команде) базовые и индексные регистры.

Байт масштаб-индекс-база sib (Scale-Index-Base**)** используется для расширения возможностей адресации операндов. На наличие байта **sib** в машинной команде указывает сочетание одного из значений 01 или 10 поля mod и значения поля r/m= 100. Байт **sib** состоит из трех полей:

- **Поле масштаба ss**. В этом поле размещается масштабный множитель для индексного компонента index, занимающего следующие три бита байта sib. В поле ss может содержаться одно из следующих значений: 1, 2, 4, 8. При вычислении эффективного адреса на это значение будет умножаться содержимое индексного регистра.
- **Поле index** — используется для хранения номера индексного регистра, который применяется для вычисления эффективного адреса операнда;
- **Поле base** — используется для хранения номера базового регистра, который также применяется для вычисления эффективного адреса операнда. Напомню, что в качестве базового и индексного регистров могут использоваться практически все регистры общего назначения.



Поле смещения в команде. 8, 16 или 32-разрядное целое число со знаком, представляющее собой, полностью или частично (с учетом вышеприведенных рассуждений), значение эффективного адреса операнда.

Поле непосредственного операнда. Необязательное поле, представляющее собой 8, 16 или 32-разрядный непосредственный операнд. Наличие этого поля, конечно, отражается на значении байта modr/m.

Для описания команд приняты обозначения:

- Структура регистра флагов eflags:

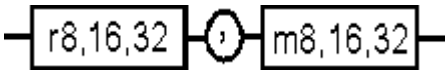
31	18	17	16	15	14	1312	11	10	09	08	07	06	05	04	03	02	01	00
0	0	VM	RF	0	NT	IOPL	OF	DF	IF	TF	SF	ZF	0	AF	0	PF	1	CF

В нижней строке этой таблицы приводятся значения флагов после выполнения команды:

- 1 — после выполнения команды флаг устанавливается (равен 1);
- 0 — после выполнения команды флаг сбрасывается (равен 0);
- r — значение флага зависит от результата работы команды;
- ? — после выполнения команды флаг не определен;
- пробел — после выполнения команды флаг не изменяется;

Для представления операндов в синтаксических диаграммах используются следующие обозначения:

- r8, r16, r32 — операнд в одном из регистров размером байт, слово или двойное слово;
- m8, m16, m32, m48 — операнд в памяти размером байт, слово, двойное слово или 48 бит;
- i8, i16, i32 — непосредственный операнд размером байт, слово или двойное слово;
- a8, a16, a32 — относительный адрес (смещение) в сегменте кода.
- На многих диаграммах в целях компактности возможные сочетания операндов показаны в виде следующей конструкции:



Конструируя команду на основе подобной синтаксической диаграммы, вы должны помнить о соответствии типов. В подобной диаграмме допустимы только следующие сочетания: "r8, m8", "r16, m16", "r32, m32". Например, сочетание "r8, m16" недопустимо. Однако есть единичные случаи, когда подобные сочетания возможны; тогда они специально оговариваются.

Описанная в данном приложении система команд в полном объеме поддерживается микропроцессором Pentium. Предыдущие модели микропроцессора могут не поддерживать отдельные команды. Чтобы прояснить этот момент, мы будем указывать в примерах для каждой команды директиву типа .286. Это будет означать, что описываемая команда поддерживается всеми моделями микропроцессора, начиная с i286. Если ничего не указывается, то это означает, что данная команда работает на всех моделях микропроцессоров Intel, начиная с i8086/8088.

3.8.2. Классификация команд

Имеются наборы команд для разных узлов процессора:

- Целочисленный процессор.
- Арифметический сопроцессор с плавающей точкой.
- Целочисленное MMX расширение.
- XMM расширение с плавающей точкой, 70 команд..

3.8.3. Целочисленный процессор

Команды общего назначения.

Команда	Описание
MOV приемник, источник	MOVe operand – пересылка операнда. Пересылка данных в приемник из источника. Адресаты - регистр, память или непосредственный операнд.
MOVS приемник, источник MOVSB MOVSW MOVSD	MOVe String Byte/Word/Double worf– пересылка строк, содержащих: • Байты • Слова • Двойные слова
MOVSX приемник, источник	MOVe and Sign eXtension – пересылка со знаковым расширением.

	Преобразование элемента со знаком меньшей размерности в эквивалентные элементы со знаком большей размерности.
MOVZX приемник, источник	MOVe and Zero eXtension – пересылка с нулевым знаковым расширением. Преобразование элемента без знака меньшей размерности в эквивалентные элементы без знака большей размерности.
XCHG операнд_1, операнд_2	eXCHanGe – обмен. Обмен данными между операндами. Это регистры или память. Команда "память - память" в микропроцессоре Intel не предусмотрена.
BSWAP reg32	Byte SWAP - перестановка байтов. Перестановка байт из порядка "младший - старший" в порядок "старший - младший". Разряды 7-0 обмениваются с разрядами 31-24, а разряды 15-8 с разрядами 23-16.
XLAT адрес_таблицы_байтов	transLATe byte from table – загрузка из таблицы. Загрузить в регистр AL байт из таблицы в сегменте данных, на начало которой указывает EBX (BX), при этом начальное значение AL играет роль смещения.
LEA приемник, источник	Load Effective Address - загрузка эффективного адреса. Получить эффективный адрес (смещение) источника.
LDS приемник, источник	Load pointer into DS - загрузить указатель сегмента в регистр DS из памяти. Загрузить пару DS:reg из памяти (m). Вначале идет слово (или двойное слово) в регистр (reg), а в DS - последующее слово.
LES приемник, источник	Аналогично предыдущему, но для пары ES:reg.
LFS приемник, источник	Аналогично предыдущему, но для пары FS:reg.
LGS приемник, источник	Аналогично предыдущему, но для пары GS:reg.

LSS приемник, источник	Аналогично предыдущему, но для пары SS:reg.
SETcc операнд	byte SET on conditon – установка байта по условию. Проверяет условие, заданное модификатором "cc". Если оно выполняется, то первый бит байта устанавливается в 1, в противном случае в 0. Модификаторы – по флагам.

Команды ввода-вывода.

Команда	Описание
IN аккумулятор, номер порта	INput operand from port – ввести операнд из порта Ввод в аккумулятор из порта ввода-вывода. Порт адресуется непосредственно или через регистр DX.
OUT номер порта, аккумулятор	OUT operand to port – вывести операнд в порта Вывод из аккумулятор в порта ввода-вывода. Порт адресуется непосредственно или через регистр DX.
INSB INSW INSD	INput String Byte/Word/Double word operands – ввод из порта элементов: Байта Слова Двойного слова. Вводит данные из порта, адрес которого находится в регистре DX, в ячейку памяти, определяемой регистрами ES:[EDI/DI].
OUTSB OUTSW [OUTSD	OUT String Byte/Word/Double word operands – вывод в порт элементов: Байта Слова Двойного слова. Выводит данные из ячейки памяти, определяемой регистрами DS:[ESI/SI], в выходной порт, адрес которого находится в регистре DX.

Инструкции работы со стеком.

Команда	Описание
PUSH	PUSH operand onto stack – поместить в стек.

источник	Поместить в стек слово или двойное слово. Поскольку при включении в стек слова нарушается выравнивание стека по границам двойных слов, рекомендуется в любом случае помещать в стек двойное слово.
PUSH const	Поместить в стек непосредственный 32-битный операнд.
PUSHA	PUSH All general registers onto stack – поместить в стек все регистры общего назначения. Поместить в стек регистры EAX, EBX, ECX, EDX, ESI, EDI, EBP, ESP.
POP пприемник	POP operand from the stack – извлечь операнд из стека. Извлечь из стека слово или двойное слово.
POPA	POP All general registers onto stack – извлечь изстека все регистры общего назначения. Извлечье из стека данные в регистры EAX, EBX, ECX, EDX, ESI, EDI, EBP, ESP.
PUSHF	PUSH Flags register onto stack Помещение в стек регистра флагов.
POPF	POP Flags register from the stack Извлечь из стека данные в регистр флагов.

Целочисленная арифметика

Команда	Описание
ADD приемник, источник	ADDition – сложение. Сложение двух операндов. Один из операндов – приемник. Первый операнд может быть регистром или ячейкой памяти, второй - регистром, ячейкой памяти, константой. Невозможно только, когда оба операнда являются ячейками памяти.
XADD приемник, источник	eXchange and ADD –обмен двух значений и суммирование. Данная операция производит в начале обмен операндами, а затем выполняет операцию ADD.
ADC приемник, источник	ADdition with Carry. Сложение с учетом флага переноса - в младший бит

	добавляется бит (флаг) переноса.
INC операнд	INCrement operand by 1. Увеличить операнд на 1.
SUB операнд1, операнд2	SUBtract. Вычитание двух операндов. Остальное аналогично сложению (команда ADD).
SBB операнд1, операнд2	SuBtract with Borrow. Вычитание с учетом предыдущего вычитания (бита заема). Из младшего бита вычитается бит (флаг) переноса.
DEC r/m	DECrement operand by 1. Уменьшить операнд на 1.
CMP операнд1, операнд2	CoMPare operands – сравнение операндов. Вычитание без изменения операндов (сравнение).
CMPXCHG приемник, источник	CoMPare and eXCHanGe – сравнение и обмен.. Сравнение с обменом. Источник – всегда аккумулятор. Если значения в операнде-получателе и аккумуляторе равны, операнд-получатель заменяется операндом-источником, исходное значение операнда-получателя загружается в аккумулятор.
CMPXCHG8B приемник, источник	CoMPare and eXCHanGe 8 Byte – сравнение и обмен.. Сравнение с обменом 8 байтов. Источник – всегда аккумулятор. Если значения в операнде-получателе и аккумуляторе равны, операнд-получатель заменяется операндом-источником, исходное значение операнда-получателя загружается в аккумулятор.
NEG источник	NEGate operand. Изменение знака операнда.
AAA	Ascii Adjust after Addition - ASCII-коррекция после сложения. Алгоритм: • проверить значение младшего полубайта регистра AL и значение флага AF; • если (значение младшего полубайта регистра AL>9) или (AF=1), то выполнить следующие действия: увеличить значение al на 6, очистить

	старший полубайт регистра al, увеличить значение ah на 1, установить флаги: AF=1, CF=1; • иначе сбросить флаги AF= 0 и CF= 0.
AAS	Ascii Adjust after Substraction - ASCII-коррекция после вычитания. Алгоритм работы: • если (младший полубайт регистра AL меньше 9) или (флаг AF=1), то выполнить следующие действия: уменьшить значение младшего полубайта регистра al на 6, обнулить значение старшего полубайта регистра AL; • установить флаги AF=1 и CF=1;.
AAM	Ascii Adjust after Multiply - ASCII-коррекция после умножения. Алгоритм работы: • разделить значение регистра AL на 10; • записать частное в регистр AH, остаток — в регистр AL.
AAD	Ascii Adjust before Division - ASCII-коррекция перед делением. Алгоритм работы: • умножить значение регистра AH на 10 и сложить полученное значение с содержимым регистра AL; • присвоить регистру AL это значениеl; • обнулить регистр AH.
DAA	Decimal Adjust for Addition - десятичная коррекция после сложения. Алгоритм работы. команда работает только с регистром AL и анализирует наличие следующих ситуаций: • Ситуация 1. В результате предыдущей команды сложения флаг AF=1 или значение младшей тетрады регистра AL>9. Напомним, что флаг AF устанавливается в 1 в случае переноса двоичной единицы из бита 3 младшей тетрады в старшую тетраду регистра AL (если значение превысило 0fh). Наличие одного из этих двух признаков говорит о том, что значение младшей тетрады превысило 9h.

	• Ситуация 2. В результате предыдущей команды сложения флаг CF=1 или значение регистра AL>9fh. Напомним, что флаг CF устанавливается в 1 в случае переноса двоичной единицы в старший бит операнда (если значение превысило 0ffh в случае регистра AL). Наличие одного из этих двух признаков говорит о том, что значение в регистре AL превысило 9fh. Если имеет место одна из этих двух ситуаций, то регистр al корректируется следующим образом: • для ситуации 1 содержимое регистра AL увеличивается на 6; • для ситуации 2 содержимое регистра AL увеличивается на 60h; • если имеют место обе ситуации, то корректировка начинается с младшей тетрады.
DAS	Decimal Adjust for Subtraction - десятичная коррекция после вычитания. Команда работает только с регистром AL и анализирует наличие следующих ситуаций: • Ситуация 1. В результате предыдущей команды сложения флаг af =1 или значение младшей тетрады регистра al>9. Напомним, что для случая вычитания флаг af устанавливается в 1 в случае заема двоичной единицы из старшей тетрады в младшую тетраду регистра al. Наличие одного из этих двух признаков говорит о том, что значение младшей тетрады превысило 9h и его нужно корректировать. • Ситуация 2. В результате предыдущей команды сложения флаг cf =1 или значение регистра al>9fh. Напомним, что для случая вычитания флаг cf устанавливается в 1 в случае заема двоичной единицы. Наличие одного из этих двух признаков говорит о том, что значение в регистре al превысило 9fh. Если имеет место одна из этих ситуаций, то регистр al корректируется следующим образом: • для ситуации 1 содержимое регистра al умень-

	шается на 6; • для ситуации 2 содержимое регистра al уменьшается на 60h; • если имеют место обе ситуации, то корректировка начинается с младшей тетрады.
MUL множитель	MULtiplay - умножение целых чисел без знака. Команда выполняет умножение двух операндов без учета знаков. Алгоритм зависит от формата операнда команды и требует явного указания местоположения только одного сомножителя, который может быть расположен в памяти или в регистре. Местоположение второго сомножителя фиксировано и зависит от размера первого сомножителя: • если операнд, указанный в команде — байт, то второй сомножитель должен располагаться в al; • если операнд, указанный в команде — слово, то второй сомножитель должен располагаться в ax; • если операнд, указанный в команде — двойное слово, то второй сомножитель должен располагаться в eax. Результат умножения помещается также в фиксированное место, определяемое размером сомножителей: • при умножении байтов результат помещается в ax; • при умножении слов результат помещается в пару dx:ax; • при умножении двойных слов результат помещается в пару edx:eax.
IMUL множ IMUL множ1, множ2 IMUL резул, множ1, множ2	Integer MULtiplay - умножение целых чисел со знаком. Алгоритм работы команды зависит от используемой формы команды. Форма команды с одним операндом требует явного указания местоположения только одного сомножителя, который может быть расположен в

	ячейке памяти или регистре. Местоположение второго сомножителя фиксировано и зависит от размера первого сомножителя: • если операнд, указанный в команде, — байт, то второй сомножитель располагается в al; • если операнд, указанный в команде, — слово, то второй сомножитель располагается в ax; • если операнд, указанный в команде, — двойное слово, то второй сомножитель располагается в eax. Результат умножения для команды с одним операндом также помещается в строго определенное место, определяемое размером сомножителей: • при умножении байтов результат помещается в ax; • при умножении слов результат помещается в пару dx:ax; • при умножении двойных слов результат помещается в пару edx:eax. Команды с двумя и тремя операндами однозначно определяют расположение результата и сомножителей следующим образом: • в команде с двумя операндами первый операнд определяет местоположение первого сомножителя. На его место впоследствии будет записан результат. Второй операнд определяет местоположение второго сомножителя; • в команде с тремя операндами первый операнд определяет местоположение результата, второй операнд — местоположение первого сомножителя, третий операнд может быть непосредственно заданным значением размером в байт, слово или двойное слово.
DIV делитель	DIVide unsigned - деление целых чисел без знака.

	Для команды необходимо задание двух операндов — делимого и делителя. Делимое задается неявно и размер его зависит от размера делителя, который указывается в команде. Алгоритм работы: • если делитель размером в байт, то делимое должно быть расположено в регистре ax. После операции частное помещается в al, а остаток — в ah; • если делитель размером в слово, то делимое должно быть расположено в паре регистров dx:ax, причем младшая часть делимого находится в ax. После операции частное помещается в ax, а остаток — в dx; • если делитель размером в двойное слово, то делимое должно быть расположено в паре регистров edx:eax, причем младшая часть делимого находится в eax. После операции частное помещается в eax, а остаток — в edx.
IDIV делитель	Integer DIVide - деление целых чисел со знаком. Для команды необходимо задание двух операндов — делимого и делителя. Делимое задается неявно, и размер его зависит от размера делителя, местонахождение которого указывается в команде. Алгоритм работы: • если делитель размером в байт, то делимое должно быть расположено в регистре ax. После операции частное помещается в al, а остаток — в ah; • если делитель размером в слово, то делимое должно быть расположено в паре регистров dx:ax, причем младшая часть делимого находится в ax. После операции частное помещается в ax, а остаток — в dx; • если делитель размером в двойное слово, то делимое должно быть расположено в паре реги-

	стров edx:eax, причем младшая часть делимого находится в eax. После операции частное помещается в eax, а остаток — в edx;
CBW	Convert Byte to Word - преобразование байта в слово Команда использует только регистры al и ax: Алгоритм работы - анализ знакового бита регистра al: • если знаковый бит al=0, то ah=00h; • если знаковый бит al=1, то ah=0ffh.
CWD	Convert Word to Double word - преобразование слова в двойное слово. Команда использует только регистры al и ax. Алгоритм работы -анализ знакового бита регистра al: • если знаковый бит al=0, то ah=00h; • если знаковый бит al=1, то ah=0ffh.
CWDE	Convert Word to Double word Extended - преобразование слова в двойное слово с расширением. Ккоманда использует только регистры ax и eax. Алгоритм работы - анализ знакового бита регистра ax: • если знаковый бит ax=0, то установить старшее слово eax=0000h; • если знаковый бит ax=1, то установить старшее слово eax=0ffffh.
CDQ	Convert Double word to Quad word Преобразование двойного слова (EAX) в учетверенное слово (EDX:EAX). Алгоритм работы: копирование значения старшего бита регистра eax на все биты регистра edx.

Логические операции

Команда	Описание
AND приемник, источник	logical AND – логическое И. Побитовая логическая "И". В приемнике бит устанавливается в 1, если отличны от нуля одноименные биты и в источнике, и в приемнике..

TEST приемник, источник	TEST операнд. Аналогична "AND", но не меняет биты приемника. Используется для проверки ненулевых бит.
OR приемник, источник	logical OR – логическое включающее ИЛИ. Побитовая логическая "ИЛИ". В приемнике бит устанавливается в 1, если отличны от нуля одноименные биты или в источнике, или в приемнике..
XOR приемник, источник	logical eXclusive OR – логическое исключающее ИЛИ. Побитовая логическая "иселючающее ИЛИ". В приемнике бит устанавливается в 1, если одноименные биты в источнике и приемнике различны..
NOT источник	NOT operand – логическое НЕ. Переключение всех бит (инверсия)..

Сдвиговые операции

Команда	Описание
RCL/RCR dest,src	Циклический сдвиг влево/вправо через бит переноса CF. Src может быть либо CL, либо непосредственный операнд.
ROL/ROR dest,src	Аналогично командам RCL/RCR, но по другому, работает с флагом CF. Флаг не участвует в цикле, но в него попадает бит, перешедший с начала на конец или наоборот.
SAL/SAR dest,src	Сдвиг влево/право. Называется еще арифметическим сдвигом. При сдвиге вправо дублируется старший бит. При сдвиге влево младший бит заполняется нулем. Ушедший бит помещается в CF.
SHL/SHR dest,src	Логический сдвиг влево/вправо. Сдвиг вправо отличается от SAR тем, что и старший бит заполняется нулем.
SHLD/SHRD dest,src,count	Трехоперандные команды сдвига влево/вправо. Первым операндом, как обычно, может быть либо регистр, либо ячейка памяти, вторым операндом должен быть регистр общего назначения, третьим - регистр CL или непосредственный операнд. Суть операции заключается в том, что dest и src в начале объединяются, а потом производится сдвиг на количество бит count. Результат снова помещается в dest.

Цепочечные операции.

Команда	Описание
REP REPE и REPZ REPNE и REPNZ	REPeat string operation – повторить цепочечную операциюю Префикс, означающий повтор следующей за ним операции до обнуления ECX. Префикс имеет разновидности: REPZ (REPE) - выполнять, пока не нуль (ZF=1), REPNZ (REPNE) – выпо лнять, пока нуль.
MOVS приемник, источник MOVSB MOVSW MOVSD	MOVe String Byte/Word/Double word operands – пересылка цепочек. Приемник, источник можно явно не указывать. Команда передает из цепочки, адресуемой DS:[ESI], в цепочку приемника, адресуемую ES:[EDI]. Разновидности для разных элементов цепочки: • байт, • слово или • двойное слово.
LODS источник MOVSB MOVSW MOVSD	LOaD String Byte/Word/Double word operands – загрузка цепочек. Загружает из ячейки памяти, адресуемой DS:ESI/si, в регистр AL/AX/EAX цепочку, и изменяет содержимое SI на величину, равную длине цепочки. Разновидности для разных элементов цепочки: • байт, • слово или • двойное слово.
STOS приемник STOSB STOSW STOSD	STOre String Byte/Word/Double word operands – сохранение цепочек. Сохраняет в ячейке памяти, адресуемой DS:ESI/SI, из регистра AL/AX/EAX цепочку, и изменяет содержимое SI на величину, равную длине цепочки. Разновидности для разных элементов цепочки: • байт, • слово или • двойное слово.
SCAS приемник SCASB SCASW SCASD	SCAn String Byte/Word/Double word – сканированение цепочек. Команда вычитает элемент цепочки приемника из содержимого аккумулятора (AL\AX\EAX) и модифицирует флаги. Разновидности для разных элементов цепочки: • байт, • слово или

	• двойное слово.
CMPS приемник, источник CMPSB CMPSW CMPSD	CoMPare String Byte/Word/Double word operands - сравнение цепочек. Команда вычитанет элемент цепочки приемника из соответствующего элемента цепочки источника и модифицирует флаги. Регистры EDI и ESI автоматически продвигаются на следующий элемент. Разновидности для разных элементов цепочки: • байт, • слово или • двойное слово.

Управление флагами.

Команда	Описание
CLC	CLear Carry flag Сброс флага переноса.
STC	SeT Carry flag Установка флага переноса.
CMC	CoMplement Carry flag Инверсия флага переноса.
CLD	CLear Direction flag Сброс флага направления – для цепочечных команд процессор будет выполнять инкремент регистров SI DI.
STD	SeT Direction flag Установка флага направления - для цепочечных команд процессор будет выполнять декремент регистров SI DI.
CLI	CLear Interrupt flag Сброс флага прерываний - запрет маскируемых аппаратных прерываний.
STI	SeT Interrupt flag Установка флага прерываний - разрешение маскируемых аппаратных прерываний.

Команды передачи управления.

Команда	Описание
JMP метка	JuMP –безусловный переход. Имеется несколько форм, различающихся расстоянием метки перехода от текущего адреса, и способом задания целевого адреса. При работе в Windows используется в основном внутрисегментный переход (NEAR) в пределах 32-битного сегмента. Адрес перехода может задаваться непосредственно (в программе это метка) или косвенно, т.е. содержаться в ячейке памяти или регистре (JMP [EAX]). Другой тип перехода - короткий переход (SHORT), занимает всего 2 байта. Диапазон смещения, в пределах которого происходит переход: -128 ... 127. Использование такого перехода весьма ограниченно.
Jcc метка	Jump if condition - условный переход. Команда осуществляет переход при выполнении условия, заданного в . поле условия. Возможны условия, кодируемые 4-ех битным кодом: • JA/JNBE - перейти, если выше. • JAE/JNB - перейти, если выше или равно. • JB/JNAE - перейти, если ниже • JBE/JNA - перейти, если ниже. • JC - перейти, если перенос • JE/JZ - перейти, если нуль • JG/JNLE - перейти, если больше. • JGE/JNL - перейти, если больше или равно • JL/JNGE - перейти, если меньше. • JLE/JNG - перейти, если меньше или равно • JNC - перейтИ, если нет переноса. • JNE/JNZ - перейти, если меньше или равно • JNO - перейти, если нет переполнения • JNP/JPO - перейти, если нет паритета • JNS - перейти, если нет знака • JO - перейти, если есть переполнения • JP/JPE - перейти, если есть паритет • JS - перейти, если есть знак • JCXZ - переход, если CX=0 • JECXZ - переход, если ECX=0

	В плоской модели команды условного перехода осуществляют переход в пределах 32-битного регистра.
LOOP метка	LOOP control by register CX - команды управления циклом по регистру CX. Команды этой группы используют счетчик цикла в регистре CX. В них осуществляется декремент CX и проверка его содержимого. Тело цикла повторяется пока содержимое CX не равно нулю. Если содержимое CX равно нулю, то управление передается команде, метка которой опреджелена в команде LOOP.
LOOPE метка LOOPZ метка	LOOP control by register CX not Equal 0 and ZF=1 - команды управления циклом по регистру CX с учетом ZF=1. Команды этой группы основаны на командах LOOP. Дополнительно в них в теле цикла анализируется флаг ZF. Цикл принудительно завершается, если ZF= 1.
LOOPNE метка LOOPNZ метка	LOOP control by register CX not Equal 0 and ZF=0 - команды управления циклом по регистру CX с учетом ZF=0. Команды этой группы основаны на командах LOOP. Дополнительно в них в теле цикла анализируется флаг ZF. Цикл принудительно завершается, если ZF= 0.
CALL цель	CALL – вызов. Передает управление процедуре (метке) с сохранением в стеке адреса, следующей за CALL командой. В плоской модели адрес возврата представляет собой 32-битное смещение. Межсегментный вызов предполагает сохранение в стеке селектора и смещения, т.е. 48-битной величины (16 бит - селектор и 32 бита - смещение).
RET [N]	RETurn from procedure – возврат из процедуры Необязательный параметр N предполагает, что команда также автоматически чистит стек (освобождает N байт). Команда имеет разновидности, которые выбираются ассемблером автоматически, в зависимости от того, является процедура ближней или дальней.

Команды поддержки языков высокого уровня.

Команда	Описание

ENTER Размер,Вложенность	Подготовка стека для локальных параметров процедуры. Алгоритм: • (EBP) => стек. • (ESP) => промежуточная переменная fp. Если Вложенность не 0, то коррекция EBP по режиму адресации и (EBP) => стек. • (fp) => EBP. • (fp) => стек. • ESP = (ESP) - Размер
LEAVE	LEAVE from procedure – выход из процедкры. Выполгяет действия, обратные комаде ENTER, приводя стек в исходное состояние: • (EBP) => ESP – восстановление состояния стека до процедуры. • EBP восстанавливается из стека.
BOUND РегистрИндекса, ГраницыМассива	BOUND check array BOUNDs – контроль нахождения индекса в гоаницах массива. РегистрИндекса содержит текущий индекс массива, а второй операнда определяет в памяти 2 слова или 2 двойных слова. Первое считается минимальным значением индекса, а второе - максимальным. Если текущий индекс оказывается вне границ, то генерируется команда INT 5. Используется для контроля нахождения индекса в заданных рамках, что является важным средством отладки.

Команды прерываний.

Команда	Описание
INT НомерПрерывания	INTerrupt – прерывание. Вызов прерывания с заданным номером. Алгоритм: • В стек содержимое регистра флагов. • В стек полный адрес возврата. Сбрасывается флаг TF. • Косвенный переход через элемент дескрипторной таблицы прерываний по НомерПрерывания.

INTO	INTerrupt if Overflow – прерывание, если переполнение. Вызов прерывания с заданным номером. Алгоритм: • Если флаг переполнения OF = 0, ничего не делается. • Если флаг переполнения OF = 1, то вызов команды INT.
IRET	Interrupt RETurn – возврат из прерывания. Алгоритм: • Если флаг NT = 0, то возврат в прерванную программу. • Если флаг NT = 1, то переключение задач. Команда извлекает из стека сохраненные в нем адрес возврата и регистр флажков. Бит уровня привилегий будет модифицироваться только в том случае, если текущий уровень привилегий равен 0.

Команды синхронизации процессора.

Команда	Описание
HLT	HaLT – останов. Останавливает процессор. Из него процессор может быть выведен внешним прерыванием или перезагрузкоц.
LOCK	LOCK signal prefix – блокировка. Представляет собой префикс блокировки шины. Он заставляет процессор сформировать сигнал LOCK# на время выполнения находящейся за префиксом команды. Этот сигнал блокирует запросы шины другими процессорами в мультипроцессорной системе.
NOP	No Operanion – нет операции. Холостая команда. Не производит никаких действий.
WAIT	WAIT – ожидание. Синхронизация с сопроцессором, останавливает основной процессор до завершение операции в сопроцессоре.. Большинство команд сопроцессора автоматически вырабатывают команду.FWAIT, выполняющую то же самое.

Команды обработки цепочки бит.

Команда	Описание
BSFприемник, источник	Bit Scan Forward – побитовое сканирование вперед. Проверка наличия битов 1 в источнике Номер первого бита, находящегося в состоянии 1, помещается в приемник, флажок ZF сбрасывается в 0. Если источник содержит 0, то флаг ZF=1, а содержимое приемника не определено.
BSR приемник, источник	Bit Scan Revers – побитовое сканирование назад. Аналогична BSF. Разница в направлении просмотра..
BT источник, индекс	Bit Test – проверка битов. Извлечение бита (с номером индекс) из источника, помещение в флаг CF..
BTC источник, индекс	Bit Test and Complement – проверка битов и инверсия. Извлечение бита (с номером индекс) из источника, помещение в флаг CF, инверсия.
BTR источник, индекс	Bit Test and Reset – проверка битов и сброс в 0. Извлечение бита (с номером индекс) из источника, помещение в флаг CF, сброс в 0.
BTS источник, индекс	Bit Test and Set – проверка битов и установка в 1. Извлечение бита (с номером индекс) из источника, помещение в флаг CF, установка в 1.

Команды управления защитой.

Команда	Описание
LGDT источник_48	Load Global Descriptor Table Загрузка 48-битного регистра глобальной дескрипторной таблицы GDTR из памяти. Источник_48 указывает на 6-байтную величину. Это 16 бит размера и 32 бита базового адреса начала таблицы GDT.
SGDT приемник_48	Store Global Descriptor Table Сохранить регистр GDTR в памяти.
LIDT источник_16	Load Interrupt Descriptor Table Загрузка 16-битного регистра дескрипторной таблицы прерываний IDTR из памяти.

SIDT приемник_16	SIDT Сохранить регистр IDTR в памяти.
LLDT источник_16	Load Local Descriptor Table Загрузка 16-битного регистра локальной дескрипторной таблицы LLDR значением селектора глобальной дескрипторной таблицы GDT из памяти (16 бит).
SLDT приемник_16	Store Local Descriptor Table Сохранить регистр LDTR в регистре или памяти (16 бит).
LMSW источник_16	Load Machine Status Word Загрузка слова состояния машины MSW (младшие 16 бит регистра CRO) значением 16-битного слова памяти или регистра.
SMSW приемник_16	Store Machine Status Word Сохранить MSW в регистре или памяти (16 бит).
LTR источник_16	Load Test Register Загрузка регистра задачи TR селектором сегмента Tss задачи из регистра или памяти (16 бит).
STR приемник_16	Store Test Register Сохранить регистр задачи TR в регистре или памяти (16 бит).
LAR приемник, источник	Load Access Rights byte Загрузка в приемник байта прав доступа, дескриптор которого задан в источнике.
LSL приемник, источник	Losd Segment Limit Загрузка в приемник лимита сегмента, дескриптор которого задан в источнике.
ARPL приемник, источник	Adjust RPL field of selector – настройка поля RPL селектора. Алгоритм: • Если RPL_прием > RPL_источ, то флаг ZF=0. • Если RPL_прием < RPL_источ, то флаг ZF=1, RPL_прием = RPL_источ.
VERR сегмент	VERify for Reading – проверка сегмента на чтение. Алгоритм: • Проверка определения сегмента в таблицах CDT

	или LDT. • Проверка, указывет ли дескриптор сегмента на сегмент кода или данных. • Проверка, является ли сегмент записываемым. • Проверка уровня привилегий.. • Если проверки положительны, то флаг ZF=1, иначе ZF=0.
VERW сегмент	VERify for Writing – проверка сегмента на чтение. Алгоритм: • Проверка определения сегмента в таблицах CDT или LDT. • Проверка, указывет ли дескриптор сегмента на сегмент кода или данных. • Проверка, является ли сегмент считываемым. • Если проверки положительны, то флаг ZF=1, иначе ZF=0.

Команды обмена с управляющими регистрами.

Команда	Описание
MOV CRn,источник	MOV operand to system register Загрузка управляющего регистра CRn.
MOV приемник,CRn	MOV operand from system register Чтение управляющего регистра CRn.
MOV DRn, источник	MOV operand to system register Загрузка регистра отладки DRn.
MOV приемник,DRn	MOV operand from system register Чтение регистра отладки DRn.
MOV TRn, источник	MOV operand to system register Загрузка регистра тестирования TRn.
MOV приемник,TRn	MOV operand from system register Чтение регистра тестирования TRn.
RDTSC	ReaD from Time Stamp Counter Чтение счетчика тактов.

Команды идентификации и управления архитектурой.

Команда	Описание
CPUID	CPU IDentification Получение информации о текущем процессоре. Требует параметр в регистре EAX. Если EAX=0, то процессор в регистрах EBX,EDX,ECX возвращает символьную строку, специфичную для производителя. Процессоры AMD возвращают строку "AuthenticAMD", процессоры Intel - "GenuineIntel". Если EAX=1, то в младшем слове регистра EAX возвращает код идентификации. Если EAX=2, то в регистрах EAX, EBX, ECX, EDX возвращаются параметры конфигурации процессора.
RDMSR	ReaD from Model Specific Register Чтение из 64 разрядного модельно-специфического регистра MSR. Номер MSR должен находиться в регистре ECX. Алгоритм: • Проверить на нулевой привилегий. • Проверка правильности номера в регистре ECX. • Если все правильно, то поместить значение адресуемого MSR в регистровую пару EDX:EAX.
WRMSR	Write to Model Specific Register Запись в 64 разрядный модельно-специфического регистра MSR. Номер MSR должен находиться в регистре ECX. Алгоритм: • Проверить на нулевой привилегий. • Проверка правильности номера в регистре ECX. • Если все правильно, то поместить в адресуемый MSR значние из регистровой пары EDX:EAX.

Управление кэшированием.

Необходимость управления кэшированием вызвана тем, что большинство мультимедийных приложений оперируют большими объемами данных, при этом может случиться, что данные, загруженные в кэш, никогда не понадобятся. Чтобы оптимизировать работу кэша, в систему команд SSE-расширения и были включены команды управления кэшем.

Мнемоника	Описание

MOVNTI	MOVe using Non-Temporal of Int32 Сохранение двойного слова из 32-разрядного регистра общего назначения в памяти без использования кэша.

Команды управления кэшированием.

Команда	Описание
INVD	INValiDate cache – недостовеность кэш-памяти. Алгоритм: • Очистка кэш-памяти первого уровня. • Генерация сигнала на. очистку кэш-памяти второго уровня.
WBINVD	Write Back and INValiDate cache - обратная запись и недостовеность кэш-памяти. Алгоритм: • Очистка кэш-памяти первого уровня. • Записать содежимое кэш-памяти второго уровня в основую память. • Очистить кэш-памяти второго уровня.
INVLPG адрес	INValiDate PaGe – недостовеность элемента буфера ассоциативной трансляции таблиц каталогов и страниц памяти TLB (Translation Lookaside Buffer). Алгоритм: • Просмотреть элементы буфера TLB на соответствие его элементов адресу в команде. • Если соответстчие выявлено, то пометить элемент как недостоверный.

3.8.4. Сопроцессор с плавающей точкой

Здесь мы коснемся основных положений работы арифметического сопроцессора.

Арифметический сопроцессор работает со своим набором команд и своим набором регистров. Однако выборку команд сопроцессора осуществляет процессор.

Арифметический сопроцессор выполняет операции со следующими типами данных:

- целое слово (16 бит),
- короткое целое (32 бита),

- длинное слово (64 бита),
- упакованное десятичное число (80 бит),
- короткое вещественное число (32 бита),
- длинное вещественное число (64 бита),
- расширенное вещественное число (80 бит).

При выполнении операции сопроцессором, процессор ждет завершения этой операции. Другими словами, перед каждой командой сопроцессора ассемблером автоматически генерируется команда, проверяющая, занят сопроцессор или нет. Если сопроцессор занят, процессор переводится в состояние ожидания. Иногда программисту требуется вручную ставить команду ожидания (WAIT) после команды сопроцессора.

Сопроцессор имеет восемь 80-битных рабочих регистров, представляющих собой стековую кольцевую структуру. Регистры называются R0,R1, ... R7, но доступ к ним напрямую невозможен. Каждый регистр может занимать любое положение в стеке. Название стековых (относительных) регистров - ST(0), ST(1), ST(2), ST(3), ST(4), ST(5), ST(6), ST(7).

Регистр состояния. По его флагам можносудить о результате выполненной операции. Биты регистра состояния:

- 0-й бит, флаг недопустимой операции.
- 1-й бит, флаг денормализованной операции.
- 2-й бит, флаг деления на ноль.
- 3-й бит, флаг переполнения.
- 4-й бит, флаг антипереполнения.
- 5-й бит, флаг неточного результата.
- 6-й бит, ошибка стека.
- 7-й бит, общий флаг ошибки.
- 8,9,10-й, флаги условий.
- 11,12,13-й, число, показывающее, какой регистр является вершиной.
- 14-й бит, условный флаг.
- 15-й бит, флаг занятости.

Регистр управления содержит в себе биты, влияющие на выполнение команд сопроцессора. Биты регистра управления:

- 0-й бит, маска недействительной операции.
- 1-й бит, маска денормализованного операнда.
- 2-й бит, маска деления на ноль.

- 3-й бит, маска переполнения.
- 4-й бит, маска антипереполнения.
- 5-й бит, маска неточного результата.
- 6,7-й бит, резерв.
- 8-9-й биты, управление точностью.
- 10,11-й биты, управление округлением.
- 12-й, управление бесконечностью.
- 13,14,15-й, резерв.

Регистр тэгов содержит 16 бит, описывающих содержание регистров сопроцессора: по два бита на каждый рабочий регистр. Тэг говорит о содержимом регистре данных. Вот значение тэгов: 00 - действительное ненулевое число, 01 - истинный нуль, 10 - специальные числа, 11 - отсутствие данных.

При вычислении с помощью команд сопроцессора большую роль играют исключения или особые ситуации. Типичной особой ситуацией является деление на 0. Биты особых ситуаций хранятся в регистре состояний. Учет особых ситуаций необходим для получения правильных результатов.

Список особых ситуаций.

- Неточный результат (округление).
- Недействительная операция.
- Деление на ноль.
- Антипереполнение (слишком маленький результат).
- Переполнение (слишком большой результат).
- Денормализованный операнд.

После обзора команды будут рассмотрены подробно.

Категория	Подкатегория	Команды
Управление сопроцессором	Инициализация	FINIT
	Работа со средой	FSAVE, FSTENV, FLDENV, FCLEX, FLDCW, FSTCW, FSTSW, FRSTOR
	Работа со стеком	FINCSTP, FDECSTP, FFREE,
	Вспомогательные	FNOP, FWAIT
Передача данных	Вещественные числа	FLD, FST, FSTP
	Целые числа	FILD, FIST
	Десятичные числа	FBLD, FBST, FBSTP

	Загрузка констант	0, 1, число PI, Log2(10), Log2(e), Log10(2), LN(2)
	Обмен	FXCH
Сравнение данных	Вещественные числа	FCOM, FCOMP, FCOMPP, FUCOMP, FUCOMPP
	Целые числа	FICOM, FICOMP
	Анализ	FTST, FXAM
Арифметические	Вещественные числа	FADD, FADDP, FSUB, FSUBP, FSUBR, FSUBRP, FMUL, FMULP, FDIV, FDIVP, FDIVR, FDIVRP
	Целые числа	FIADD, FISUB, FISUBR, FIMUL, FIDIV, FIDIVR
	Вспомогптельные	FABS, FCHS, FSQRT, FSCALE, FPREM, FPREM1, FRNDINT
Трансцендентные	Тригонометрия	Синус, Косинус, Синус + Косинус, Тангенс, Арктангенс
	Логарифмы и степени	2 в степени X минус 1, Y*Log2(X), Y*Log2(X+1)

Инициализация. Перед использованием любых команд математического сопроцессора необходимо выполнить его инициализацию командой FINIT (Floating-point INITialisation a device FPU). Алгоритм работы команды выглядит следующим образом:

- В регистр управления (cwr) помещается число 37h, означающее, что округление будет выполняться до ближайшего целого (поле rc = 0).
- Биты с нулевого по пятый устанавливаются в 1, что означает маскирование всех исключений.
- Поле управления точностью устанавливается для работы с максимальной точностью в 64 бит (pc =11).
- Регистр состояния (swr) обнуляется, а это означает, что исключения отсутствуют и физический регистр R0 регистрового стека становится вершиной стека st(O).
- В регистр тегов (twr) заносятся единицы, а это означает, что все регистры сопроцессора считаются пустыми.
- Регистры указателей данных (dpr) и команд (ipr) обнуляются.

Команда	Описание
FINIT	Floating-point INITializing a device FPU

	Инициализация сопроцессора.

Работа со средой.

Команда	Описание
FSAVE приемник	Floating-point SAVE fpu state. Сохранение состояния оборудования и файла регистров в памяти.
FSTENV источник	Floating-point STore ENVironment. Сохранение состояния оборудования (источника - SR, CR, TAGW, FIP, FDP) в памяти.
FLDENV источник	Floating-point LoaD ENVironment. Загрузка состояния оборудования (источника - SR, CR, TAGW, FIP, FDP) из памяти.
FRSTOR src	Floating-point ReSTORe fpu state. Восстановление состояния оборудования и файла регистров в памяти.
FCLEX	Floating-point CLear EXception. Сброс исключений.
FLDCW источник	Floating-point LoaD Control Word. Загрузка управляющего слова (16 бит) из источника в регистр управления SWR
FSTCW приемник	Floating-point STore Control Word. Сохранение управляющего слова в приемнике.
FSTSW приемник	Floating-point STore Status Word. Запись слова состояния из регистра управления SWR в приемник (регистр или память).

Работа со стеком.

Команда	Описание
FINCSTP	Floating-point INCrement Stack Top Pointer. Инкремент указателя стека.
FDECSTP	Floating-point DECrement Stack Top Pointer. Декремент указателя стека.

FFREE ST(i)	Floating-point FREE fpu stack register. Показать регистр ST(i) как свободный.

Вспомогательные.

FNOP	Floating-point No Operation. Холостая операция сопроцессора.
FWAIT	Floating-point WAIT. Ожидание процессором завершения текущей операции со-процессором.

Команды передачи данных. Команды передачи данных передают данные между регистрами стека и памятью. Эти команды можно разделить на группы:

- вещественные числа (с плавающей точкой);
- целые числа;
- десятичных числа;
- загрузка констант;
- обмен;
- условные пересылки.

Команды передачи вещественных чисел (с плавающей точкой).

Команда	Описание
FLD src	Floating-point LoaD real value Загрузить вещественное число в ST(0) (вершину стека) из области памяти. Область памяти может быть 32-, 64-, 80-битная.
FST приемник	Floating-point STore real value Сохранить вещественное число из верхушки стека в приемнике. Запись вещественного числа из ST(0) в память. Область памяти 32-, 64- или 80-битная.
FSTP приемник	Floating-point STore real value and Pop Сохранить вещественное число из верхушки стека в приемнике с выталкиванием из стека. Запись вещественного числа из ST(0) в память. Область памяти 32-, 64- или 80-битная. При этом происходит выталкивание вершины из стека.

Команды передачи целых чисел.

Команда	Описание
FILD источник	Floating-point Integer LoaD Загрузить целое число в ST(0) из памяти. Область памяти может быть 16-, 32-, 64-битной.
FIST приемник	Floating-point Integer STore Сохранить целое число из верхушки стека в приемнике. Запись целого числа из ST(0) в память. Область памяти 32-, 64- или 80-битная.

Команды передачи двоично-десятичных чисел.

Команда	Описание
FBLD источник	Floating-point Binary LoaD Загрузить BCD-число в ST(0) из 80-битной области памяти.
FBST приемник	Floating-point Binary STore decimal coded Сохранить в формате двоично-десятичного числа значение из верхушки стека в приемнике Запись BCD-числа в память. Область памяти 80-битная.
FBSTP приемник	Floating-point Binary STore decimal coded and Pop Сохранить в формате двоично-десятичного числа значение из верхушки стека в приемнике с выталкиванием из стека. Запись BCD-числа в память. Область памяти 80-битная. При этом происходит выталкивание вершины из стека.

Команды передачи констант.

Команда	Описание
FLDZ	Floating-point LoaDing Zero Загрузка константы 0 в верхушку стека сопроцессора. Загрузить 0 в ST(0).
FLD1	Floating-point LoaD constant 1 – загрузка константы 1 в верхушку стека сопроцессора. Загрузить1 в ST(0).
FLDPI	Floating-point LoaDing PI

	Загрузка константы PI в верхушку стека сопроцессора. Загрузить PI в ST(0).
FLDL2T	Floating-point LoaDing a binary (2) Logarithm Ten Загрузка константы двоичный логарифм 10 в верхушку стека сопроцессора. Загрузить LOG2(10) в ST(0).
FLDTL2E	Floating-point LoaDing a binary (2) Logarithm E Загрузка константы двоичный логарифм E=2.87.. в верхушку стека сопроцессора Загрузить LOG2(e) в ST(0).
FLDLG2	Floating-point LoaDing a decimal LoGarithm two (2) Загрузка константы десятичный логарифм 2 в верхушку стека сопроцессора Загрузить LG(2) в ST(0).
FLDLN2	Floating-point LoaDing Natural Logarithm two (2) Загрузка константы натуральный логарифм 2 в верхушку стека сопроцессора Загрузить LN(2) в ST(0).

Команды обмена.

Команда	Описание
FXCH st(i)	Floating-point eXCHange content Обмен значениями вершины стека и регистра стека с номерм i.

Команды сравнения данных Эти команды выполняют сравнение содержимого вершины стека с указанным в команде операндом.

Вещественные числа.

Команда	Описание
FCOM	Floating-point COMpare Сравнение вещественных чисел ST(0) и ST(1). Флаги устанавливаются, как при операции ST(0)-ST(1).
FCOM источник	Floating-point COMpare Сравнение ST(0) с операндом в памяти. Операнд может быть 32- или 64-битным.

FCOMP источник	Floating-point COMpare and Pop Сравнение вещественного числа в ST(0) с операндом с выталкиванием ST(0) из стека. Операнд может быть регистром и областью памяти.
FCOMPP источник	Floating-point COMpare and Pop fnd Pop Сравнение вещественного числа в ST(0) с операндом с двойным выталкиванием ST(0) и ST(1) из стека. Операнд может быть регистром и областью памяти.
FUCOMP ST(i)	Floating-point Unorder COMpare and Pop Сравнение ST(0) с ST(i) без учета порядков. При выполнении операции происходит выталкивание из стека.
FUCOMPP ST(i)	Floating-point Unorder COMpare and Pop and Pop Сравнение ST(0) с ST(i) без учета порядков. При выполнении операции происходит двойное выталкивание из стека.

Целые числа.

Команда	Описание
FICOM источник	Floating-point Integer COMpare Сравнение целых чисел в ST(0) с операндом. Операнд может быть 16- или 32-битным.
FICOMP источник	Floating-point Integer COMpare and Pop Сравнение целых чисел в ST(0) с операндом. Операнд может быть 16- или 32-битной областью памяти или регистром. При выполнении операции происходит выталкивание ST(0) из стека.

Анализ

Команда	Описание
FTST	Floating-point TeST zero Проверка ST(0) на нуль.
FXAM	Floating-point eXAMine Анализ содержимого вершины стека. Результат помещается в биты C3-C0 регистра CWR: • Знак =.> бит C1, • c3c2c0 = 000 - неподдерживаемый формат.

	• c3c2c0 = 001 - не число. • c3c2c0 = 010 - нормализованное число. • c3c2c0 = 011 - бесконечность. • c3c2c0 = 100 – нуль • c3c2c0 = 101 - пустой операнд. • c3c2c0 = 110 - денормализованное число.

Арифметические команды. В группу арифметических команд входят команды, реализующие операции сложения, вычитания, умножения и деления. Арифметические команды можно разделить на две подгруппы:

- для работы с целочисленными операндами.
- для работы с вещественными операндами.

Команды для работы с вещественными операндами.

Команда	Описание
FADD источник FADD ST(i),ST(0)	Floating-point ADDition Сложение. • ST(0) = ST(0) + источник (32- или 64-битное число). • ST(i) = ST(i) + ST(0)
FADDP ST(i),ST(0)	Floating-point ADDition and Pop Сложение. ST(i) = ST(i) + ST(0). Выталкивание из стека ST(0).
FSUB источник FSUB ST(i),ST(0)	Floating-point SUBtraction Вычитание. • ST(0) = ST(0) - источник (32- или 64-битное число). • ST(i) = ST(i) - ST(0).
	Floating-point SUBtraction Вычитание. ST(i) = ST(i) - ST(0).
FSUBP ST(i),ST(0)	Floating-point SUBtraction and Pop Вычитание. ST(i) = ST(i) - ST(0). Выталкивание из стека ST(0).
FSUBR ST(i),ST(0)	Floating-point SUBtracttion Revers Обратное вычитание. ST(0) = ST(i) - ST(0)
FSUBRP	Floating-point SUBtracttion Revers and Pop

ST(i),ST(0)	Обратное вычитание ST(0). ST(0) = ST(i) - ST(0). Выталкивание из стека ST(0).
FMUL FMUL ST(i) FMUL ST(i),ST	Floating-point MULtiplay with real value Умножение: • ST(0) = ST(0) * ST(1) • ST(0) = ST(i) * ST(0) • ST(i) = ST(i) * ST(0).
FMULP ST(i),ST(0)	Floating-point MULtiplay and Pop Умножение. ST(i) = ST(i) * ST(0)
FDIV FDIV ST(i) FDIV ST(i),SY	Floating-point DIVide Деление двух вещественных чисел.: • ST(0) = ST(0) / ST(1) • ST(0) = ST(0) / ST(i) • ST(i) = ST(0) / ST(i)
FDIVP ST(i),ST(0)	Floating-point DIVide and Pop Деление. ST(i)<-ST(0)/ST(i). Выталкивание из стека ST(0).
FDIVR ST(i),ST(0)	Floating-point DIVide Revers Обратное деление. ST(0) = ST(i) / ST(0)
FDIVRP ST(i),ST(0)	Floating-point DIVide Revers and Pop Обратное деление ST(0) = ST(i) / ST(0) Выталкивание из стека ST(0).

Команды для работы с целочисленными операндами.

Команда	Описание
FIADD источник	Floating-point Integer ADDition Сложение. ST(0) = ST(0) + источник (16- или 32-битное число).
FISUB источник	Floating-point Integer SUBtracttion Вычитание. ST(0) = ST(0) - источник (16- или 32-битное число).
FISUBR источник	Floating-point Integer SUBtracttion Revers Вычитание. ST(0) = источник (16- или 32-битное число) -.ST(0)
FIMUL источник	Floating-point Integer MULtiply

	Умножение. ST(0) = ST(0) * источник (16- или 32-битное число).
FIDIV источник	Floating-point Integer DIVide Деление. ST(0) = ST(0) / источник (16- или 32-битное число).
FIDIVR источник	Floating-point Integer DIVide Revers Обратное деление целых чисел. ST(0) = источник / ST(0).

Вспомогательные команды.

Команда	Описание
FABS	Floating-point ABSolute value Нахождение абсолютного значения. ST(0) = ABS(ST(0)).
FCHS	Floating-point CHange Sign Изменение знака ST(0) = -ST(0).
FSQRT	Floating-point calculation of SQuare Root Извлечь квадратный корень из ST(0) и поместить обратно. Исходный операнд предварительно должен быть занесен в стек ST(0) = X.
FSCALE	Floating-point SCALE. Масштабирование. Умножение X на 2 в степени Y. Исходные операнды предварительно должны быть занесены в стек ST(0) = X, ST(1) = Y. ST(0) = ST(0) * 2^ST(1).
FPREM	Floating-point Partial REMinder Нахождение частичного остатка от деления ST(0) на ST(1).
FPREM1	Floating-point Partial REMinder IEEE-754 Нахождение частичного остатка от деления ST(0) на ST(1) (в стандарте IEEE).
FRNDINT	Floating-point RouND INTeger Округление до ближайшего целого числа, находящегося в ST(0). ST(0) = int(ST(0)).

Тригонометрические.

Команда	Описание
FCOS	Floating-point calculation of COSine. Косинус. ST(0) = COS(ST(0)). Содержимое в ST(0) интерпретируется как угол в радианах.
FPTAN	Floating-point Partial TANgent. Частичный тангенс. Содержимое в ST(0) интерпретируется как угол в радианах. Значение тангенса возвращается на место аргумента, а затем в стек включается 1.
FPATAN	Floating-point Partial ArcTANgent. Частичный арктангенс. Вычисляется функция Arctg(ST(1)/ST(0)). После вычисления функции происходит выталкивание из стека, после чего значение функции помещается в вершину стека
FSIN	Floating-point calculation of SINe. Синус. ST(0) = SIN(ST(0)). Содержимое в ST(0) интерпретируется как угол в радианах.
FSINCOS	Floating-point calculation of SINe and COSine. Синус и косинус. ST(0) = SIN(ST(0)) и ST(1) = COS(ST(0))

Логарифмы и степени.

Команда	Описание
F2XM1	Floating-point 2^X minus 1 Вычисление 2^X - 1. ST(0) = 2^ST(0) - 1.
FYL2X	Floating-point compute Y*Log2(X) Вычисление Y*LOG2(X). Исходные операнды предварительно должны быть занесены в стек ST(0) = Y, ST(1) = X. Резкльтат помещается в ST(1). Затем происходит выталкивание из стека, и результат оказывается в вершине стека ST(0).
FYL2XP1	Floating-point compute Y*Log2(X Plus 1) Вычисление Y*LOG2(X + 1). Исходные операнды предварительно должны быть занесены в стек ST(0) = Y, ST(1) = X. Резкльтат помещается в ST(1). Затем происходит выталкивание из стека, и результат оказывается в вершине стека ST(0).

3.8.5. Целочисленное MMX расширение

Обработка данных MMX-расширения может выполняться с использованием:

- циклической арифметики (wraparound arithmetic),
- арифметики с насыщением (saturation arithmetic).

Большинство команд технологии MMX обрабатывают данные по правилам циклической арифметики, а некоторые команды задействуют арифметику с насыщением. Если команда задействует циклическую арифметику (другое название — арифметика с циклическим переносом) и результат операции выходит за двоичную разрядную сетку используемого типа данных, то «лишние» старшие биты результата отбрасываются.

Если команда использует арифметику с насыщением и результат операции превышает максимальное представимое значение, то в выходной операнд записывается это максимальное значение (происходит «насыщение»). Аналогично, если результат операции оказывается меньше нижней границы допустимого диапазона, то в выходной операнд записывается минимальное возможное значение.

В арифметике с насыщением MMX-команды сложения, вычитания и упаковки данных могут обрабатывать числа со знаком или без знака. Данные со знаком и без знака имеют различный допустимый диапазон. Следовательно, если используется арифметика с насыщением, то при выходе результата операции за пределы допустимого диапазона в выходной операнд записываются различные значения, в зависимости от типа данных.

Синтаксис MMX-команд. Большинство команд в мнемонике имеют **суффикс**, который определяет тип данных и используемую арифметику:

- US (Unsigned Saturation) — арифметика с насыщением, данные без знака или, по-другому, беззнаковое насыщение. Если команда использует арифметику с насыщением и результат операции превышает максимальное представимое значение, то в выходной операнд записывается это максимальное значение (происходит «насыщение»). Аналогично, если результат операции оказался меньше нижней границы допустимого диапазона, то в выходной операнд записывается минимально возможное значение.
- S или SS (Signed Saturation) — арифметика с насыщением для данных со знаком (или знаковое насыщение).
- Если в суффиксе нет ни символа S ни символов SS, то применяется циклическая арифметика (wraparound). Если в этом случае результат операции

выходит за двоичную разрядную сетку используемого типа данных, то «лишние» старшие биты результата отбрасываются.

- b, w, d, q — эти буквы в конце имени указывают тип данных. Если в суффиксе есть две из этих букв, то первая соответствует входному операнду, вторая — выходному.

Синтаксис MMX команд:

- Мнемоника приемник, источник
- Мнемоника приемник, источник, маска
- Мнемоника приемник

После обзора команды будут рассмотрены подробно.

Категория	Подкатегория	Команды
Инициализация		EMMS
Передача данных	Пересылки	MOVD, MOVQ
Упаковка данных		PACKSSWB, PACKSSDW – числа со зна PACKUSWB – числа без знака
Распаковка данных		PUNPCKHBW, PUNPCKHWD, PUNPCKH PUNPCKLBW, PUNPCKLWD, PUNPCKLD
Арифметика	Сложение Вычитание Умножение	PADD, PADDS, PADDUS PSUB, PSUBS, PSUBUS PMULHW, PMULLW, PMADDWD
Логика		PAND, PANDN, POR, PXOR
Сдвиги		PSLLW, PSLLD, PSLLQ – логические вле PSRLW, PSRLD, PSRLQ – логические вп PSRAW, PSRAD – арифметические (впр
Сравнения		PCMPEQB, PCMPEQW, PCMPEQD – на PCMPGTB, PCMPGTW, PCMPGTD – на
Дополнительные	Вычисления	PAVGB, PAVDW - среднее значеие PSADBW – сумма разностей
	Извлечения	PEXTRW – извлечь слово PINSRW – вставить слово PMAXUB, PMAXSW – извлечь максимал PMINUB, PMINSW – извлечь минимальн
	Маска из знаков байтов	PMOVMSKB

Инициализация.

Команда	Описание
EMMS	Empty MMx State Очистка стека регистров MMX. Установка всех единиц в слове тегов.

Передача данных. Данные передаются между:

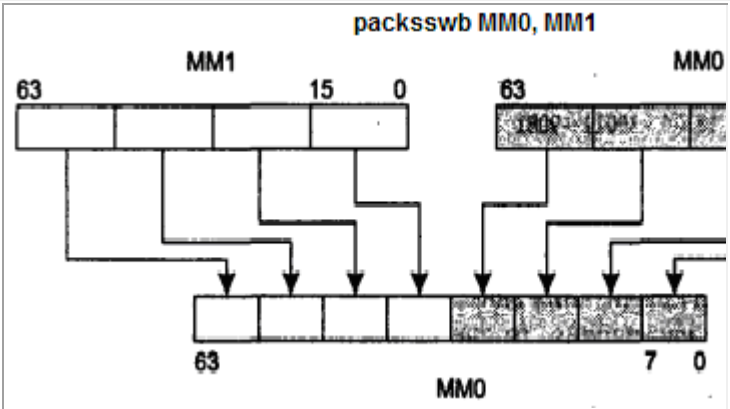
- регистрами MMX,
- регистром MMX и регистром основного процессора,
- регистром MMX и памятью.

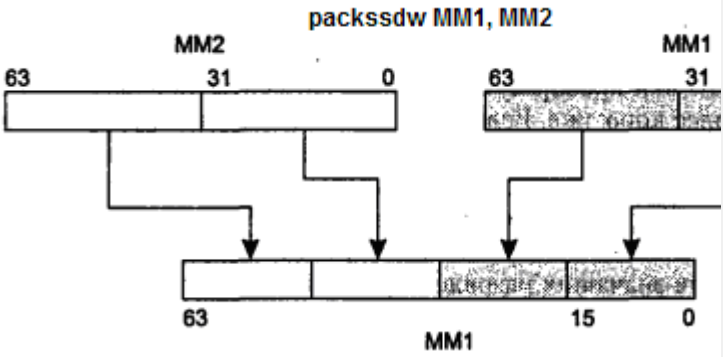
Если размер приемника меньше размера источника, то лишние разряды обнуляются.

Команда	Описание
MOVD приемник, источник	MOVe Double word – переместить двойное слово (32 разряда).
MOVQ приемник, источник	MOVe Quarter word – переместить учетверенное слово (64 разряда).

Упаковка данных. MMX-команды упаковки преобразуют длинные элементы данных A6- и 32-разрядные слова) в более короткие. Если исходное значение «не помещается» в коротком элементе данных, то происходит «насыщение» — результатом считается граничное значение допустимого диапазона выходного типа данных.

Мнемоника	Описание
PACKSSWB	PACK with Signed Saturation Words to Bytes Упаковка со знаковым насыщением слов приемника (регистра MMX) источника (память или регистр MMX) в байты, расположенные в приемнике (регистре MMX). Пример:

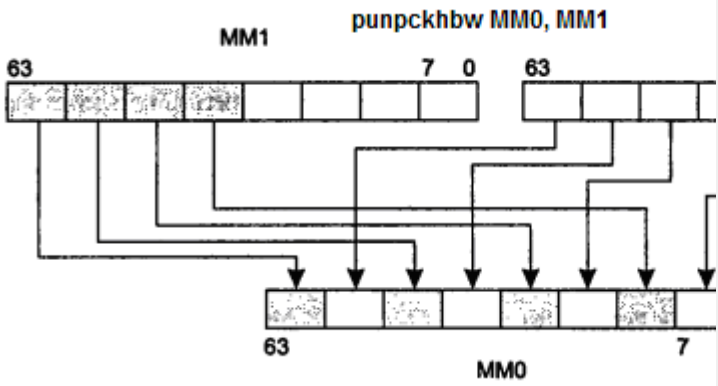
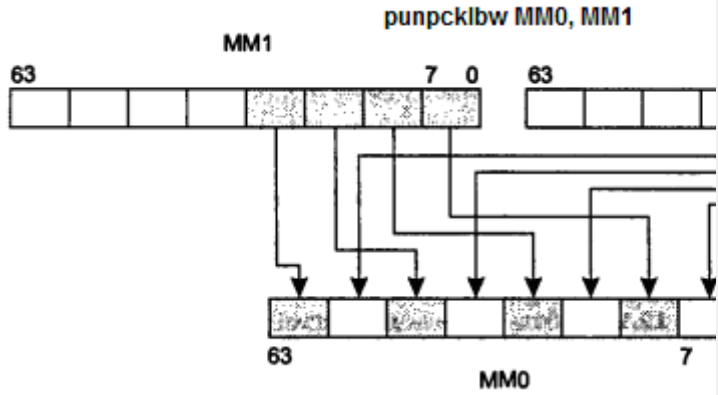
	 Слова приемника MM0 упаковываются в байты младшей части приемника MM0. Слова источника MM1 упаковываются в байты старшей части приемника MM0.
PACKUSWB	PACK with Unigned Saturation Words to Bytes Упаковка с беззнаковым насыщением знаковых слов приемника (регистра MMX) источника (память или регистр MMX) в байты без знака, расположенные в приемнике (регистре MMX). Отрицательные числа преобразуются в нули.
PACKSSDW	PACK with Signed Saturation Double words to Words Упаковка со знаковым насыщением двойных слов приемника (регистра MMX) и источника (память или регистр MMX) в слова, расположенные в приемнике (регистре MMX). Пример:

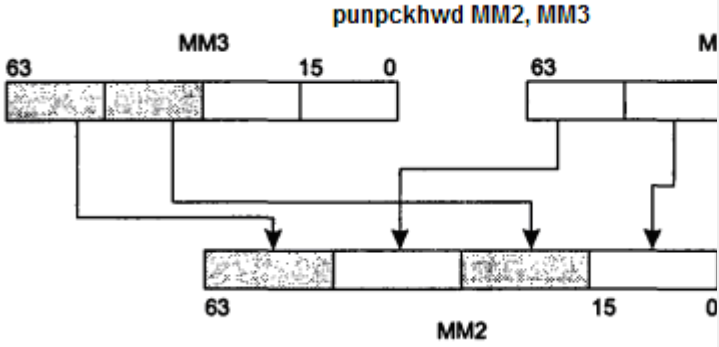
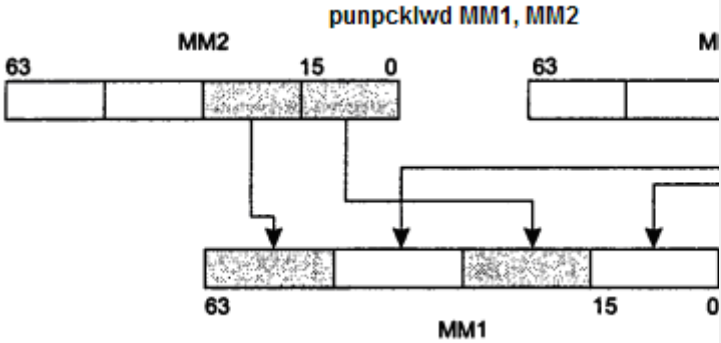


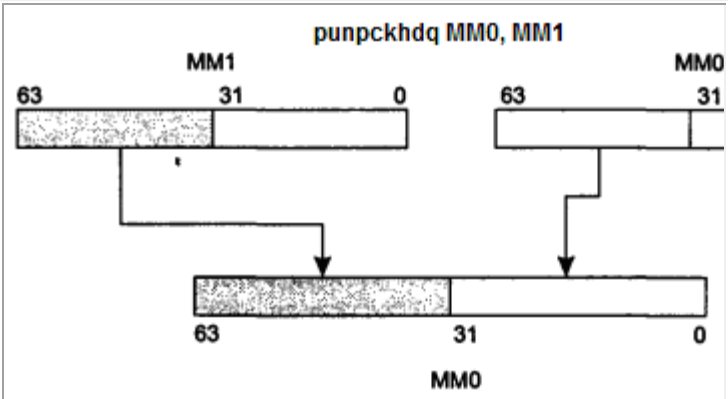
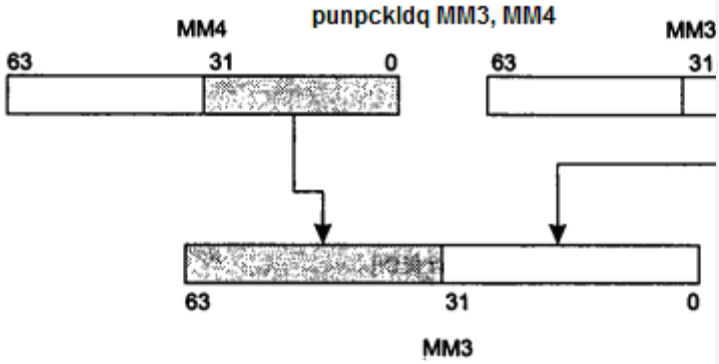
Двойные слова приемника MM1 упаковываются в байты младшей части применика MM1.
Двойные слова источника MM2 упаковываются в байты старшей части применика MM1.

Распаковка данных. MMX-команды распаковки попарно объединяют элементы данных из обоих операндов в более длинные элементы выходного операнда. Этими командами можно пользоваться для увеличения числа значащих разрядов при вычислениях.

Мнемоника	Описание
PUNPCKHBW	P UNPaCK High packed Byfes to Words Распаковывает старшие части приемника и источника в приемник. Происходит чередование в регистре назначения байт старшей половины источника с байтами старшей половины приемника.

	 Байты старшей половины приемника MM0 распаковываются в нечетные байты приемника MM1. Байты старшей половины источника MM1 распаковываются в четные байты приемника MM0.
PUNPCKLBW	P UNPaCK Low packed Byfes to Words Распаковывает младшие части приемника и источника в приемник. Происходит чередование в регистре назначения байт младшей половины источника с байтами младшей половины приемника. Пример: 
PUNPCK	P UNPaCK High packed Words to Double words

HWD	Распаковывает старшие части приемника и источника в приемник. Происходит чередование в регистре назначения слов старшей половины источника со словами старшей половины приемника. 
PUNPCK LWD	P UNPaCK Low packed Words to Double words Распаковывает младшие части приемника и источника в приемник. Происходит чередование в регистре назначения слов младшей половины источника со словами младшей половины приемника. Пример: 
PUNPCK HDQ	P UNPaCK High packed Double words to Quad words. Распаковывает старшие части приемника и источника в приемник. Происходит чередование в регистре назначения слов старшей половины источника со словами старшей половины приемника.

	 .
PUNPCK LDQ	P UNPaCK Low packed Double words to Quad words. Распаковывает младшие части приемника и источника в приемник. Происходит чередование в регистре назначения двойных слов младшей половины источника с двойными словами младшей половины приемника. Пример: 

Сложение и вычитание. MMX-команды сложения и вычитания работают с упакованными байтами и словами со знаком и без знака, а также с упакованными двойными словами со знаком. Они могут использовать как циклическую арифметику, так и арифметику с насыщением.

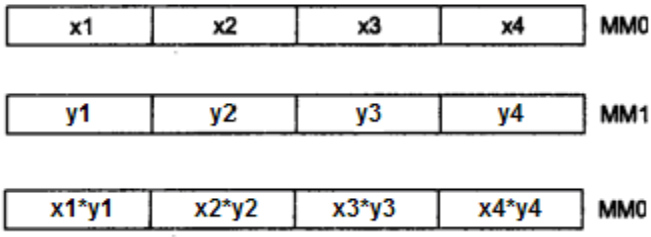
Входной операнд – регистр MMX или память.

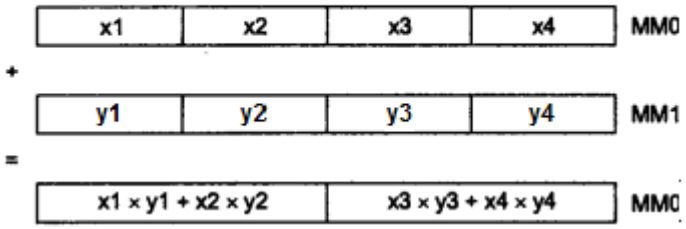
Выходной операнд – регистр MMX.

Мнемоника	Описание
PADDB	Packed ADDition Bytes Сложение упакованных байт. без насыщения (с циклическим переполнением)
PADDW	Packed ADDition Words Сложение упакованных слов. без насыщения (с циклическим переполнением).
PADDD	Packed ADDition Double words Сложение упакованных двойных слов без насыщения (с циклическим переполнением).
PADDSB	Packed ADDition with sighed Saturation Bytes Сложение упакованных байт со знаковым насыщением.
PADDSW	Packed ADDition with sighed Saturation Words Сложение упакованных слов со знаковым насыщением.
PADDUSB	Packed ADDition with Unsighed Saturation Bytes Сложение упакованных байт с беззнаковым насыщением.
PADDUSW	Packed ADDition with Unsighed Saturation Words Сложение упакованных слов с беззнаковым насыщением. Пример; paddusw MM0, MM1 63 15 0 34 30544 45012 1093 MM0 + 99 39218 14077 34 MM1 = 133 65535 59089 1127 MM0 Серым цветом выделено слово результата, в котором произошло насыщение.
PSUBB	Packed SUBtraction Bytes Вычитание упакованных байт без насыщения (с циклическим антипереполнением).
PSUBW	Packed SUBtraction Words

	Вычитание упакованных слов без насыщения (с циклическим анти-переполнением).
SUBD	Packed SUBtraction Double words Вычитание упакованных двойных слов без насыщения (с цикличе-ским антипереполнением).
PSUBSB	Packed SUBtraction with signed Saturanion Bytes Вычитание упакованных знаковых байт с насыщением.
PSUBSW	Packed SUBtraction with signed Saturanion Words Вычитание упакованных знаковых слов с насыщением.
PSUBUSB	Packed SUBtraction with Unsigned Saturanion Bytes Вычитание упакованных беззнаковых байт с насыщением.
PSUBUSW	Packed SUBtraction with Unsigned Saturanion Bytes Вычитание упакованных беззнаковых слов с насыщением.

Умножение. MMX-команды умножения попарно перемножают 16-разрядные слова операндов, что дает четыре 32-разрядных произведения. Все команды формируют результат по принципу циклической арифметики:

Мнемоника	Описание
PMULHW	Packed Multiplay and return High Words Попарное умножение 4 знаковых слов источника (регистр MMX или память) на 4 знаковых слова приемника (регистр MMX). Результатом операции являются 4 32-разрядных произведения, при этом старшие разряды произведений сохраняются в 16-разрядных словах приемника, а младшие разряды произведений теряются. Пример: pmulhw MM0,MM1 

PMULLW	Packed Multiplay and return Low Words Попарное умножение 4 знаковых слов источника (регистр MMX или память) на 4 знаковых слова приемника (регистр MMX). Результатом операции являются 4 32-разрядных произведения, при этом младшие разряды произведений сохраняются в 16-разрядных словах приемника, а старшие разряды произведений теряются.
PMADDWD	Packed Multiplay and ADD Word to Double word Попарное умножение 4 знаковых слов источника (регистр MMX или память) на 4 знаковых слова приемника (регистр MMX). Два двойных слова результатов умножения младших слов суммируются и записываются в младшее двойное слово операнда назначения. Два двойных слова результатов умножения старших слов суммируются и записываются в старшее двойное слово операнда назначения. Пример: pmaddwd MM0, MM1 

Логика. Логические MMX-команды выполняют поразрядные логические операции над всеми 64 битами своих операндов. Они реализуют логические операции И, ИЛИ, И-НЕ, исключающего ИЛИ:

Мнемоника	Описание
PAND	Packed logical AND Логическое "И".
PANDN	Packed logical AND and Not Логическое "И-НЕ".
POR	Packed logical OR Исключающее "ИЛИ".
PXOR	Packed logical eXclusive OR

	Исключающее "ИЛИ".

Сдвиги. MMX-команды сдвига выполняют сдвиг каждого элемента данных (16-, 32- или 64-разрядного слова) в приемнике на величину, задаваемую источником. Среди команд сдвига выделяют команды арифметического и логического сдвига. При выполнении команд арифметического сдвига освобождающиеся разряды элементов заполняются знаком числа (старший бит) и могут принимать значение как 0, так и 1, в то время как при выполнении команд логического сдвига освободившиеся разряды заполняются нулями.

Мнемоника	Описание
PSLLW	Packed Shift Left Logical Words Логический сдвиг влево упакованных слов приемника на количество бит, указанных в источнике, с заполнением младших бит нулями.
PSLLD	Packed Shift Left Logical Double words Логический сдвиг влево упакованных двойных слов приемника на количество бит, указанных в источнике, с заполнением младших бит нулями.
PSLLQ	Packed Shift Left Logical Quarter words Логический сдвиг влево упакованного учетвернного слова приемника на количество бит, указанных в источнике, с заполнением младших бит нулями.
PSRLW	Packed Shift Right Logical Words Логический сдвиг вправо упакованных слов приемника на количество бит, указанных в источнике, с заполнением старших бит нулями.
PSRLD	Packed Shift Right Logical Double words Логический сдвиг вправо упакованных двойных слов приемника на количество бит, указанных в источнике, с заполнением старших бит нулями.
PSRLQ	Packed Shift Right Logical Quarter words Логический сдвиг вправо упакованного учетверенного слова приемника на количество бит, указанных в источнике, с заполнением старших бит нулями.
PSRAW	Packed Shift Right Arithmetic Words

	Арифметический сдвиг вправо упакованных знаковых слов приемника на количество бит, указанных в источнике, с заполнением освобождаюшихся бит битами знаковых разрядов.
PSRAD	Packed Shift Right Arithmetic Double words Арифметический сдвиг вправо упакованных двойных знаковых слов премника на количество бит, указанных в источнике, с заполнением освобождаюшихся бит битами знаковых разрядов.

Сравнения. MMX-команды сравнения попарно сравнивают элементы данных (байты, 16- или 32-разрядные слова) входного и выходного операндов. В зависимости от результата сравнения соответствующий элемент данных выходного операнда заполняется нулями либо единицами. Эти команды, как и все остальные MMX-команды, не устанавливают флагов (признаков). В свою очередь, они делятся на две группы: команды обычного сравнения (равно или не равно) и команды сравнения по величине (больше или меньше). Операции сравнения проводятся для упакованных байтов, слов и двойных слов.

Мнемоника	Описание
PCMPEQB	Packed CoMParE for Equal Byte Попарное сравнение (на равенство) упакованных байт. Все биты элемента результата будут единичными (true) при совпадении соответствующих элементов операндов и нулевыми (false) - при несовпадении.
PCMPEQW	Packed CoMParE for Equal Word Попарное сравнение (на равенство) упакованных слов. Все биты элемента результата будут единичными (true) при совпадении соответствующих элементов операндов и нулевыми (false) - при несовпадении.
PCMPEQD	Packed CoMParE for Equal Double word Попарное сравнение (на равенство) упакованных двойных слов. Все биты элемента результата будут единичными (true) при совпадении соответствующих элементов операндов и нулевыми (false) - при несовпадении.
PCMPGTB	Packed CoMParE for Greater Than Byte Попарное сравнение (по величине) упакованных знаковых байт. Все биты элемента результата будут единичными (true), если соответствующий элемент приемника больше элемента источни-

	ка, и нулевыми (false) в противном случае.
PCMPGTW	Packed CoMParE for Greater Than Word Попарное сравнение (по величине) упакованных знаковых слов. Все биты элемента результата будут единичными (true), если соответствующий элемент приемника больше элемента источника, и нулевыми (false) в противном случае.
PCMPGTD	Packed CoMParE for Greater Than Double word Попарное сравнение (по величине) упакованных знаковых двойных слов. Все биты элемента результата будут единичными (true), если соответствующий элемент приемника больше элемента источника, и нулевыми (false) в противном случае.

Дополнительные команды. Сейчас мы рассмотрим еще одну группу команд, которые трудно отнести к какому-либо определенному типу, но которые являются весьма полезными при разработке программ.

Мнемоника	Описание
PAVGB	Packed AveraGe Bytes Попарно вычисляет средние значения упакованных чисел, представленных байтами. Значения операндов интерпретируются как беззнаковые целые числа. В качестве источника могут выступать MMX-регистр или 64-разрядная ячейка памяти, приемником служит один из MMX-регистров.
PAVGW	Packed AveraGe Words Попарно вычисляет средние значение упакованных чисел, представленных словами. Значения операндов интерпретируются как беззнаковые целые числа. В качестве источника могут выступать MMX-регистр или 64-разрядная ячейка памяти, приемником служит один из MMX-регистров.
PEXTRW	Packed EXTRact Word Извлекает одно из 4 упакованных слов входного операнда. Команда имеет три аргумента: источние, приемник и маска. Младшие два бита маски задают в источнике номер упакованного слова, подлежащего извлечению. Извлеченное слово сохраняется в младшем слове приемника. Приемником этой команды может выступать один из 32-разрядных регистров общего назначения. Старшее слово приемника обнуляется.

PINSRW	Packed INSeRt Word Вставляет слово в одно из 4 упакованных слов приемника. Приемником является один из MMX-регистров, а источником может выступать один из 32-разрядных регистров общего назначения, младшее слово которого будет вставлено в приемник. Номер позиции, куда помещается операнд, определяется младшими двумя битами маски и может принимать значения от 0 до 3;
PMAXUB	Packed MAXimum Unsigned integer Byte Извлекает максимальное значение из каждой пары упакованных элементов в приемнике и источнике. Операция выполняется над беззнаковыми байтами . В качестве приемника может выступать MMX-регистр, а в качестве источника — MMX-регистр или 64-разрядная ячейка памяти.
PMAXSW	Packed MAXimum Signed integer Word Извлекает максимальное значение из каждой пары упакованных элементов в приемнике и источнике. Операция выполняется над знековыми словами. В качестве приемника может выступать MMX-регистр, а в качестве источника — MMX-регистр или 64-разрядная ячейка памяти.
PMINUB	Packed MINimum Unsigned integer Byte Извлекает минимальное значение из каждой пары упакованных элементов в приемнике и источнике. Операция выполняется над беззнаковыми байтами . В качестве приемника может выступать MMX-регистр, а в качестве источника — MMX-регистр или 64-разрядная ячейка памяти.
PMINSW	Packed MINmum Signed integer Word Извлекает минимальное значение из каждой пары упакованных элементов в приемнике и источнике. Операция выполняется над знековыми словами. В качестве приемника может выступать MMX-регистр, а в качестве источника — MMX-регистр или 64-разрядная ячейка памяти.
PMOVMSKB	Packed MOVe MaSK Byte to integer Формирует байтовую маску, содержащий знаковые биты 8 байтов, содержащихся в источнике, в качестве которого может выступать один из MMX-регистров. Примником является 32-разрядный регистр общего назначения, младший байт которого будет содержать

	результат. Эта команда очень удобна для формирования условных ветвлений в программах.
PSADBV	Packed Sum of Absolute Differences Вычисляет сумму абсолютных значений разностей значений беззнаковых байтов источника и приемника. Источником могут выступать MMX-регистр или 64-разрядная ячейка памяти, а приемником — один из MMX-регистров.

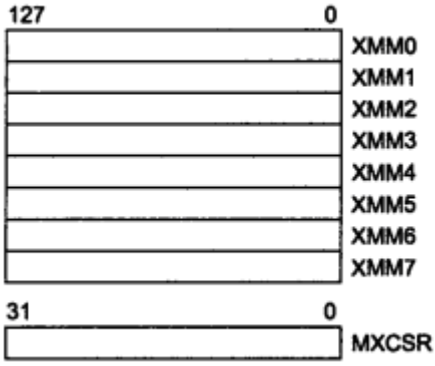
3.8.6. XMM расширение с плавающей точкой

Синтаксис XMM команд:

- Мнемоника приемник, источник
- Мнемоника приемник, источник, маска
- Мнемоника приемник

Приемник – всегда регистр XMM, источник - регистр XMM или ячейка памяти,

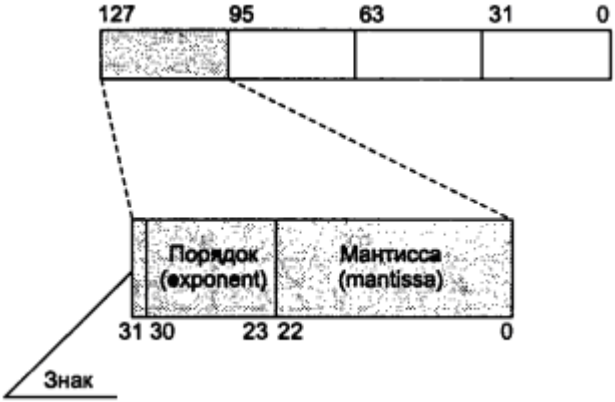
Типы данных. XMM-расширение реализовано в виде аппаратно-программного модуля SSE, который включает дополнительные восемь регистров разрядностью в 128 бит, имеющих обозначение XMM0 - XMM7,



и 32-разрядный регистр управления/состояния MXCSR.

Программная часть XMM-расширения включает в себя набор команд для работы с данными в формате плавающей точки. Содержимое XMM-регистра представляет собой четыре 32-разрядных операнда с плавающей точкой в коротком формате (Single Precision Floating Point, SPFP). Представление данных SSE-

расширения соответствует стандарту IEEE-754, а сам формат данных показан на рисунке.



Здесь мантисса (mantissa) и порядок (exponent) формируют число в формате SPFP в соответствии с формулой

$$\text{мантисса} * 2^{\text{порядок}}.$$

Этот формат данных несопоставим с тем, который принят для математического сопроцессора (число в 80-разрядном расширенном формате), поэтому в некоторых случаях при разных границах выравнивания результаты вычислений с использованием форматов FPU и SSE могут различаться.

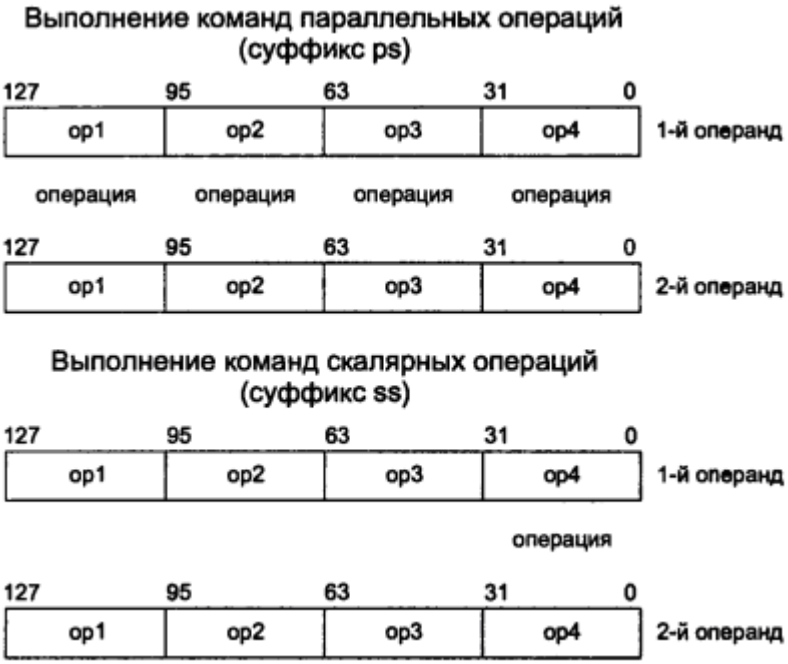
Поскольку аппаратно модуль SSE-расширения реализован независимо от других модулей, то это позволяет выполнять XMM-комапды параллельно с командами математического сопроцессора и MMX-командами. При этом для синхронизации вычислений инструкции наподобие EMSS не требуются.

Структура полей регистра управления/состояния (MXCSR) во многом напоминает ту, что реализована в регистрах состояния (swr) и управления (cwr) математического сопроцессора. Состоянием вычислений можно управлять путем установки определенных значений в поля этого регистра.

Набор инструкций XMM-расширения включает 70 команд.

Значительная часть команд может выполняться в двух контекстах: скалярном и параллельном. Это относится к арифметическим командам, а также к командам сравнения. Команды обрабатывают:

- В параллельном контексте одновременно 4 двойных слова и имеют в своей мнемонике суффикс ps.
- В скалярном контексте только младшие 32-разрядные двойные слова упакованных операндов, не затрагивая при этом три старших двойных слова. Исключение составляют некоторые команды скалярной пересылки данных. Мнемоническое обозначение этих команд включает суффикс ss.



В процессе обработки данных команды XMM-расширения могут возбуждать исключительные ситуации, которые возникают, если происходит одно из следующих событий:

- некорректная операция (invalid operation);
- денормализованный операнд (denormalized operand);
- деление на 0 (divide-by-zero);
- арифметическое переполнение (numeric overflow);
- потеря значащих разрядов (numeric underflow);

- потеря точности (inexact result).

При возникновении исключительных ситуаций устанавливаются биты 0-5 в регистре управления/состояния (MXCSR). Каждая исключительная ситуация может быть замаскирована путем установки в 1 битов 7-12 регистра MXCSR. Если какое-либо исключение замаскировано, то оно обрабатывается процессором по стандартному алгоритму, после чего продолжается выполнение программного кода. Формат полей регистра MXCSR и их назначение показаны на рис. 13.5.

Биты 13-14 регистра MXCSR поля RC (или rc, что одно и то же) задают режим округления. По умолчанию устанавливается режим округления к ближайшему значению числа с плавающей точкой в коротком формате. Эти биты можно установить программно, причем возможны следующие комбинации:

- 00 — округление к ближайшему числу;
- 01 — округление к меньшему числу;
- 10 — округление к большему числу;
- 11 — округление с отбрасыванием дробной части.

Бит 15 используется, если результат операции близок к нулю. При этом процессор выполняет следующие действия:

- возвращает значение 0 и знак результата;

- устанавливает флаги P (бит 5) и U (бит 4);
- маскирует биты исключений.

Программная реализация SSE-расширения включает несколько десятков команд. Все их условно можно разделить на несколько групп:

- команды передачи данных;
- арифметические команды;
- команды сравнения;
- команды преобразования;
- логические команды;
- дополнительные команды.
- Классификация команд

После обзора команды будут рассмотрены подробно.

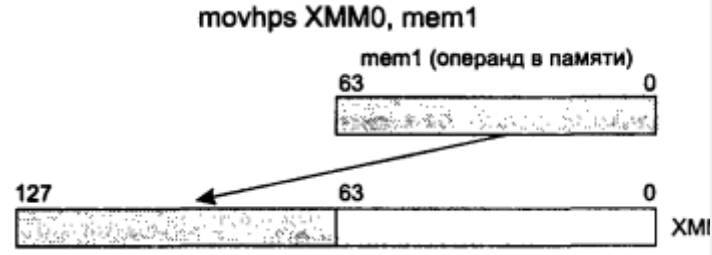
Категория	Подкатегория	Команды
Передача данных		
Арифметика	Скалярные	
	Параллельные	
Сравнение		
Преобразования		
Логика		
Дополнительные	Вычисления	
	Извлечения	
	Извлечения знака	

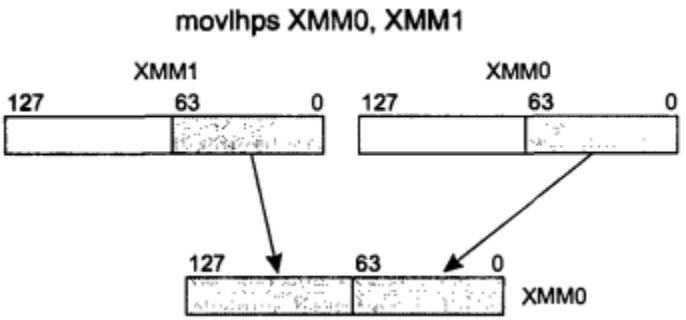
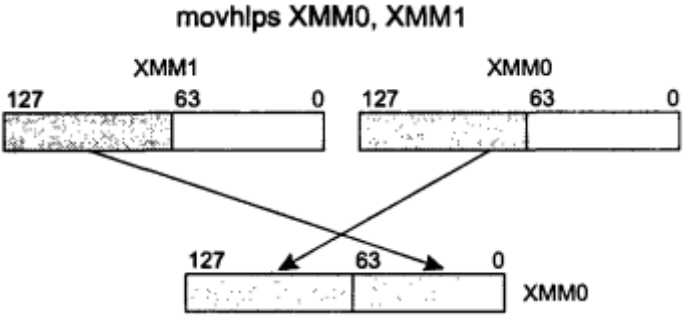
Передача данных. Данные передаются между:

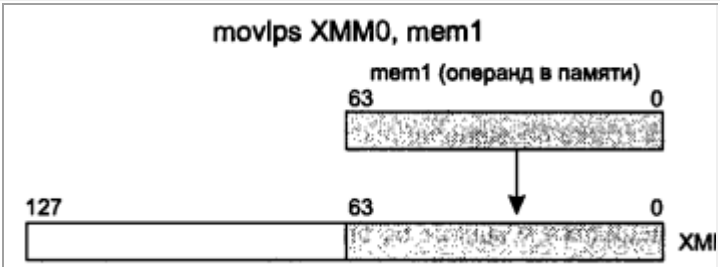
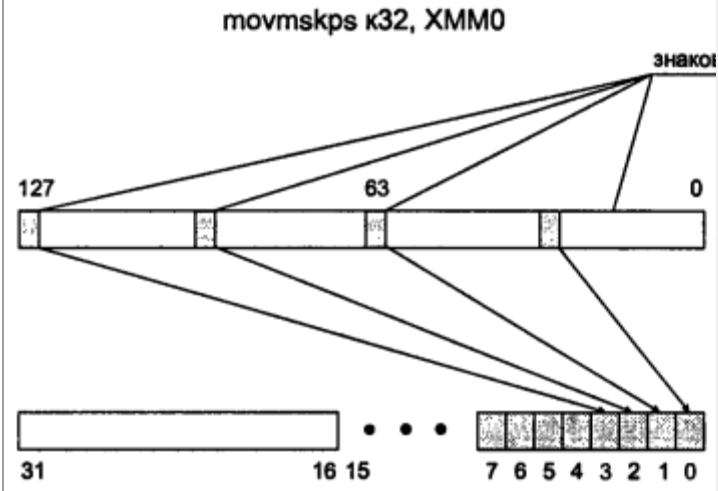
- регистрами MMX,
- регистром MMX и регистром основного процессора,
- регистром MMX и памятью.

Если размер приемника меньше размера источника, то лишние разряды обнуляются.

Мнемоника	Описание
MOVAPS	MOVe Aligned four Packed Single precision float point

	Пересылка выровненных по 16-байтовой границе 128-разрядных операндов. В качестве входного операнда могут выступать XMM-регистр или 128-разрядная ячейка памяти, выходным операндом может служить один из XMM-регистров. Если адрес ячейки памяти не будет выровнен по 16-байтовой границе, то произойдет исключение общей защиты.
MOVUPS	MOVe Unaligned four Packed Single precision float point Пересылка невыровненных данных; в качестве входного операнда могут выступать XMM-регистр или 128-разрядная ячейка памяти, выходным операндом может служить один из XMM-регистров. Команда не требует выравнивания по 16-байтовой границе..
MOVHPS	MOVe unaligned High Packed Single precision float point Пересылка невыровненных 64 бит из входного операнда в выходной. Один из операндов обязательно должен быть XMM-регистром, в качестве второго может выступать 64-разрядная ячейка памяти. Пересылаются только старшие 64 бит входных операндов. Младшие 64 бит обоих операндов не изменяются. Если данные передаются из XMM-регистра, то пересылке подлежат только старшие 64 бит. Команда не требует выравнивания по 16-байтовой границе адреса ячейки памяти. Пример: 
MOVLHPS	MOVe Low to High Packed Single precision float point Пересылка невыровненных 64 бит из входного операнда в выходной. Оба операнда должны находиться в XMM-регистрах. Пересылаются только старшие 64 бит входных операндов. В результате выполнения этой команды изменяются младшие 64 бит приемника. Пример:

	
MOVHLPS	MOVe High to Low Packed Single precision float point Пересылка невыровненных 64 бит из входного операнда в выходной. Оба операнда должны находиться в XMM-регистрах. Пересылаются только старшие 64 бит входных операндов. В результате выполнения этой команды изменяются младшие 64 бит регистра-приемника. Пример: 
MOVLPS	MOVe unaligned Low Packed Single precision float point пересылка невыровненных 64 бит из входного операнда в выходной. Один из операндов обязательно должен быть XMM-регистром, в качестве второго может выступать 64-разрядная ячейка памяти. Пересылаются толь- ко младшие 64 бит входных операндов. Старшие 64 бит обоих операндов не изменяются. Если данные передаются из XMM-регистра, то пересылке подлежат только младшие 64 бит. Команда не требует выравнивания по 16-байтовой границе адреса ячейки памяти. Пример:

	
MOVMSKPS	MOVe sign MaSK Packed Single precision float point to integer Пересылка знакового бита каждого из четырех упакованных чисел входного операнда в младшие четыре бита выходного операнда. В качестве входного операнда может выступать только XMM-регистр, а в качестве выходного операнда — 32-разрядный регистр общего назначения. Эта команда используется для организации условных переходов. Пример: 
MOVSS	MOVe Scalar Single precision float point Пересылка 32 младших битов из источника в приемник, при этом как минимум один из операндов должен быть XMM-регистром. Второй операнд должен быть 32-разрядной ячейкой памяти. При выполне-

	нии операции пересылки из операнда в памяти младшие 32 бит помещаются в младшие 32 бит XMM-регистра. Если выполняется пересылка из XMM-регистра, то выбираются младшие 32 разряда регистра, остальные разряды не изменяются.

Арифметика. В мнемонике суффикс:

- SS – скалярный контекст.
- PS – параллельный контекст.

Мнемоника	Описание
ADDSS ADDPS	Сложение ADDition Scalar Single precision float point ADDition Packed Single precision float point
SUBSS SUBPS	Вычитание SUBtracition Scalar Single precision float point SUBtraction Packed Single precision float point
MULSS MULPS	Умножение MULtiply Scalar Single precision float point MULtiply Packed Single precision float point
DIVSS DIVPS	Деление DIVide Scalar Single precision float point DIVide Packed Single precision float point
SQRTSS SQRTPS	Извлечение квадратного корня Square RooT Scalar Single precision float point Square RooT Packed Single precision float point
MAXSS MAXPS	Максимальное значение MAXimum Scalar Single precision float point MAXimum Packed Single precision float point
MINSS MINPS	Минимальное значение MINimum Scalar Single precision float point MINimum Packed Single precision float point -
RCPSS RCPPS	Обратное значение ReCiProcal Scalar Single precision float point ReCiProcal Packed Single precision float point
	Обратное значение квадратного корня

RSQRTSS RSQRTPS	Reiprocal SQuare RooT Scalar Single precision float point Reiprocal SQuare RooT Packed Single precision float point

Сравнения. В мнемонике суффикс:

- SS – скалярный контекст.
- PS – параллельный контекст.

Мнемоника	Описание
CMPEQS CMPEQPS	Равенство (equal) CoMPare EQual Scalar Single precision float point CoMPare EQual Packed Single precision float point
CMPNEQS CMPNEQPS	Неравенство (not equal) CoMPare Not EQual Scalar Single precision float point CoMPare Not EQual Packed Single precision float point -
CMPLTS CMPLTPS	Меньше чем (less-than) CoMPare Less-Then Scalar Single precision float point CoMPare Less-Than Packed Single precision float point
CMPNLTS п CMPNLTPS	Не меньше чем (not-less-than) CoMPare Not Less-Then Scalar Single precision float point CoMPare Not Less-Than Packed Single precision float point
CMPLES CMPLEPS	Меньше или равно (less-than-or-equal) CoMPare Less-then-or-Equal Scalar Single precision float point CoMPare Less-than-or-Equal Packed Single precision float point
CMPNLES CMPNLEPS	Не меньше или равно (not less-than-or-equal) CoMPare Not Less-then-or-Equal Scalar Single precision float point CoMPare Not Less-than-or-Equal Packed Single precision float point
CMPORDS CMPORDPS	Упорядоченность (order) CoMPare ORDer Scalar Single precision float point CoMPare ORDer Packed Single precision float point
CMPUNORDS CMPUNORDPS	Неупорядоченность (unorder) CoMPare UNORDer Scalar Single precision float point CoMPare UNORDer Packed Single precision float point

Преобразования. В мнемонике суффикс:

- SI – скалярный контекст.
- PI – параллельный контекст.

Мнемоника	Описание
CVTSS2SI CVTPS2PI	Преобразование 32-разрядного числа с плавающей точкой в коротком формате в 32-разрядное целое число ConVerT Scalar Single precision float point to Scalar signed Int32 ConVersion Two Packed Single precision float point to Packed signed Int32
CVTTSS2SI CVTTPS2PI	Преобразование 32-разрядного числа с плавающей точкой в коротком формате в 32-разрядное целое число путем отсечения дробной части ConVerT Trancate Scalar Single precision float point to Scalar signed Int32 ConVersion Trancate Two Packed Single precision float point to Packed signed Int32
CVTSI2SS CVTTPS2PI	Преобразование 32-разрядного числа в целочисленном формате в 32-разрядное число с плавающей точкой в коротком формате. ConVerT Scalar signed Int32 to Scalar Single precision float point ConVersion Two Packed signed Int32 to Packed Single precision float point

Логика. Логические MMX-команды выполняют поразрядные логические операции над всеми 64 битами своих операндов. Они реализуют логические операции И, ИЛИ, И-НЕ, исключающего ИЛИ:

Мнемоника	Описание
ANDPS	bit-wise logical AND for Packed Single precision float point Параллельная операция логического И над парами битов упакованных чисел с плавающей точкой операнда-источника и операнда-приемника.
ANDNPS	bit-wise logical AND-Not for Packed Single precision float point Параллельная операция логического И-НЕ над парами битов упакованных чисел с плавающей точкой операнда-источника и операнда-приемника
ORPS	bit-wise logical OR for Packed Single precision float point

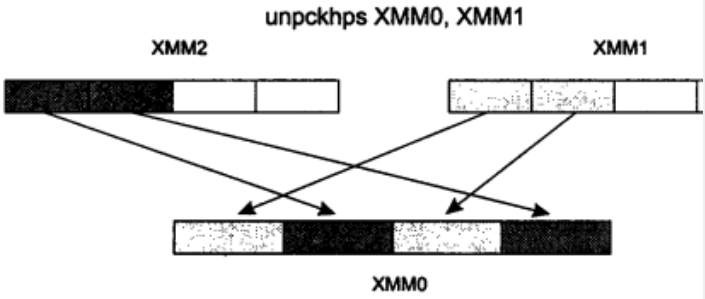
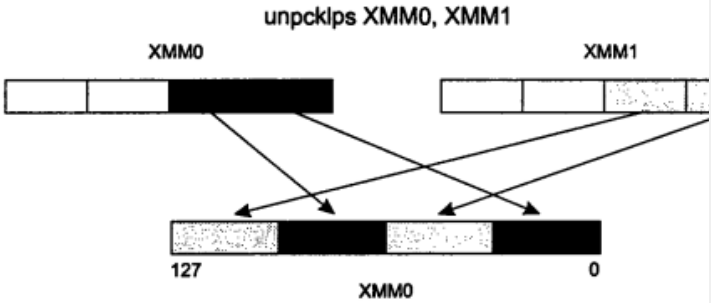
	Параллельная операция логического ИЛИ над парами битов упакованных чисел с плавающей точкой операнда-источника и операнда-приемника
XORPS	bit-wise logical eXclusive OR for Packed Single precision float point Параллельная операция логического Исключающкго ИЛИ над парами битов упакованных чисел с плавающей точкой операнда-источника и операнда-приемника.

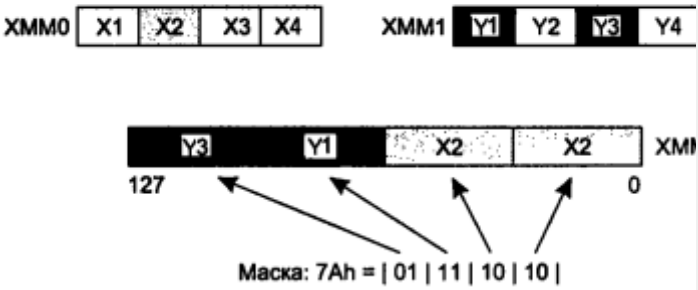
Управление состоянием. К этой группе относятся команды, выполняющие загрузку/сохранение регистров состояния и управления:

Мнемоника	Описание
LDMXCSR источник	LoaD MXCSR Загрузка регистра управления/состояния (MXCSR) содержимым 32-разрядной ячейки памяти, которая и является единственным операндом.
STMXCSR приемник	STore MXCSR Сохранение регистра управления/состояния (MXCSR) в 32-разрядной ячейки памяти, которая и является единственным операндом.
FXRSTOR источник	Fp and mmX ReSTORe Загрузка предварительно сохраненного состояния сопроцессора, MMX- и SEE-расширения из области памяти размером 512 байт. В качестве операнда выступает адрес области памяти, который должен быть выровнен по 16-байтовой границе.
FXSAVE приемник	Fp and mmX SAVE Сохранение состояния сопроцессора, MMX- и SSE-расширения в область памяти размером в 512 байт. В качестве операнда выступает адрес области памяти.

Распаковка данных. XMM-команды распаковки попарно объединяют элементы данных из обоих операндов в более длинные элементы выходного операнда. Этими командами можно пользоваться для увеличения числа значащих разрядов при вычислениях.

Мнемони-	Описание

ка	
UNPCKHPS	UNPaCK High Packed Single precision float point data Параллельное перемещение старших двойных слов из операнда-источника и операнда-приемника в операнд-приемник.Пример: 
UNPCKLPS	UNPaCK Low Packed Single precision float point data Параллельное перемещение младших двойных слов из операнда-источника и операнда-приемника в операнд-приемник. Пример: 

SHUFPS	UNPaCK Low Packed Single precision float point data Параллельная перестановка 32-разрядных упакованных операндов в соответствии с заданной маской. Команда имеет три операнда: входной, выходной и операнд-маску. Маска представляет собой непосредственное 8-разрядное значение, задающее порядок перестановки операндов. Каждая пара битов маски определяет номер упакованного 32-разрядного операнда в приемнике или источнике, который должен помещаться в операнд-приемник. При этом порядок размещения 32-разрядных операндов таков: младшие 4 бита маски указывают номера двух упакованных чисел приемника, которые становятся младшими упакованными значениями результата, а старшие 4 бита — номера упакованных чисел источника, которые становятся старшими упакованными значениями результата. Пример: **shufps XMM0, XMM1, 7Ah** 

Управление кэшированием. Необходимость управления кэшированием вызвана тем, что большинство мультимедийных приложений оперируют большими объемами данных, при этом может случиться, что данные, загруженные в кэш, никогда не понадобятся. Чтобы оптимизировать работу кэша, в систему команд SSE-расширения и были включены команды управления кэшем.

Мнемоника	Описание
MASKMOVQ mm1, mm2	MASK MOVe Quadword Выборочная запись байтов в память из источника (MMX-регистр mm1) по маске. Маска – это знаковые биты 8 байтов MMX-

	регистра mm2. Последовательно проверяются знаковые биты всех 8 байтов, и если какой-то из этих битов равен 1, то соответствующий байт источника копируется в область памяти, начальный адрес которой содержится в регистрах DS:DI/EDI. Если знаковый бит равен нулю, то соответствующий байт в приемнике будет нулевым.
MOVNTQ	MOVe using Non-Temporal of Quadword Сохранение учетверенного слова из XMM-регистра в памяти без использования кэша.
MOVNTDQ	MOVe using Non-Temporal of Double Quadword Сохранение двойного учетверенного слова из XMM-регистра в памяти без использования кэша.
MOVNTPD	MOVe Non Temporal aligned Packed Double precision float point data Запись в память, минуя кэш, упакованных чисел с плавающей точкой с удвоенной точностью. Операндом-источником здесь выступает XMM-регистр, а операндом-приемником — 128-разрядная ячейка памяти, адрес которой должен быть выровнен по 16-байтовой границе.
MOVNTPS	MOVe Non-Temporal Packed Single precision float point Запись в память, минуя кэш, упакованных чисел с плавающей точкой в коротком формате. Операндом-источником здесь выступает XMM-регистр, а операндом-приемником — 128-разрядная ячейка памяти, адрес которой должен быть выровнен по 16-байтовой границе.

Дополнительные команды. Сейчас мы рассмотрим еще одну группу команд, которые трудно отнести к какому-либо определенному типу, но которые являются весьма полезными при разработке программ.

Перераспределение.

Мнемоника	Описание
SHUFPS приемник, источник	Packed AveraGe Bytes Попарно вычисляет средние значения упакованных чисел, представленных байтами. Значения операндов интерпретируются как беззнаковые целые числа. В качестве источника могут выступать MMX-регистр

	или 64-разрядная ячейка памяти, приемником служит один из MMX-регистров.
UNPCKHPS приемник, источник	Packed AveraGe Bytes Попарно вычисляет средние значения упакованных чисел, представленных байтами. Значения операндов интерпретируются как беззнаковые целые числа. В качестве источника могут выступать MMX-регистр или 64-разрядная ячейка памяти, приемником служит один из MMX-регистров.
UNPCKLPS приемник, источник	Packed AveraGe Bytes Попарно вычисляет средние значения упакованных чисел, представленных байтами. Значения операндов интерпретируются как беззнаковые целые числа. В качестве источника могут выступать MMX-регистр или 64-разрядная ячейка памяти, приемником служит один из MMX-регистров.

Перераспределение.

Команда	Описание
SHUFPS приемник, источник	Packed AveraGe Bytes Попарно вычисляет средние значения упакованных чисел, представленных байтами. Значения операндов интерпретируются как беззнаковые целые числа. В качестве источника могут выступать MMX-регистр или 64-разрядная ячейка памяти, приемником служит один из MMX-регистров.
UNPCKHPS приемник, источник	Packed AveraGe Bytes Попарно вычисляет средние значения упакованных чисел, представленных байтами. Значения операндов интерпретируются как беззнаковые целые числа. В качестве источника могут выступать MMX-регистр или 64-разрядная ячейка памяти, приемником служит один из MMX-регистров.
UNPCKLPS приемник, источник	Packed AveraGe Bytes Попарно вычисляет средние значения упакованных чисел, представленных байтами. Значения операн-

	дов интерпретируются как беззнаковые целые числа. В качестве источника могут выступать MMX-регистр или 64-разрядная ячейка памяти, приемником служит один из MMX-регистров.

Управление состоянием.

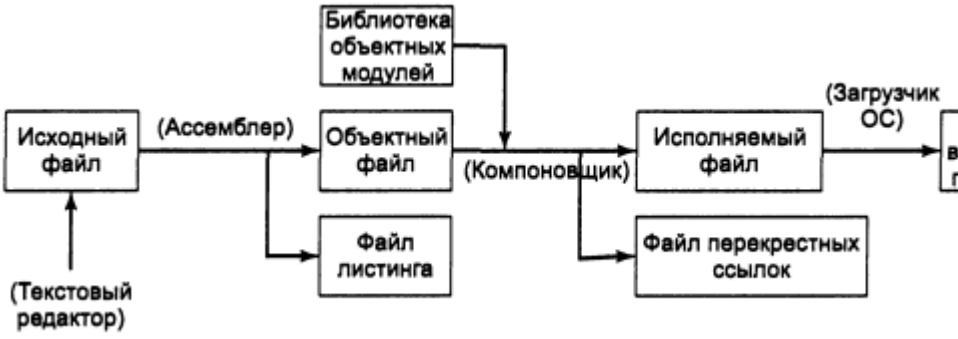
Команда	Описание
IDMXCSR приемник, источник	Packed AveraGe Bytes Попарно вычисляет средние значения упакованных чисел, представленных байтами. Значения операндов интерпретируются как беззнаковые целые числа. В качестве источника могут выступать MMX-регистр или 64-разрядная ячейка памяти, приемником служит один из MMX-регистров.
STMXCSR приемник, источник	Packed AveraGe Bytes Попарно вычисляет средние значения упакованных чисел, представленных байтами. Значения операндов интерпретируются как беззнаковые целые числа. В качестве источника могут выступать MMX-регистр или 64-разрядная ячейка памяти, приемником служит один из MMX-регистров.
FXSTOR приемник, источник	Packed AveraGe Bytes Попарно вычисляет средние значения упакованных чисел, представленных байтами. Значения операндов интерпретируются как беззнаковые целые числа. В качестве источника могут выступать MMX-регистр или 64-разрядная ячейка памяти, приемником служит один из MMX-регистров.
FXSAFE приемник, источник	Packed AveraGe Bytes Попарно вычисляет средние значения упакованных чисел, представленных байтами. Значения операндов интерпретируются как беззнаковые целые числа. В качестве источника могут выступать MMX-регистр или 64-разрядная ячейка памяти, приемником служит один из MMX-регистров.

Управление кэшированием

Команда	Описание
MASKMOVQ приемник, источник	Packed AveraGe Bytes Попарно вычисляет средние значения упакованных чисел, представленных байтами. Значения операндов интерпретируются как беззнаковые целые числа. В качестве источника могут выступать MMX-регистр или 64-разрядная ячейка памяти, приемником служит один из MMX-регистров.
MOVNTG приемник, источник	Packed AveraGe Bytes Попарно вычисляет средние значения упакованных чисел, представленных байтами. Значения операндов интерпретируются как беззнаковые целые числа. В качестве источника могут выступать MMX-регистр или 64-разрядная ячейка памяти, приемником служит один из MMX-регистров.

3.9. Цикл трансляции, компоновки и выполнения

Процесс редактирования исходного ассемблерного файла (т.е. написания программы), его компиляции, компоновки и выполнения схематически показан на рисунке.



Ниже приведено подробное описание каждого этапа.

1. С помощью текстового редактора программист создает исходный текстовый файл {source file), содержащий программу на ассемблере.
2. На вход программы ассемблера подается исходный файл, а на выходе получается объектный файл, содержащий машинный код. В качестве дополнительной возможности, ассемблер может создать файя листинга {listing file) программы. Если при компиляции возникнут ошибки, программист должен вернуться к п. 1 и устранить причину их появления. Файл листинга программы предназначен, в основном, для получения твердой копии программы принтере. Поэтому, кроме текста самой программы, разбитого на страницы, в нем содержатся номера строк, адреса команд (точнее, их смешений относительно сегмента кода), оттранслированный машинный код, представленный в шестнадцатеричном виде, и таблица символов.
3. Содержимое объектного файла анализируется компоновщиком. Он определяет, есть ли в программе так называемые внешние ссылки, т.е. содержит ли программа команды вызова процедур, находящихся в одной из библиотек объектных модулей (link library). Компоновщик находит эти ссылки в объектном файле программы, копирует необходимые процедуры из библиотек, объединяет их вместе с объектным файлом (этот процесс называется разрешением внешних ссылок) и создает исполняемый файл (executable flle). В качестве дополнительной возможности компоновщик может создать файл перекрестных ссылок {mapfile), содержащий план полученного исполняемого файла.
4. Компонент операционной системы, называемый загрузчиком (loader), считывает данные из исполняемого файла, загружает программу в память и передает управление по адресу точки входа. В результате программа начинает выполняться.

Файлы, создаваемые и модифицируемые компоновщиком.

Файл перекрестных ссылок. Это обычный текстовый файл, имеющий расширение .MAP, в котором содержится информация о сегментах, содержащихся в компонуемой программе, а также следующие данные.

- Имя исполняемого модуля, которое представляет собой базовое имя (т.е. без расширения) исходного ASM-файла.
- Дата и время, полученные из заголовка исполняемого файла (а не из элемента каталога файловой системы).
- Список сегментов программы, упорядоченный по группам. Для каждой группы указывается начальный адрес, длина, имя группы и класс.

- Список глобальных (public) символов с указанием для каждого символа его адреса, имени, линейного адреса и имени модуля, где определен этот символ.
- Адрес точки входа в программу.

Файл базы данных программы. Если при запуске ассемблера указать в командной строке ключ -Zi (он задает режим отладки), MASM создаст специальный файл базы данныхпрограммы (j)rogram databasefik) с расширением .PDB. На этапе компоновки редактор связей считывает информацию из PDB-файла и обновляет ее. Если после этого загрузить программу в отладчик, тот сможет показать вам в своем окне

3.10. Ассемблер CISC

3.10.1. Введение

Когда-то ассемблер был языком, без знания которого нельзя было заставить компьютер сделать что-либо полезное. Постепенно ситуация менялась. Появлялись более удобные средства общения с компьютером. Но, в отличие от других языков, ассемблер не умирал, более того он не мог сделать этого в принципе. Почему? В поисках ответа попытаемся понять, что такое язык ассемблера вообще.

Если коротко, то язык ассемблера — это символическое представление машинного языка. Все процессы в машине на самом низком, аппаратном уровне приводятся в действие только командами (инструкциями) машинного языка. Отсюда понятно, что, несмотря на общее название, язык ассемблера для каждого типа компьютера свой. Это касается и внешнего вида программ, написанных на ассемблере, и идей, отражением которых этот язык является.

По-настоящему решить проблемы, связанные с аппаратурой (или даже, более того, зависящие от аппаратуры как, к примеру, повышение быстродействия программы), невозможно без знания ассемблера.

Программист или любой другой пользователь может использовать любые высокоуровневые средства, вплоть до программ построения виртуальных миров и, возможно, даже не подозревать, что на самом деле компьютер выполняет не команды языка, на котором написана его программа, а их трансформированное представление в форме скучной и унылой последовательности команд совсем другого языка — машинного. А теперь представим, что у такого пользователя возникла нестандартная проблема или просто что-то не заладилось. К примеру, его программа должна работать с некоторым необычным устройством или вы-

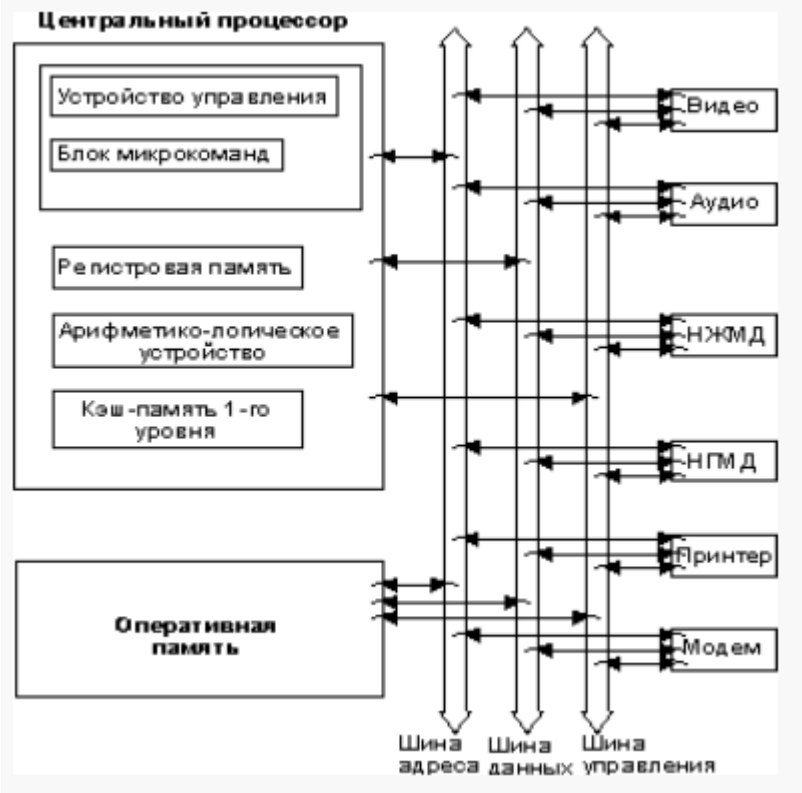
полнять другие действия, требующие знания принципов работы аппаратуры компьютера. И вот здесь-то и начинается совсем другая история.... Каким бы умным ни был программист, каким бы хорошим ни был язык, на котором он написал свою чудную программу, без знания ассемблера ему не обойтись. И не случайно практически все компиляторы языков высокого уровня содержат средства связи своих модулей с модулями на ассемблере либо поддерживают выход на ассемблерный уровень программирования.

Конечно, время компьютерных универсалов уже прошло. Как говорится нельзя объять необъятное. Но есть нечто общее, своего рода фундамент, на котором строится любое сколь-нибудь серьезное компьютерное образование. Это знания о принципах работы компьютера, его архитектуре и языке ассемблера как отражении и воплощении этих знаний.

Типичный современный компьютер (на базе i486 или Pentium) состоит из следующих компонентов (рис. 1).

Из рисунка видно, что компьютер составлен из нескольких физических устройств, каждое из которых подключено к одному блоку, называемому системным. Если рассуждать логически, то ясно, что он играет роль некоторого координирующего устройства. Давайте заглянем внутрь системного блока (не нужно пытаться проникнуть внутрь монитора — там нет ничего интересного, к тому же это опасно): открываем корпус и видим какие-то платы, блоки, соединительные

провода. Чтобы понять их функциональное назначение, посмотрим на структурную схему типичного компьютера (рис. 2). Она не претендует на безусловную точность и имеет целью лишь показать назначение, взаимосвязь и типовой состав элементов современного персонального компьютера.



Обсудим схему на рис. 2 в несколько нетрадиционном стиле. Человеку свойственно, встречаясь с чем-то новым, искать какие-то ассоциации, которые могут помочь ему познать неизвестное. Какие ассоциации вызывает компьютер? У меня, к примеру, компьютер часто ассоциируется с самим человеком. Почему?

У компьютера есть органы восприятия информации из внешнего мира — это клавиатура, мышь, накопители на магнитных дисках. На рис. 2 эти органы расположены справа от системных шин. У компьютера есть органы “перевариваю-

щие" полученную информацию — это центральный процессор и оперативная память.И, наконец, у компьютера есть органы речи, выдающие результаты переработки. Это также некоторые из устройств справа.

Современным компьютерам, конечно, далеко до человека. Их можно сравнить с существами, взаимодействующими с внешним миром на уровне большого, но ограниченного набора безусловных рефлексов. Этот набор рефлексов образует систему машинных команд. На каком бы высоком уровне вы не общались с компьютером, в конечном итоге все сводится к скучной и однообразной последовательности машинных команд. Каждая машинная команда является своего рода раздражителем для возбуждения того или иного безусловного рефлекса. Реакция на этот раздражитель всегда однозначная и "зашита" в блоке микрокоманд в виде микропрограммы. Эта микропрограмма и реализует действия по реализации машинной команды, но уже на уровне сигналов, подаваемых на те или иные логические схемы компьютера, тем самым управляя различными подсистемами компьютера. В этом состоит так называемый **принцип микропрограммного управления**.

Продолжая аналогию с человеком, отметим: для того, чтобы компьютер правильно питался, придумано множество операционных систем, компиляторов сотен языков программирования и т. д. Но все они являются, по сути, лишь блюдом, на котором по определенным правилам доставляется пища (программы) желудку (компьютеру). Только (вот досада!) желудок компьютера любит диетическую, однообразную пищу — подавай ему информацию структурированную, в виде строго организованных последовательностей нулей и единиц, комбинации которых и составляют машинный язык.

Таким образом, внешне являясь полиглотом, компьютер понимает только один язык — язык машинных команд. Конечно, для общения и работы с компьютером, необязательно знать этот язык, но практически любой профессиональный программист рано или поздно сталкивается с необходимостью его изучения. К счастью, программисту не нужно пытаться постичь значение различных комбинаций двоичных чисел, так как еще в 50-е годы программисты стали использовать для программирования символический аналог машинного языка, который назвали языком **ассемблера**. Этот язык точно отражает все особенности машинного языка. Именно поэтому, в отличие от языков высокого уровня, язык ассемблера для каждого типа компьютера свой.

Из всего вышесказанного можно сделать вывод, что, так как язык ассемблера для компьютера "родной", то и самая эффективная программа может быть написана только на нем (при условии, что ее пишет квалифицированный програм-

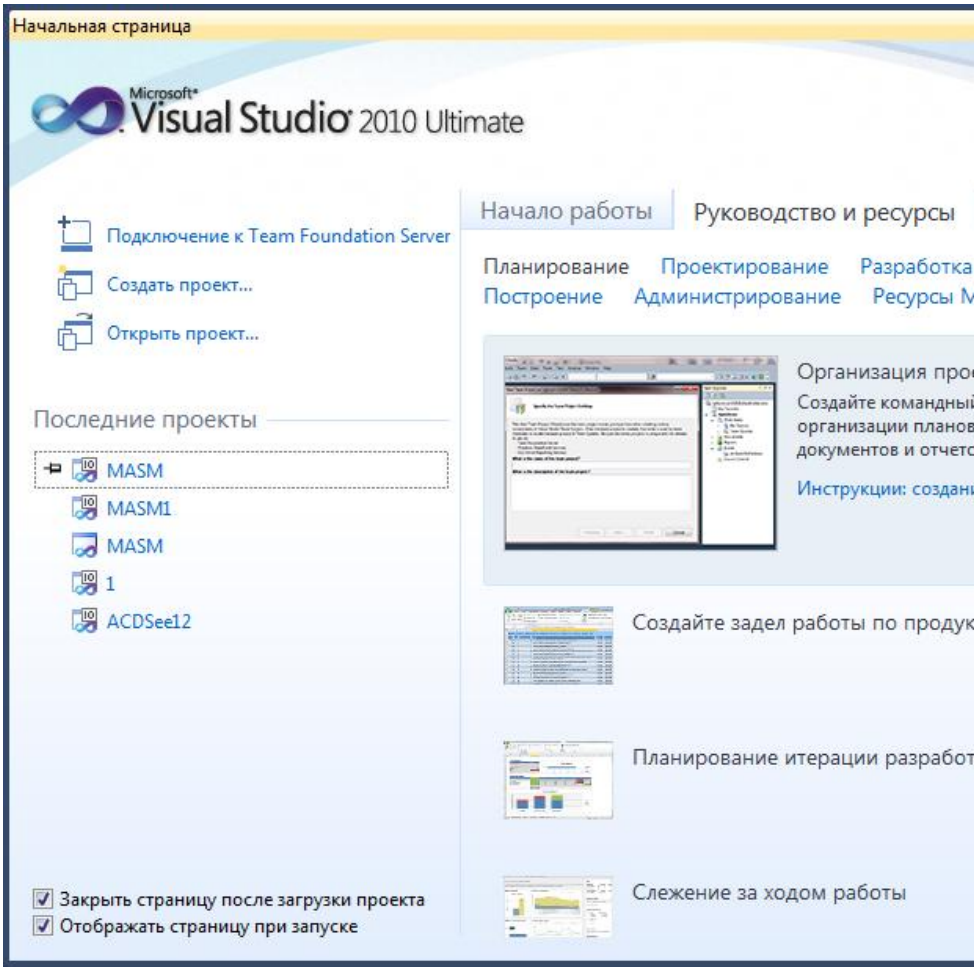
мист). Здесь есть одно маленькое “но”: это очень трудоемкий, требующий большого внимания и практического опыта процесс. Поэтому реально на ассемблере пишут в основном программы, которые должны обеспечить эффективную работу с аппаратной частью. Иногда на ассемблере пишутся критичные по времени выполнения или расходованию памяти участки программы. Впоследствии они оформляются в виде подпрограмм и совмещаются с кодом на языке высокого уровня.

3.10.2. Средства программирования и отладки

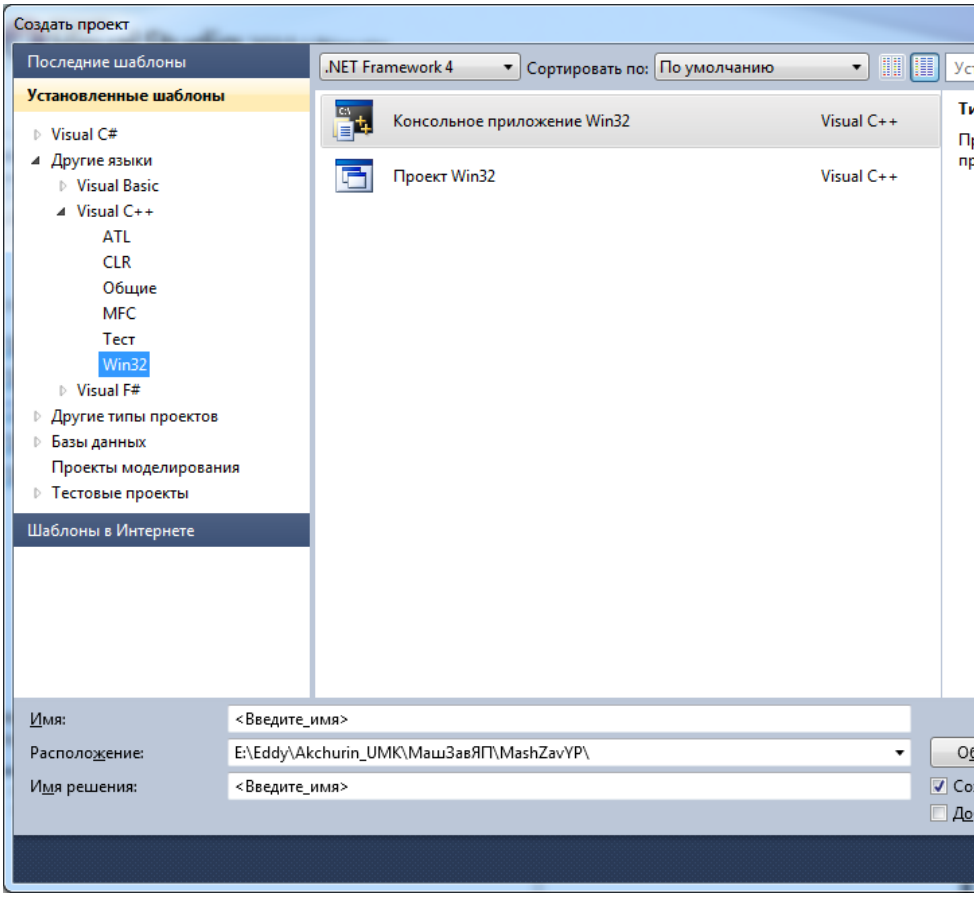
Традиционно для программирования на ассемблере использовались два комплекта программ:

- MASM – набор от компании Microsoft.
- TASM - набор от компании Borland.

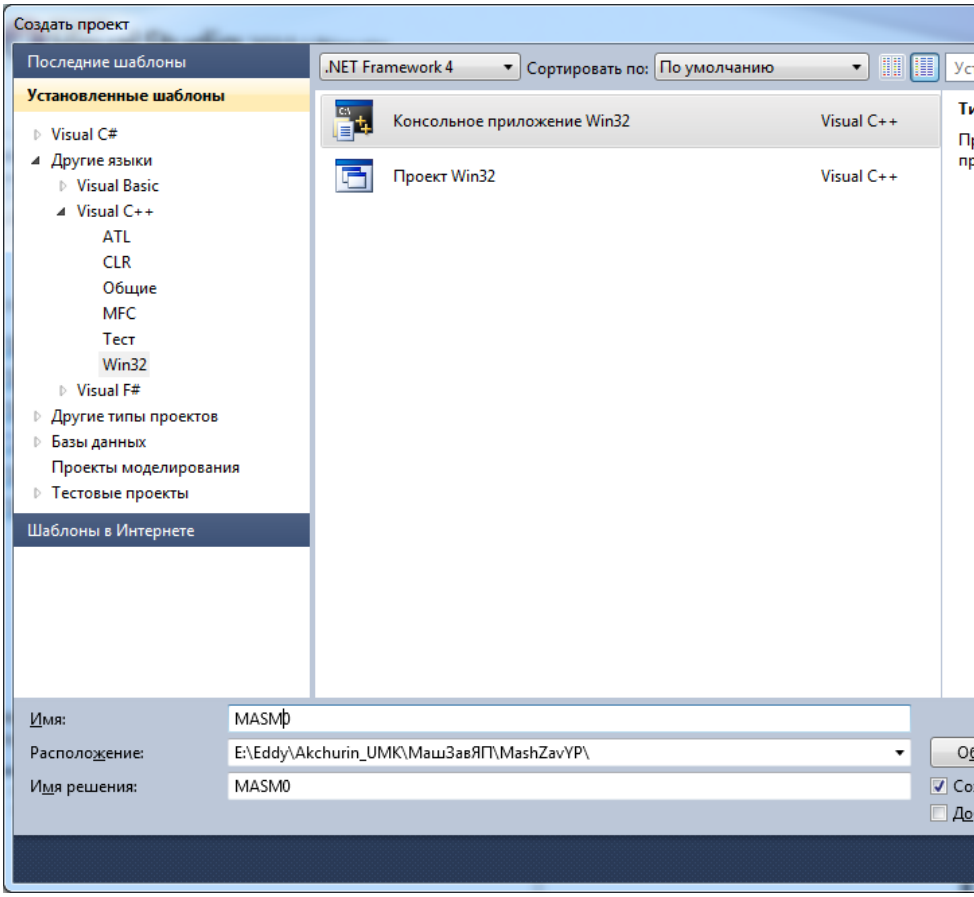
Оба набора в настоящее время не обновляются. В результате они не работают под управлением оперционной системы Windows 7. Рекоменуется испольовать ИСР Visual Studio 2010, в которой при работе с языком программирования Visual C++ можно использовать ассемблерные вставки в код C++. Стартовое окно ИСР:



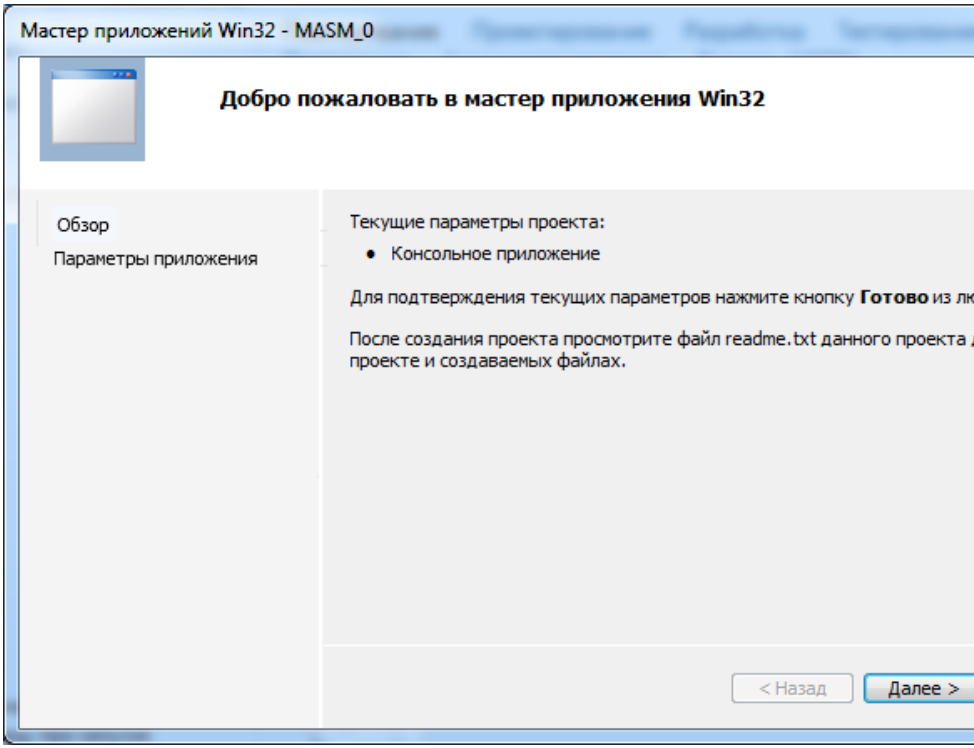
В нем выбираем команду Создать проект.



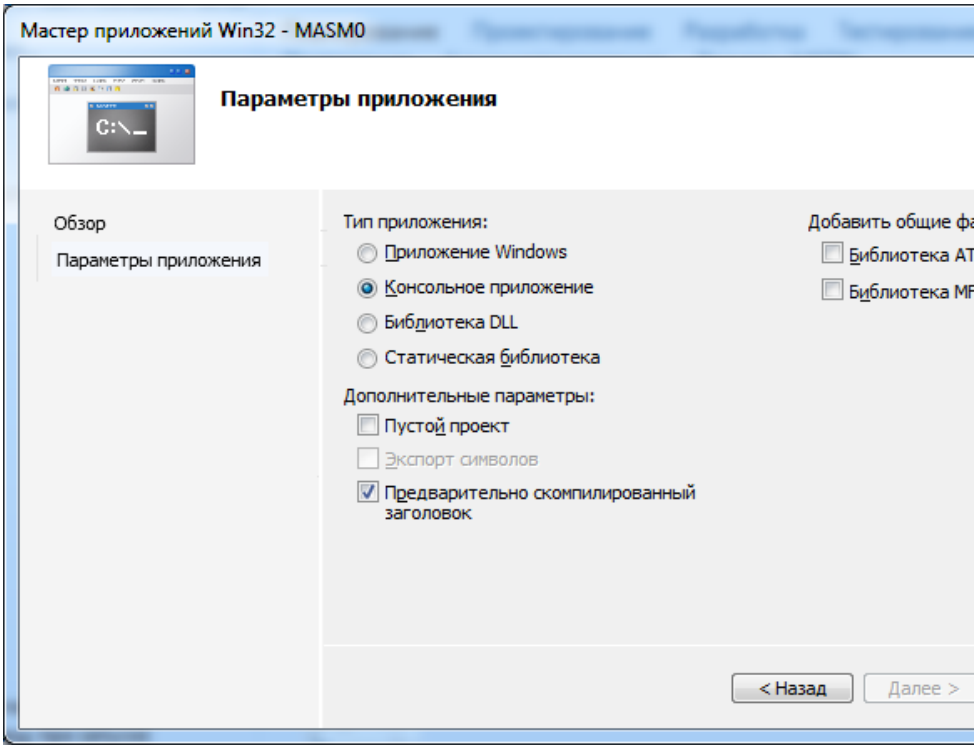
В разделе выбора языка выбираем Visual C++ И шаблон Win32. Из списка принимаем **Консольное приложение Win32**. Задаем имя проекта MASM1, выбираем место размещения файлов проекта. Устанавливаем флаг **Создать каталог для решения**.



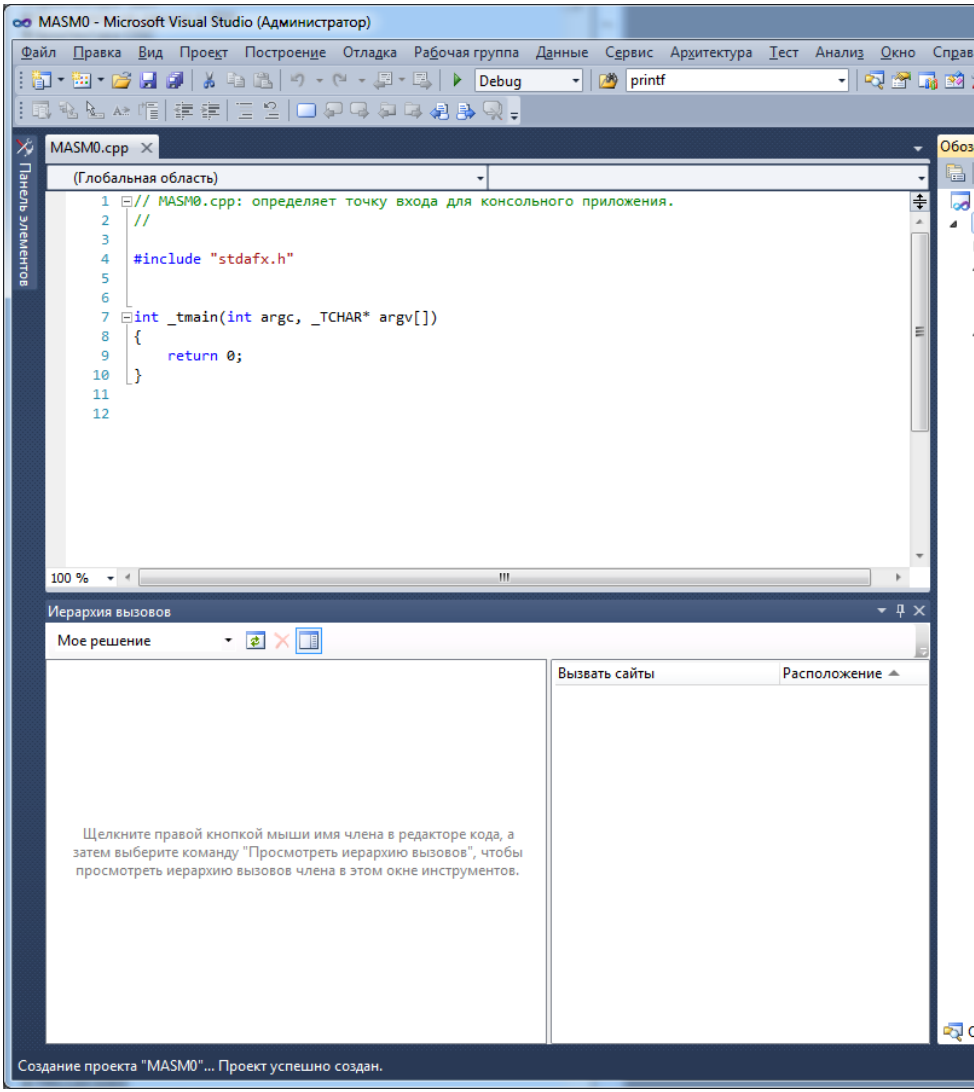
Далее работает мастер приложения:



Дополнительно выбираем предварительно мкомпилированный заголовок.



В итоге получаем шаблон проекта. В него можно добавить конкретные команды.

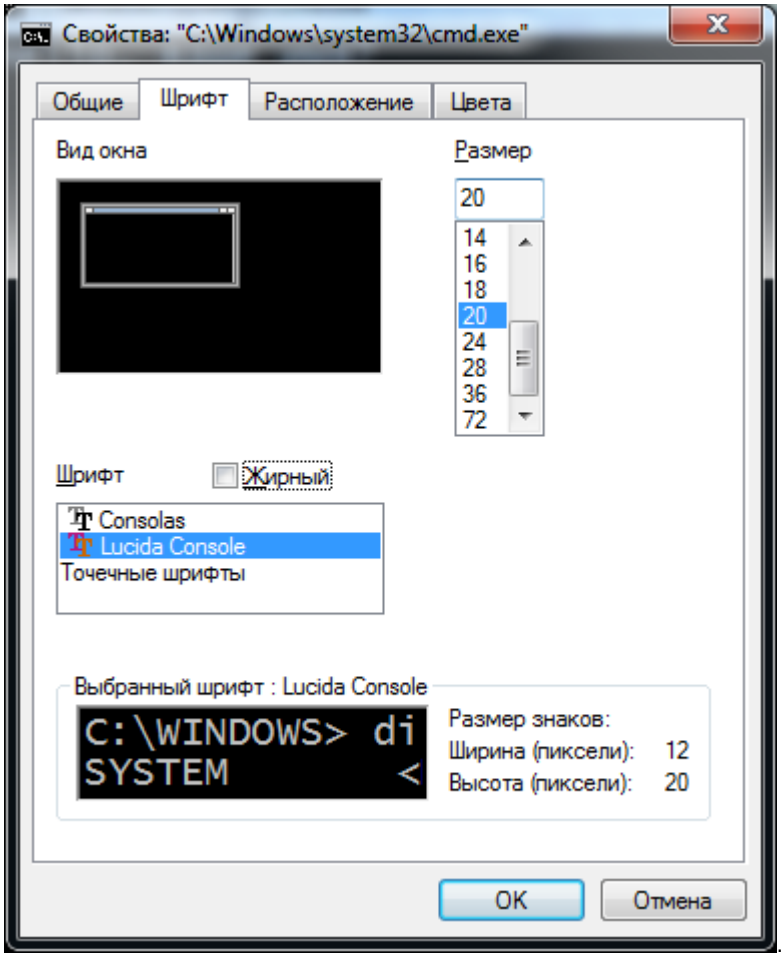


При запуске консольного приложения по умолчанию используется версия шрифта, не поддерживающая кириллицу , используемую в редакторе кода. Например, при выводе фразы «Hello, ваше имя?» в консоли получаем:

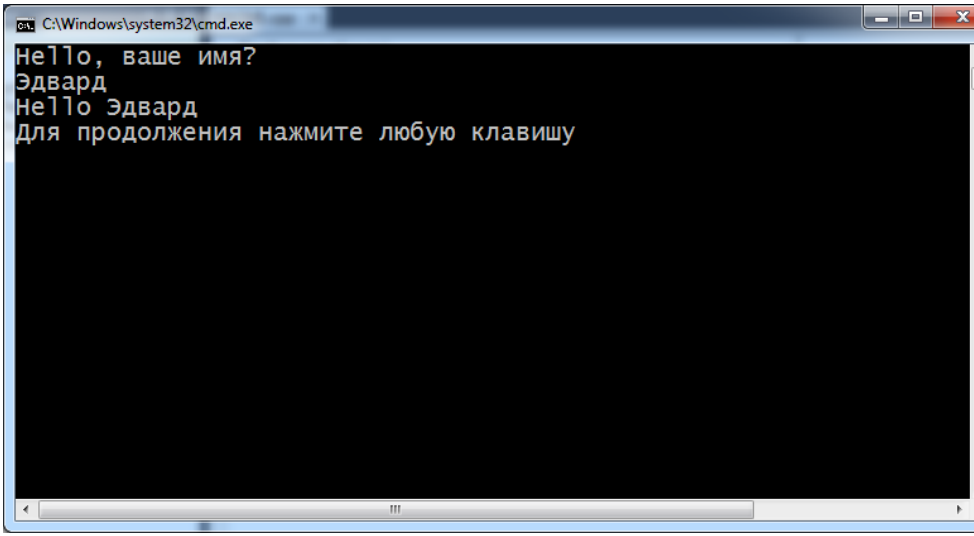
Для решения проблемы нужно:

- Включить файл #include <windows.h>
- В код программы нужно вставить уоманду выбора кодировки SetConsoleOutputCP(1251), которая определена в файле #include <windows.h>
- Настроить консоль.

Настройка консоли. Для этого запусить прогамму на выполнение. Щелчком правой кнопки мыши по заголовку консоли вызвать меню, в котором выбрать команду Свойства. Отображается окно настроек, в котором для шрифта выбрать такой же шрифт, что в редакторе – Lucida Console.

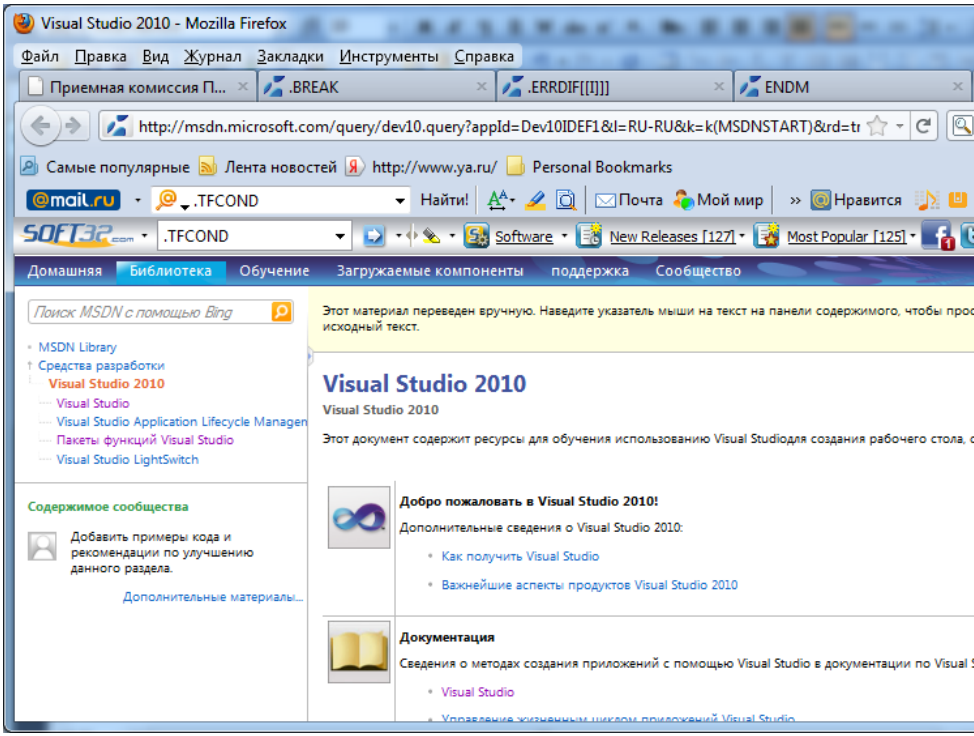


Теперь консоль правильная.

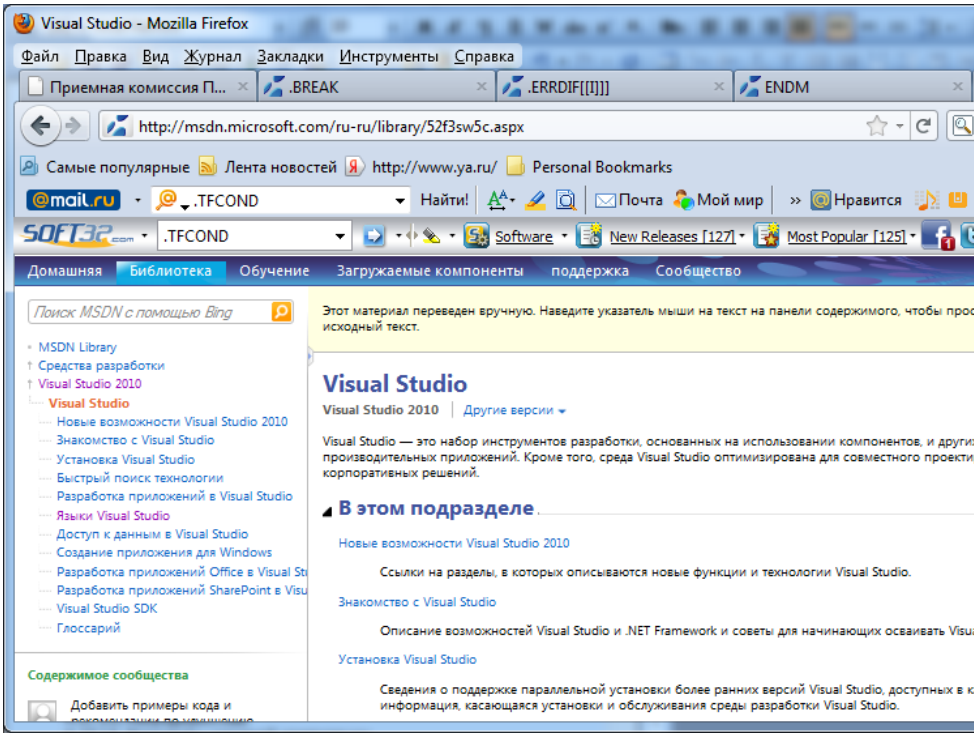


3.11. Описание MASM

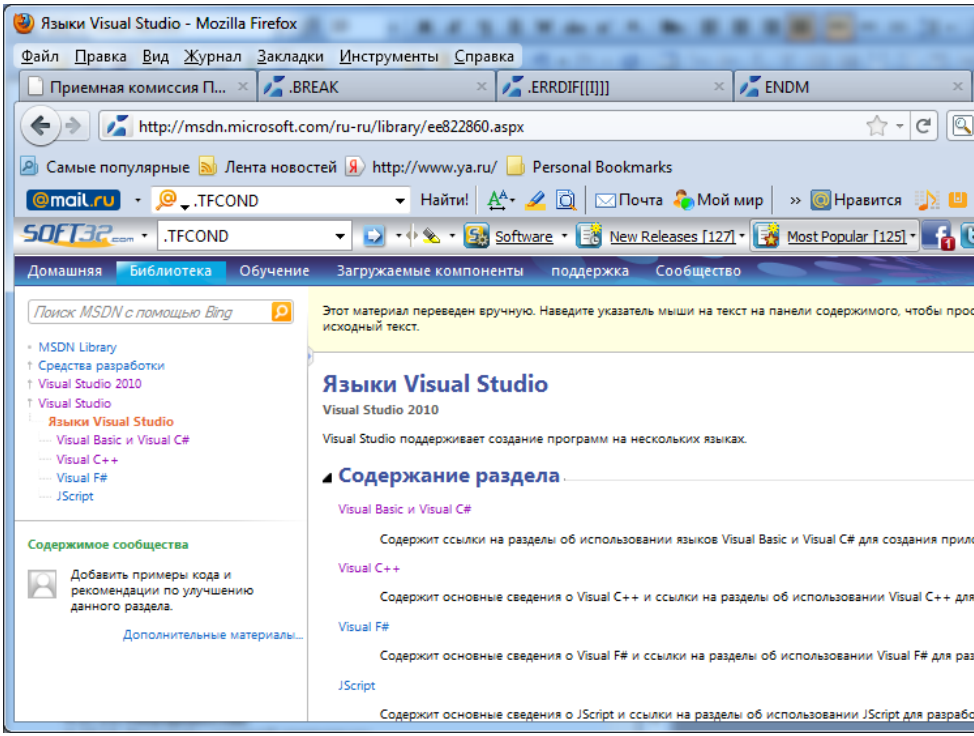
В ИСР имеется русифицированная справка, в которой можно найти необходимые сведения по языку ассемблера. Используем команду Справка=>Просмотр справки. В .браузере отображается окно доступа к справке в Интернет.



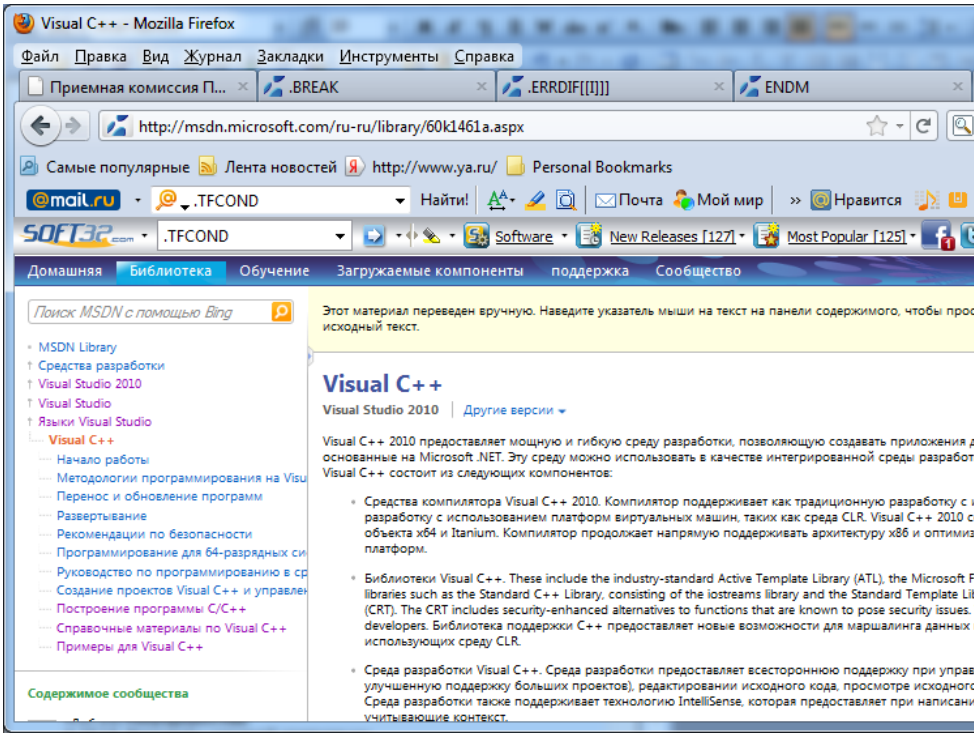
Выбираем Visual Studio 2010. Отображаются подробности.



В окне выбираем Языки Visual Studio. Отображаются подробности.



В окне выбираем Visual C++. Отображаются подробности.

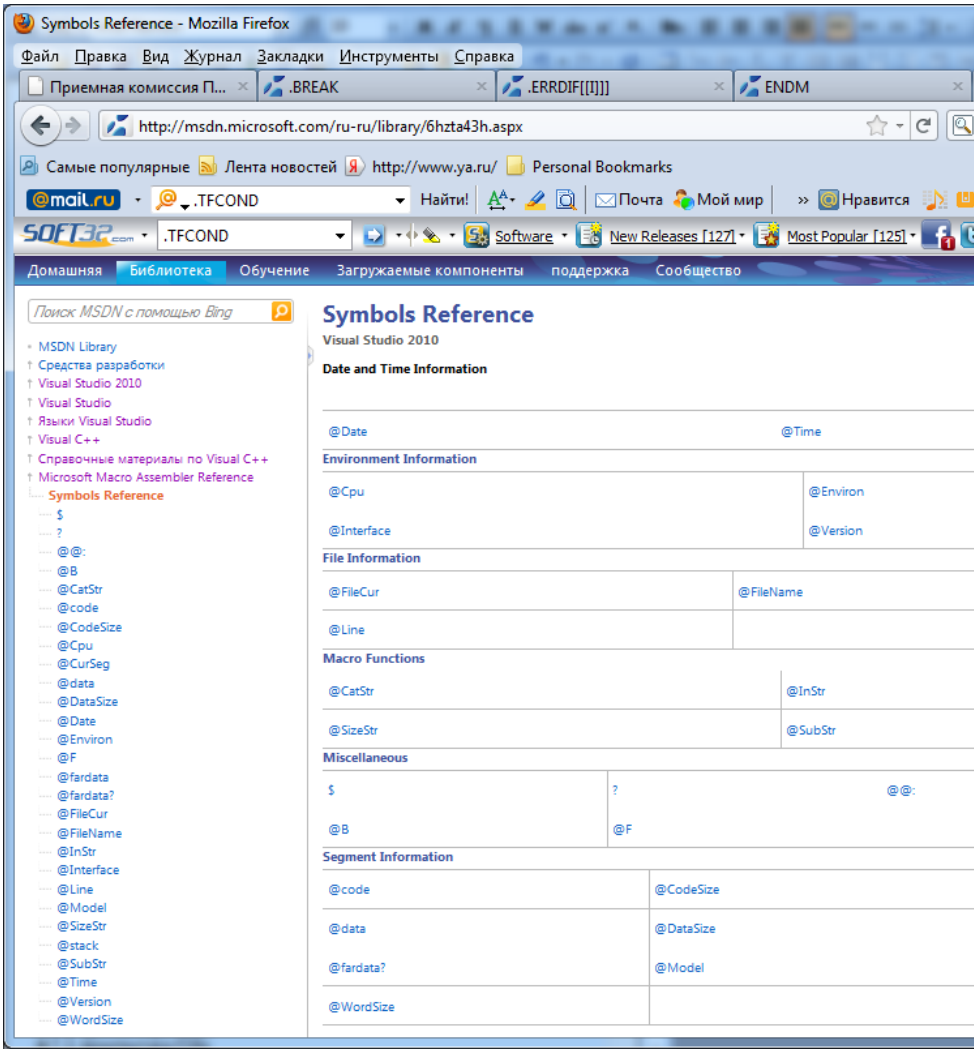


В окне выбираем Справочные материалы по Visual C++. Отображаются подробности.

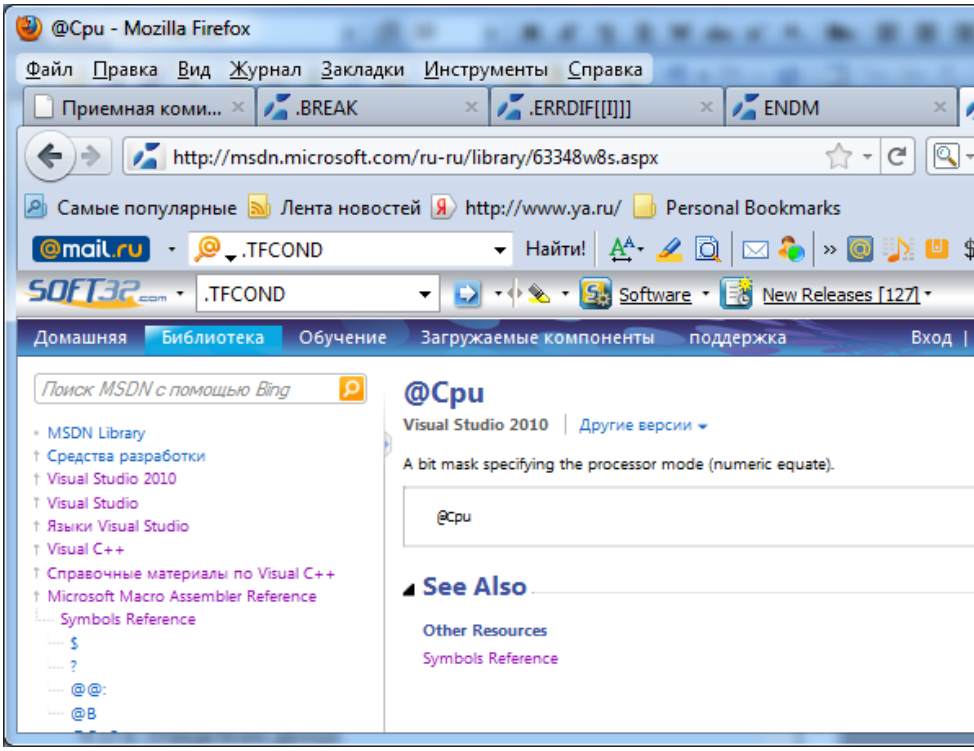
В окне выбираем Microsoft Macro Assembler Отображаются подробности. Теперь можно посмотреть материалы по разделам MASM64. К сожалению на июль 2011 подробная справка пока на английском языке. Доступны разделы:

- Символы
- Директивы
- Операторы

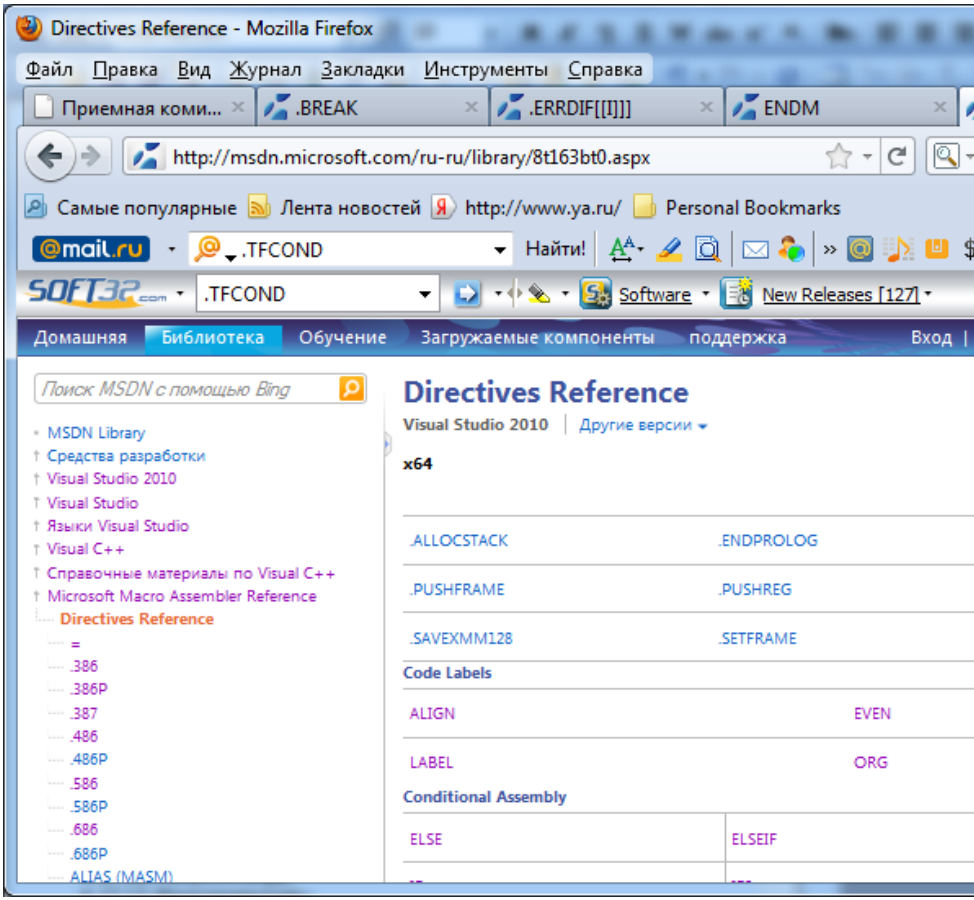
Посмотрим Символы. В левом поле список по алфавиту, в правом - по категориям.



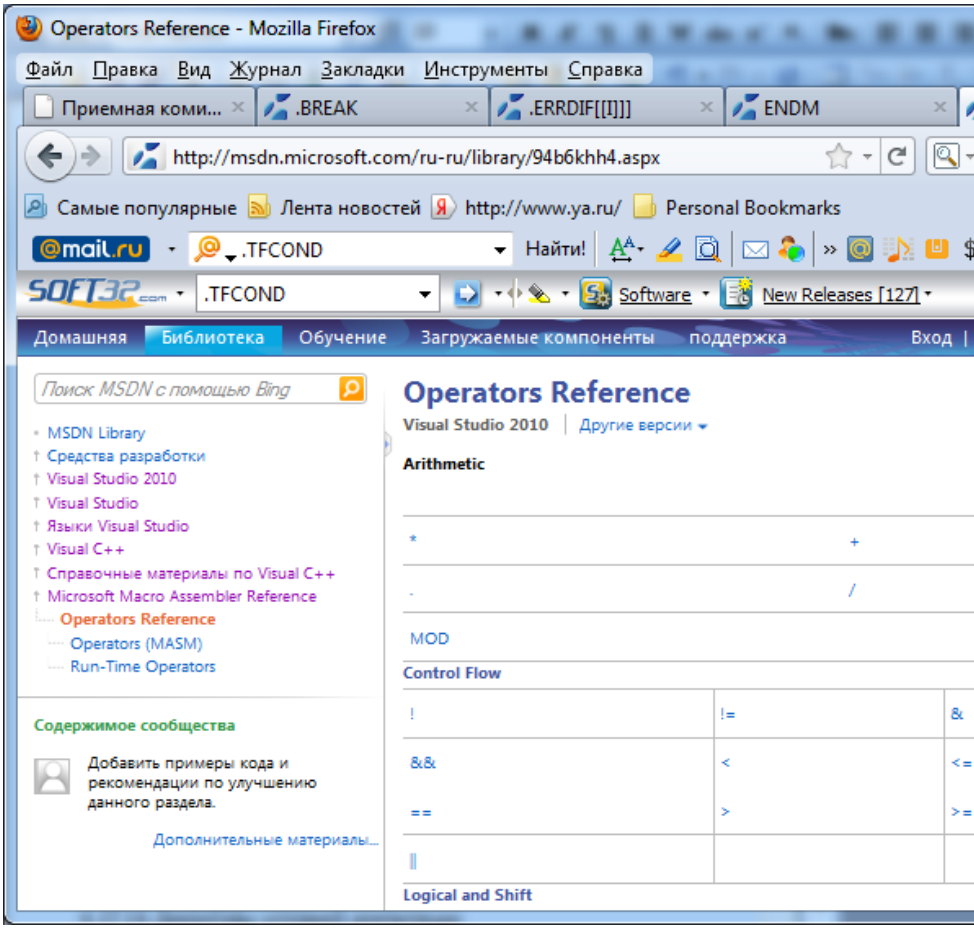
Справка по любому символу отображается в отдельном окне по гиперссылке. Например, для символа @cpu получаем. ссобщение, что это битовая маска, указывающая режим процессора.



Посмотрим Директивы. В левом поле список по алфавиту, в правом - по категориям.

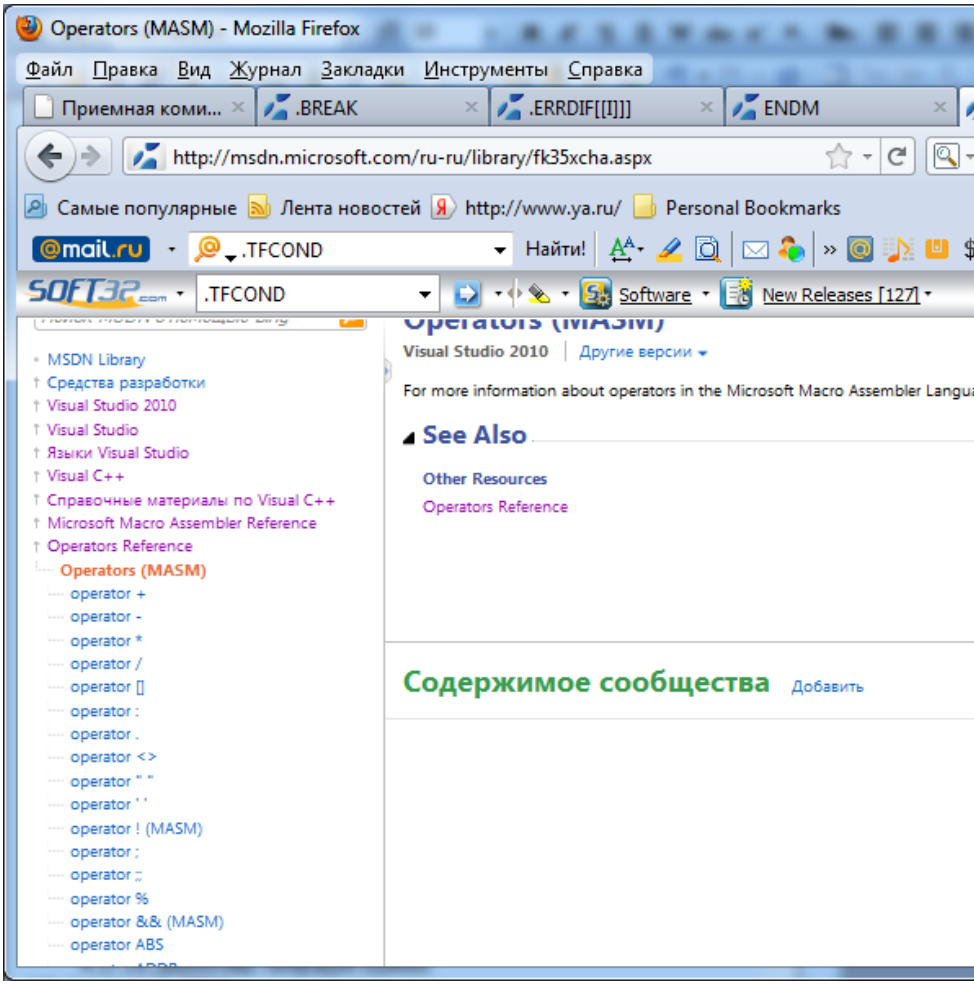


Посмотрим Операторы. В левом поле список по алфавиту, в правом - по категориям.

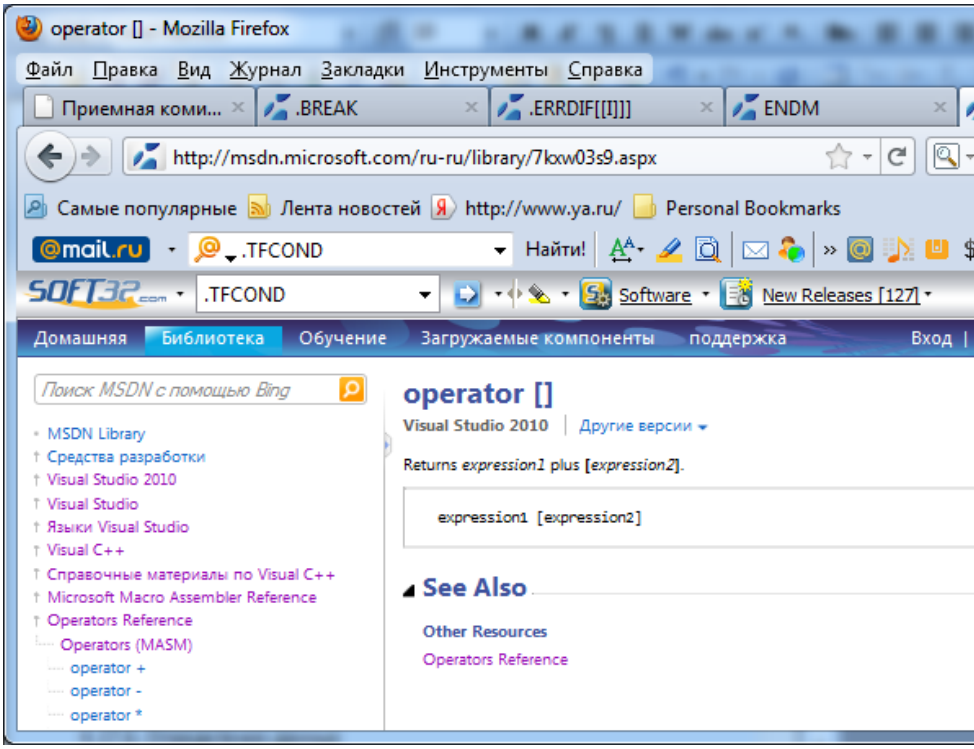


Различаются операторы MASM и времени исполнения.

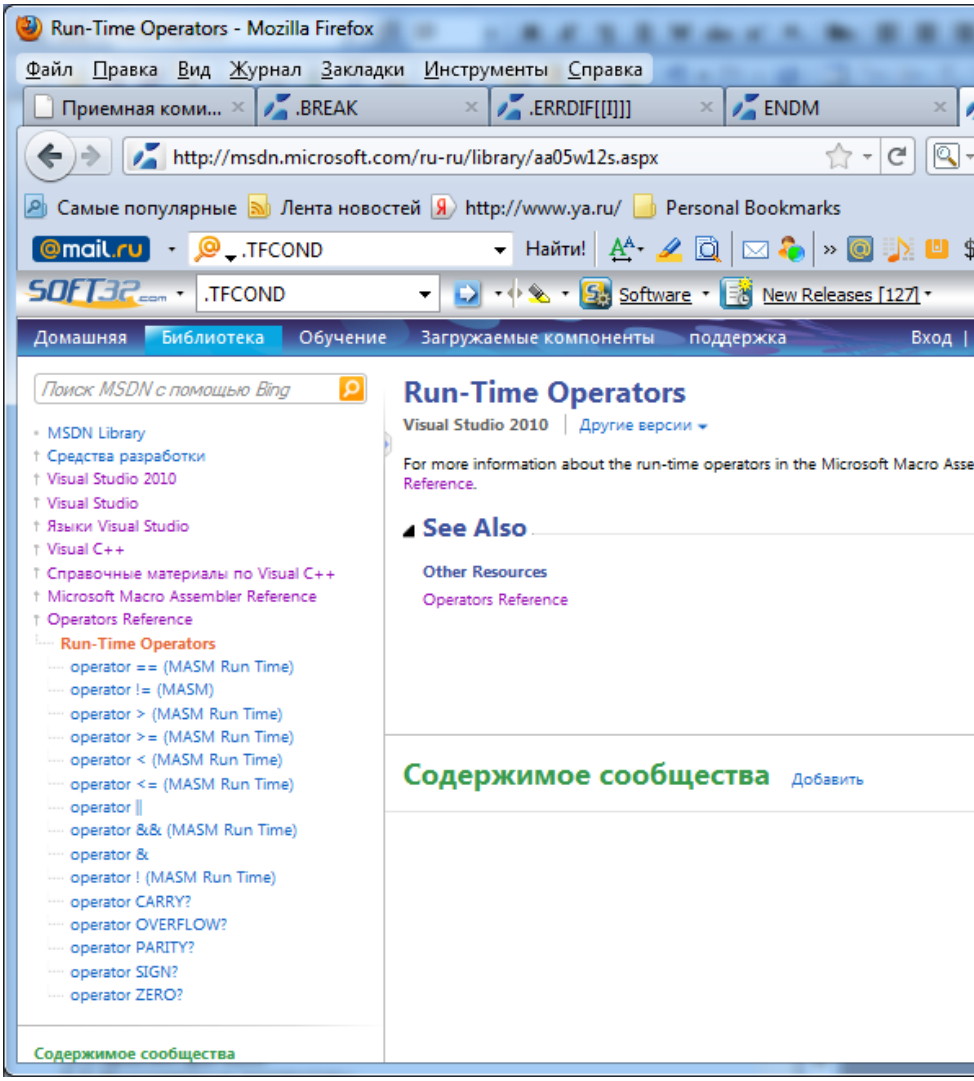
Посмотрим Операторы MASM. В левом поле список по алфавиту.



Например, оператор [ ]:



А это операторы реального времени



3.12. Структура программы на ассемблере

Программа на ассемблере представляет собой совокупность блоков памяти, называемых сегментами. Программа может состоять из одного или нескольких

таких блоков-сегментов. Сегменты программы имеют определенное назначение, соответствующее типу сегментов: кода, данных и стека. Названия типов сегментов отражают их назначение. Деление программы на сегменты отражает сегментную организацию памяти процессоров Intel (архитектура IA-32). Каждый сегмент состоит из совокупности отдельных строк, в терминах теории компиляции называемых **предложениями** языка.

В описании предложений ассемблера могут быть компоненты, которые могут быть пропущены. Из принято заключать в **квадратные скобки.**

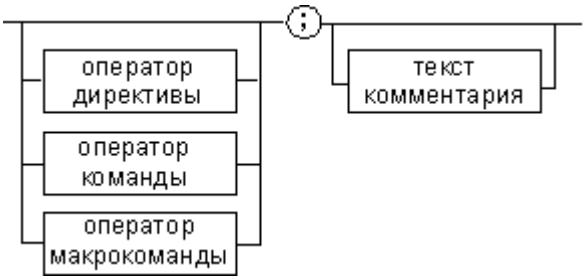
Для языка ассемблера предложения, составляющие программу, могут представлять собой синтаксические конструкции четырех типов.

- **Команды** (инструкции) представляют собой символические аналоги машинных команд. В процессе трансляции инструкции ассемблера преобразуются в соответствующие команды системы команд процессора.
- **Макрокоманды** — это оформляемые определенным образом предложения текста программы, замещаемые во время трансляции другими предложениями.
- **Директивы** - указание транслятору ассемблера на выполнение некоторых действий. У директив нет аналогов в машинном представлении.
- **Комментарии** содержат любые символы, в том числе и буквы русского алфавита. Комментарии игнорируются транслятором.
- Понятие о метасинтаксических языках

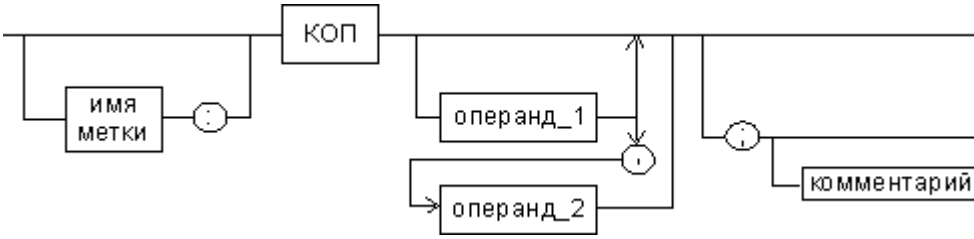
Для распознавания транслятором ассемблера этих предложений их нужно формировать по определенным синтаксическим правилам. Для формального описания синтаксиса языков программирования используются различные метасинтаксические языки, которые представляют собой совокупность условных знаков, образующих нотацию метасинтаксического языка, и правил формирования из этих знаков однозначных описаний синтаксических конструкций целевого языка.

В учебных целях для описания синтаксиса Ассемблера удобно использовать синтаксические диаграммы. В них компоненты предложения отображены блоками.

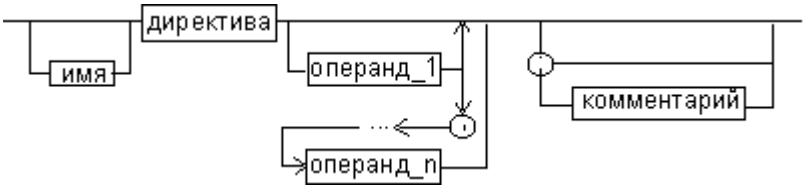
Синтаксическая диаграмма предложения ассемблера.



Синтаксическая диаграмма команд и микрокоманд.



Синтаксическая диаграмма директив.



Компоненты диаграмм:

- Имя метки – символьный идентификатор, значением которого является **адрес первого байта** того предложения исходного текста программы, которое он обозначает.
- Имя — идентификатор, отличающий данную директиву от других одноименных директив. Его значением является **адрес в таблице сиволов**. В результате обработки ассемблером определенной директивы этому имени могут быть присвоены определенные характеристики;
- КОП и директива - это мнемоническое обозначения соответствующей машинной команды, макрокоманды или дирекивы ассемблера.

- Операнды - объекты, над которыми производятся действия. Операнды ассемблера описываются выражениями с числовыми и текстовыми константами, метками и идентификаторами переменных с использованием знаков операций и некоторых зарезервированных слов.
- Комментарий.
- Разделитель точка с запятой (;). За ним следует комментарий.
- Разделитель запятая (,). Применятся в списке операдов..
- Разделитель вертикальное даоеточие (:). Следует после метки, с его помощью идентифицируется метка.
- Лексемы ассемблера

Предложения ассемблера формируются из **лексем**, представляющих собой синтаксически неразделимые последовательности допустимых символов языка, имеющие смысл для транслятора.

Вначале определим алфавит ассемблера, то есть допустимые для написания текста программ символы:

- Символы ASCII (American Standard Code for Information Interchange – американский стандартный код для обмена информацией). Это все латинские буквы A - Z, a – z . В языке ассеммблера **прописные и строчные буквы считаются эквивалентными**.
- Десятичные цифры от 0 до 9.
- специальные знаки _, ?, @, $, &.
- разделители: „ . , [ ] ( ) < > { } + / * % ! " ? \ = #.

Лексемами языка ассемблера являются ключевые слова, идентификаторы, цепочки символов и целые числа. Ключевые слова — это служебные символы языка ассемблера. По умолчанию **регистр символов ключевых слов не имеет значения**. К ключевым словам относятся:

- названия регистров - AL, AH, BL, BH, CL, CH, DL, OH, AX, EAX, BX, EBX, CX, ECX, DX, EDX, BP, EBP, SP, ESP, DI, EDI, SI, ESI, CS, DS, ES, FS, GS, SS, CRO, CR2, CR3, DRO, DRl, DR2, DR3, DR6, DR7.
- операторы - BYTE, SBYTE, WORD, SWORD, DWORD, SDWORD, FWORD, QWORD, TBYTE, REAL4, REALS, REAL10, NEAR16, NEAR32, FAR16, FAR32, AND, NOT, HIGH, LOW, HIGHWORD, LOWWORD, OFFSET, SEG, LROFFSET, TYPE, THIS, PTR, WIDTH, MASK, SIZE, SIZEOF, LENGTH, LENGTHOF, ST, SHORT, TYPE, OPATTR, MOD, NEAR, FAR, OR, XOR, EQ, NE, LT, LE, GT, GE, SHR, SHL и др..

- Названия команд (КОП) ассемблера, префиксов.

Лексемами являются:

- Идентификаторы.
- Комментарии.
- Зарезервированные слова.
- Цепочки символов.
- Целые числа.

Идентификаторы. Идентификатором называется любое имя, назначенное программистом некоторому объекту программы (переменной, константе или метке). При выборе имен идентификаторов необходимо учитывать правила.

- Длина идентификатора до 255 символов, хотя транслятор воспринимает лишь первые 32, а остальные игнорирует.
- Первым символом идентификатора должна быть одна из букв латинского алфавита (A. . z или a. . z) либо символы подчеркивания (_), коммерческого "эт" (@) или знак доллара ($). Последующие символы могут быть также цифрами.
- Идентификатор не должен совпадать с одним из зарезервированных слов языка ассемблера.

Комментарии. Комментарии очень важны для документирования программы. По сути, они являются средством общения разработчика программы с тем, кто булет сопровождать эту программу впоследствии. В начало листинга программы обычно помешается перечисленная ниже информация:

- короткое описание назначения программы;
- фамилия и имя программиста, кто написал программу или внес в нее изменения;
- дата создания программы, а также даты всех последующих изменений в ней.

Комментарии в программах бываютдвух видов:

- **Однострочные**, начинающиеся с символа точки с запятой (;). При этом все символы. расположенные после точки с запятой и до конца текущей строки, игнорируюгся компилятором и поэтому могут быть использованы для размещения комментариев к программе.
- **Блочные**, начинающиеся с директивы COMMENT, за которой следует символ комментария, определяемый прграммистом. При этом компилятор иг-

норирует все строки, расположенные между директивой COMMENT и символом, указанным программистом. Например:

```
COMMENT &
    Это строка комментария.
    А вот еще одна строка комментария
&
```

Зарезервированные слова. В языке ассемблера существует специальный список так называемых зарезервированных слов. Каждое из этих слов несет определенный смысл и поэтому может использоваться только в заранее оговоренном контексте. Резервными являются слова, перечисленные ниже:

- Мнемоники команд, такие как MOV, ADD или MUL, которые соответствуют встроенным командам языка ассемблера, напрямую связанными с машинными командами процессоров семейства IA-32.
- Директивы компилятора, которые определяют порядок ассемблирования программ.
- Атрибуты, с помощью которых определяются характеристики используемых переменных и операндов, такие как размер, например: BYTE или WORD.
- Операторы, используемые в константных выражениях.
- Встроенные идентификаторы ассемблера, такие как @data,
- Операнды

Операнды — это объекты, над которыми или при помощи которых выполняются действия, задаваемые инструкциями или директивами. Машинные команды могут либо совсем не иметь операндов, либо иметь 1 - 3 операнда. Большинство команд требует двух операндов, один из которых является источником, а другой — приемником (операндом назначения).

В двухоперандной машинной команде возможны следующие сочетания операндов:

- регистр — регистр,
- регистр — память,
- память — регистр,
- непосредственный операнд — регистр,
- непосредственный операнд — память.

Один операнд может располагаться в регистре или памяти, а второй операнд **обязательно** должен находиться в регистре или непосредственно в команде. Непосредственный операнд может быть только источником.

Для приведенных ранее правил сочетания типов операндов есть исключения, которые касаются:

- команд работы с цепочками, которые могут перемещать данные из памяти в память;
- команд работы со стеком, которые могут переносить данные из памяти в стек, также находящийся в памяти;
- команд типа умножения, которые, кроме операнда, указанного в команде, неявно используют еще и второй операнд.

Операндами могут быть числа, регистры, ячейки памяти, символьные идентификаторы. При необходимости для расчета некоторого значения или определения ячейки памяти, на которую будет воздействовать данная команда или директива, используются выражения, то есть комбинации чисел, регистров, ячеек памяти, идентификаторов с арифметическими, логическими, побитовыми и атрибутивными операторами.Возможно провести следующую классификацию операндов:

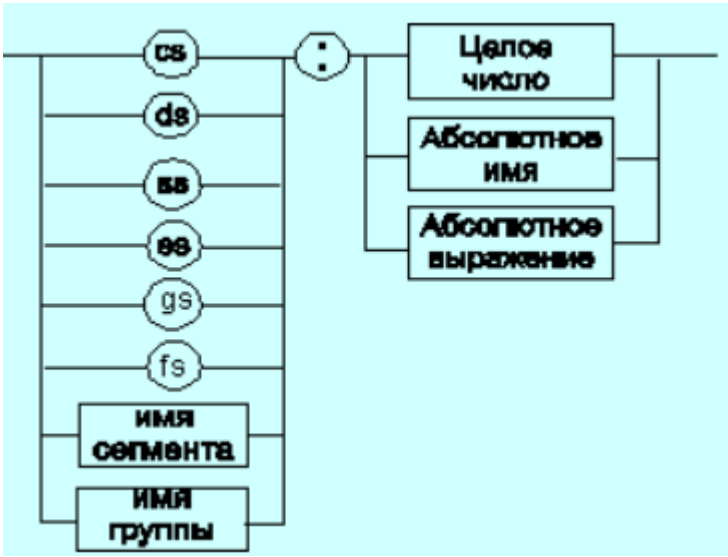
- постоянные, или непосредственные, операнды,
- адресные операнды,
- перемещаемые операнды,
- счетчик адреса,
- регистровый операнд,
- стековый,
- порт ввода-вывода,
- базовый и индексный операнды,
- структурные операнды,
- записи

Непосредственные операнды. Непосредственный операнд задается в самой команде. Это может быть число, строка, имя или выражение, имеющее некоторое фиксированное (константное) значение. Физически непосредственный операнд находится в коде команды, то есть является ее частью. Для его хранения в команде выделяется поле длиной до 32 битов. Непосредственный операнд может быть только вторым операндом (**источником**). Операнд-приемник может находиться либо в памяти, либо в регистре. Например:

- Команда mov ax,0ffffh пересылает в регистр AX 16-ричную константу Offffh.
- Команда add sum,2 складывает содержимое поля по адресу sum с целым числом 2 и записывает результат по месту первого операнда, то есть в память.

Если непосредственный операнд - имя, то оно не должно быть перемещаемым, то есть зависеть от адреса загрузки программы.

Адресные операнды. Задают физическое расположение операнда в памяти с помощью указания двух составляющих адреса: **сегмента** (слева от символа вертикального двоеточия) и **смещения** (справа от него).



Перемещаемые операнды. Это любые символьные имена, представляющие некоторые адреса памяти. Эти адреса могут обозначать местоположение в памяти некоторых инструкции (если операнд — метка) или данных (если операнд — имя области памяти в сегменте данных). Перемещаемые операнды отличаются от адресных тем, что они не привязаны к конкретному адресу физической памяти. Сегментная составляющая адреса перемещаемого операнда неизвестна и будет определена после загрузки программы в память для выполнения.

Счетчик адреса. Специфический вид операнда. Он обозначается знаком **$**. Специфика этого операнда в том, что когда транслятор ассемблера встречает в

исходной программе этот символ, то он подставляет вместо него текущее значение счетчика адреса. Значение счетчика адреса представляет собой смещение текущей машинной команды относительно начала сегмента кода. При обработке транслятором очередной команды ассемблера счетчик адреса увеличивается на длину сформированной машинной команды.

Обработка директив ассемблера не влечет за собой изменения счетчика. Директивы, в отличие от команд ассемблера, — это лишь указания транслятору на выполнение определенных действий по формированию машинного представления программы, и для них транслятором не генерируется никаких конструкций в памяти.

Регистровый операнд. Это просто имя регистра. В программе на ассемблере можно использовать имена всех регистров общего назначения и большинства системных регистров:

- 32-разрядные регистры EAX, EBX, ECX, EDX, ESI, EDI, ESP, EBP;
- 16-разрядные регистры AX, BX, CX, DX, SI, DI, SP, BP;
- 8-разрядные регистры AH, AL, BH, BL, CH, CL, DH, DL;
- сегментные регистры CS, DS, SS, ES, FS, GS;
- системные регистры CRO, CR2, CR3, CR4, DRO, DR1, DR2, DR3, DR6, DR7.

Стековый операнд находится в стеке. Стек – раздел памяти для хранения промежуточных значений. Он использует алгоритм доступа LIFO (Last In First Out – последним пришел, первым ушел). Стек в IF-32 растет в сторону младших адресов.

Порт ввода-вывода. Помимо адресного пространства оперативной памяти процессор поддерживает адресное пространство ввода-вывода, которое используется для доступа к устройствам ввода-вывода. Объем адресного пространства ввода-вывода составляет 64 Кбайт. Для любого устройства компьютера в этом пространстве выделяются адреса. Конкретное значение адреса в пределах этого пространства называется портом ввода-вывода. Физически портуввода-вывода соответствует аппаратный регистр (не путать с регистром процессора), доступ к которому осуществляется с помощью специальных команд ассемблера IN и OUT.

Регистры, адресуемые с помощью порта ввода-вывода, могут иметь разрядность 8,16 или 32 бита, но для конкретного порта разрядность регистра фиксирована. Команды IN и OUT работают с фиксированной номенклатурой объектов. В качестве источника информации или получателя применяются так называе-

мые регистры -аккумуляторы EAX, AX, AL. Выбор регистра определяется разрядностью порта. Номер порта может задаваться непосредственным операндом в командах IN и OUT или значением в регистре DX. Последний способ позволяет динамически определить номер порта в программе.

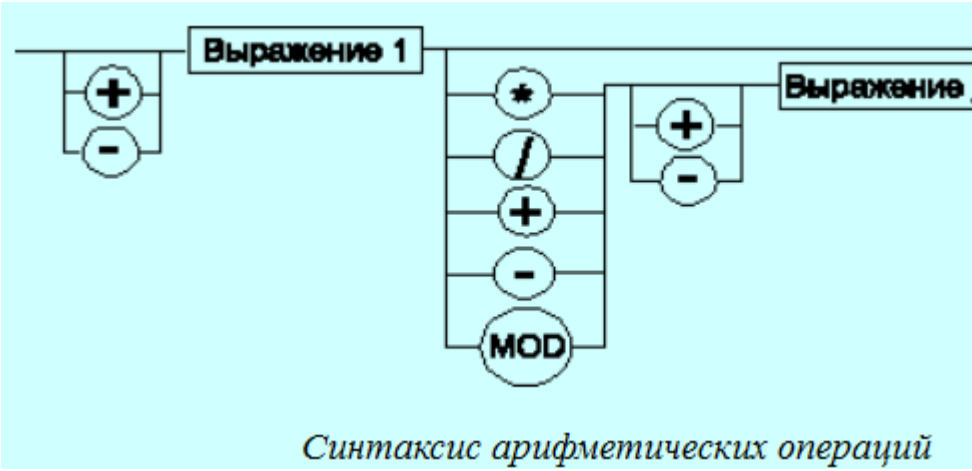
Базовый и индексный операнды. Эти типы операндов используется для реализации косвенной базовой, косвенной индексной адресации или их комбинаций и расширений.

Структурные операнды. Используются для доступа к конкретному элементу сложного типа данных, называемого структурой.

Записи. Аналогично структурному типу используются для доступа к битовому полю некоторой записи.

Операторы. Операнд команды может быть выражением, представляющим собой комбинацию операндов и операторов ассемблера. Транслятор ассемблера рассматривает выражение как единое целое и преобразует его в числовую константу. Логически значением этой константы может быть адрес некоторой ячейки памяти или некоторое абсолютное значение.

Арифметические операторы. Синтаксис описания:

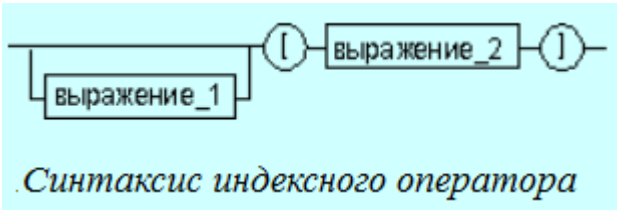


Синтаксис арифметических операций

Синтаксис	
(выражение)	Круглые скобки для выражений, вложенных в выражения.

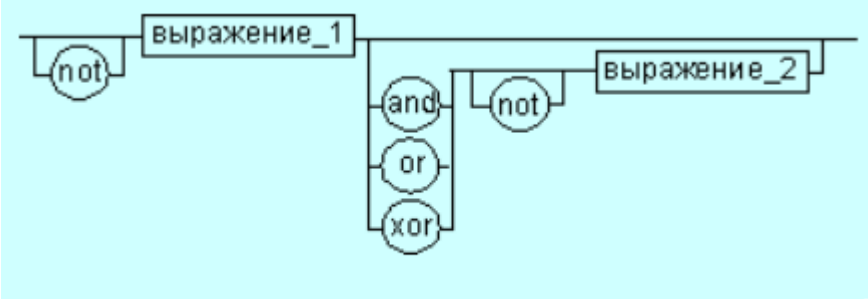
+ выражение	Унарный плюс. Показывает, что значение выражениея положительно.
- выражение	Унарный минус. Изменяет знак выражения
выражение_1 + выражение_2	Сложение выражений.
выражение_1 – выражение_2	Вычитание выражений.
выражение_1 * выражение_2	Целочисленное перемножение выражений.
выражение_1 / выражение_2	Деление целочисленное выражений. Остаток отбрасывается.
выражение_1 MOD выражение_2	Возвращает остаток от деления выражений.
выражение_1 [выражение_2]	Операция [ ] может использоваться для задания сложения выражений.

Индексный оператор. Скобки тоже являются оператором, и транслятор их наличие воспринимает как указание сложить значение выражение_1 за этими скобками с выражение_2, заключенным в скобки. Его синтаксис:



Синтаксис индексного оператора

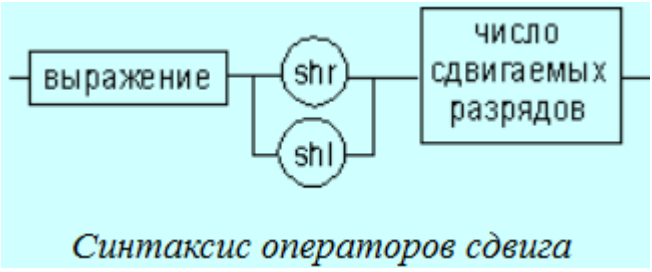
Логические операторы. выполняют над выражениями побитовые операции. Выражения должны быть абсолютными, то есть такими, численное значение которых может быть вычислено транслятором



. Синтаксис логических операторов

Синтаксис	
выражение_1 OR выражение_2	Выполняет для двух выражений поразрядную логическую операцию OR (ИЛИ).
выражение_1 XOR выражение_2	Выполняет для двух выражений поразрядную логическую операцию XOR (исключающее ИЛИ).
выражение_1 AND выражение_2	Выполняет для двух выражений поразрядную логическую операцию AND (И).
NOT выражение	Поразрядное дополнение (инвертирование) выражения.

Операторы сдвига. Выполняют сдвиг выражения на указанное количество разрядов

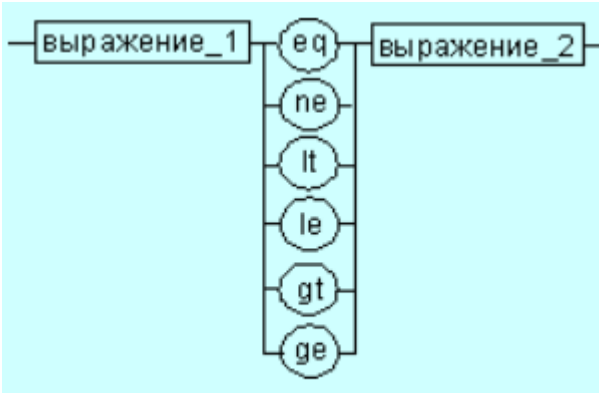


Синтаксис операторов сдвига

Синтаксис	
выражение SHL счетчик	Сдвигает выражение влево на число бит, заданных счетчиком.

	Отрицательное значение счетчика задает сдвиг в противоположном направлении.
выражение SHR счетчик	Сдвигает выражение вправо на число бит, заданных счетчиком. Отрицательное значение счетчика задает сдвиг в противоположном направлении.

Операторы сравнения. Возвращают значение True (истина) или False (ложь) предназначены для формирования логических выражений. Логическое значение “истина” соответствует цифровой единице, а “ложь” — нулю.



Синтаксис операторов сравнения

Синтаксис	
выражение_1 EQ выражение_2	Возвращает значение True, если выражение_1 равно выражение_2.
выражение_1 NE выражение_2	Возвращает значение True, если выражение_1 не равно выражение_2.
выражение_1 GE выражение_2	Возвращает значение True, если выражение_1 больше или равно выражение_2.
выражение_1 GT выражение_2	Возвращает значение True, если выражение_1 больше выражение_2.
выражение_1 LE выражение_2	Возвращает значение True, если выражение_1 меньше или равно выражение_2.
выражение_1 LT выражение_2	Возвращает значение True, если выражение_1 меньше выражение_2.

Операторы опций.

Синтаксис	
LARGE выражение	Задает для смещения выражения размер 32 бита.
SMALL выражение	Задает для смещения выражения размер 16 бит.
SHORT выражение	Выражение будет указателем на код короткого типа (от -128 до +127 байт от текущего адреса программы).

Специальные операторы.

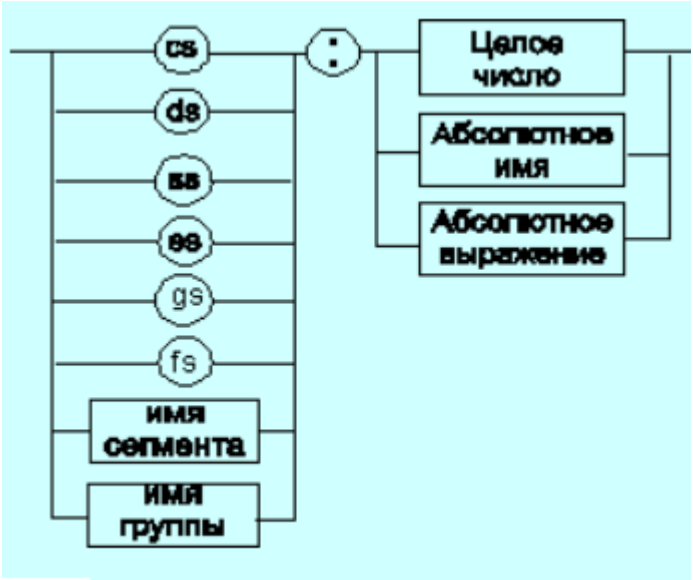
Синтаксис	
& имя	Подставляет фактическое значение параметра макрокоманды "имя".
<выражение>	Интерпретирует текст, как литерал, независимо от возможно содержащихся в нем специальных символов.
! символ	Интерпретирует символ, как литерал, независимо от специального значения, которое он может иметь.
% текст	Интерпретирует текст, как выражение, вычисляет его значение и заменяет текст полученным результатом. Текстом может быть либо числовое выражение, либо текстовое присваивание.
;; комментарий	Подавляет для комментария в макроопределении выделение памяти.

Операторы выборки.

Синтаксис	
HIGH выражение	Возвращает старшую часть (8 бит размера типа) выражения.
LOW выражение	Возвращает младшую часть (8 бит размера типа) выражения.
LENGTH имя	Возвращает число элементов данных, выделенных для имени.
OFFSET выражение	Смещение выражения в текущем сегменте.
.TYPE выражение	Возвращает байт, описывающий режим и область дейст-

	вия выражения.
TYPE выражение	Возвращает число, указывающее размер или тип выражения.

Операторы адресности.



Синтаксис описания адресных операндов

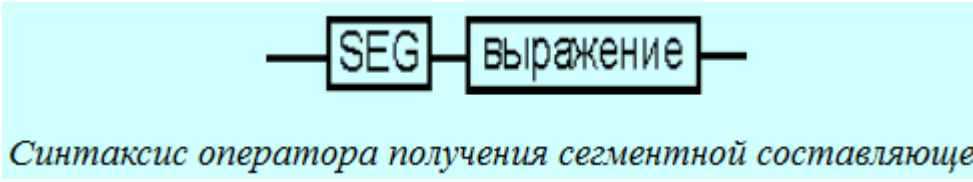
Синтаксис	
CODEPTR выражение	Возвращает используемый по умолчанию размер адреса процедуры.
SIZE имя	Возвращает размер элемента данных, выделенного для переменной с указанным именем.
BYTE PTR выражение	Приводит адресное выражение к размеру в байт.
WORD PTR выражение	Приводит адресное выражение к размеру в слово.
DWORD PTR выражение	Приводит адресное выражение к размеру в двойное слово.
QWORD PTR выра-	Приводит адресное выражение к размеру в четверное

жение	слово.
TBYFE PTR выражение	Приводит адресное выражение к размеру в 10 байт.
PWORD PTR выражение	Приводит адресное выражение к размеру 32-разрядных дальних указателей.
FWORD PTR выражение	Приводит адресное выражение к размеру 32-разрядных дальних указателей.
NEAR PTR выражение	Приводит к тому, что адресное выражения будут ,ближними указателями.
FAR PTR выражение	Приводит к тому, что адресное выражения будут дальними указателями.
PROC PTR выражение	Приводит к тому, что адресное выражения будут ближними или дальними указателями.
тип PTR выражение	Приводит к тому, что адресное выражения будут иметь рахмер типа.
THIS тип	Создает операнд, адресом которого будет текущий сегмент и счетчик адреса. Тип описывает размер операнда и то, представляет ли он собой код или данные.
UNKNOWN выражение	Удаляет из адресного выражения информацию о типе.

Операторы для структур.

Синтаксис	
указательЭлемента.ИмяПоля	Выбирает элемент структуры.
MASK ПолеЗаписи MASK Запись	Возвращает битовую маску для поля записи или всей записи.
WITH ПолеЗаписи WITH Запись	Возвращает длину поля записи или всей записи в битах.

Оператор получения сегментной составляющей адреса выражения. возвращает физический адрес сегмента для выражения, в качестве которого могут выступать метка, переменная, имя сегмента, имя группы или некоторое символическое имя

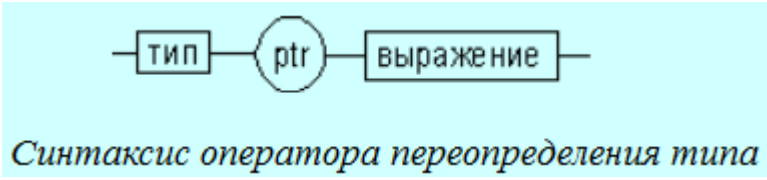


Синтаксис оператора получения сегментной составляюще

Оператор получения смещения выражения. позволяет получить значение смещения выражения (рис. 13) в байтах относительно начала того сегмента, в котором выражение определено.

Синтаксис оператора получения смещения

Оператор переопределения типа. Оператор переопределения типа **ptr** применяется для переопределения или уточнения типа метки или переменной, определяемых выражением. Тип может принимать одно из следующих значений: **byte, word, dword, qword, tbyte, near, far**. Позволяет вызвать не весь операнд, а только его часть.

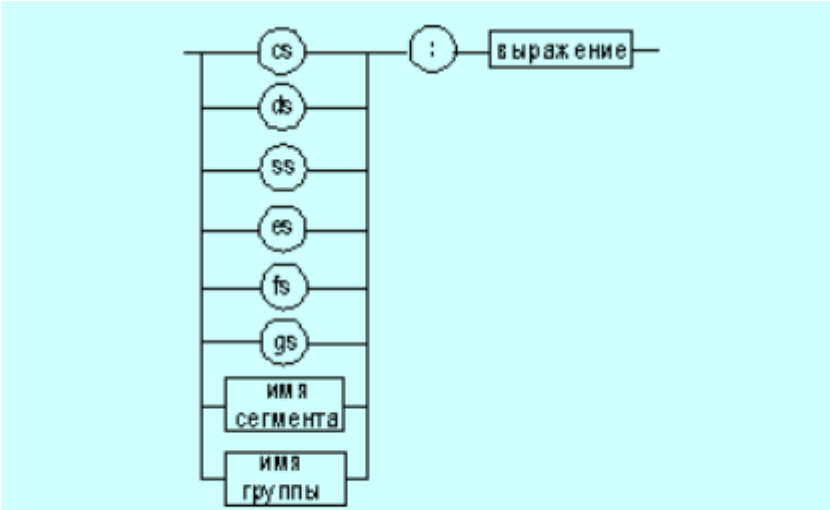


Синтаксис оператора переопределения типа

Синтаксис	
сегмент:выражение группа:выражение	Переопределение сегмента или группы.
Dx ?	Инициализация с неопределенными данными (где Dx - это DB, DD, DF, DP, DQ, DT или DW).
счетчик DUP (выражение)	Повторяет операцию выделения памяти для выражения столько раз, сколько задано значением "счетчик".

Оператор переопределения сегмента. Оператор переопределения сегмента **:** (двоеточие) заставляет вычислять физический адрес относительно конкретно

задаваемой сегментной составляющей: “имя сегментного регистра”, “имя сегмента” из соответствующей директивы SEGMENT или “имя группы”.



Синтаксис оператора переопределения сегмента

Приоритеты операторов. Как и в языках высокого уровня, выполнение операторов ассемблера при вычислении выражений осуществляется в соответствии с их приоритетами (см. таблицу). Операции с одинаковыми приоритетами выполняются последовательно слева направо. Изменение порядка выполнения возможно путем расстановки круглых скобок, которые имеют наивысший приоритет.

Операторы и их приоритет

Оператор	Приоритет
length, size, width, mask, (,), [,], <, >	1
.	2
:	3
ptr, offset, seg, type, this	4
high, low	5
+, - (унарные)	6
*, /, mod, shl, shr	7
+, -, (бинарные)	8
eq, ne, lt, le, gt, ge	9
not	10
and	11
or, xor	12
short, type	13

Команды. В языке ассемблера командой называется оператор программы, который **непосредственно** выполняется процессором после того, как программа будет скомпилирована в машинный код, загружена в память и запущена на выполнение (т.е. на этапе выполнения программы). Любая команда состоит из четырех основных частей:

- необязательной метки;
- мнемоники команды, которая присутствует всегда;
- одного или нескольких операндов (как правило, они присутствуют в любой команде, хотя есть ряд команд, для которых операнды не требуются);
- необязательного комментария.

Любая строка исходного кода программы может также содержать только метку или только комментарий. Стандартный формат команды ассемблера

Метка. Метка является обычным идентификатором, с помошью которого в программе помечается некоторый участок кода или данных. В процессе обработки исходного текста программы ассемблер назначает каждому оператору программы числовой адрес. Таким образом, метке, размешенной непосредственно перед командой, также назначается адрес этой команды. Аналогично, если разместить метку перед переменной, ей будет назначен адрес этой переменной.

Для чего вообще нужны метки? Ведь в программах на языке ассемблера можно непосредственно использовать числовые адреса. Например, приведенная ниже команда загружает 16-разрядное число, расположенное по адресу0020, в регистр ax.

```
mov ax,[0020]
```

Очевидно, что при вставке в программу новой переменной, адреса всех последующих за ней переменных автоматически изменятся. Поэтому программист в каждом подобном случае должен вручную скорректировать в программе ссылки наподобие [0020].

Разумеется, что подобный стиль программирования создает массу неудобств и эффективность его крайне низкая. Следовательно, если присвоить переменной, расположенной по адресу 0020h, метку, то ассемблер будет автоматически подставлять ее значение при компиляции. Теперь приведенную выше команду можно переписать так:

```
myVariable BYTE   4              ; myVariable = 4
mov        ax,myVariable         ; myVariable загружена в регистр ax
```

Метки кода. Метки, расположенные в коде программы (т.е. в сегменте кода, где размешаются команды процессора), **должны заканчиваться символом вертикального двоеточия (:)**. Подобные метки обычно используются для указания участка программы, которому будет передано управление в командах перехода или организации циклов. Например, приведенная ниже команда безусловного перехода JMP (от англ. "jump") передает управление команде, помеченной как **target**, в результате чего в программе создается цикл:

```
target:
```

```
    mov ax,bx
    …
    jmp target
```

Метка в коде программы может находиться на одной строке с командой, либо занимать самостоятельную строку:

target: mov ax,bx

либо так:

```
target:
    mov ax,bx
```

Метки данных. При использовании метки в сегменте данных программы (т.е. там, где размещаются и определяются переменные), она **не должна заканчиваться символом вертикального двоеточия.** Ниже приведен пример определения переменной под именем first:

```
first BYTE 10
```

При выборе имен меток следует учитывать общие правила для имен идентификаторов. Кроме того, имя, выбранное для метки, должно быть уникальным в пределах одного исходного файла программы. Например, если в файле с исходным кодом вашей программы уже есть метка с именем first, вы не можете присвоить это же имя другой метке, расположенной в том же файле.

3.13. Типы данных

При программировании на языке ассемблера используются данные следующих типов.

Непосредственные данные, представляющие собой числовые или символьные значения, являющиеся частью команды. Непосредственные данные формируются программистом в процессе написания программы для конкретной команды ассемблера.

Данные простого типа. Они описываются с помощью ограниченного набора директив резервирования памяти, позволяющих выполнить самые элементарные операции по размещению и инициализации числовой и символьной информации. При обработке этих директив ассемблер сохраняет в своей **таблице символов** информацию о местоположении данных (значения сегментной составляющей адреса и смещения) и типе данных, то есть единицах памяти, вы-

деляемых для размещения данных в соответствии с директивой резервирования и инициализации данных.

Данные сложного типа, которые были введены в язык ассемблера с целью облегчения разработки программ. Сложные типы данных строятся на основе базовых типов, которые являются как бы кирпичиками для их построения. Введение сложных типов данных позволяет несколько сгладить различия между языками высокого уровня и ассемблером. У программиста появляется возможность сочетания преимуществ языка ассемблера и языков высокого уровня (в направлении абстракции данных), что в конечном итоге повышает эффективность конечной программы.

Простые типы данных. Понятие простого типа данных носит двойственный характер. С точки зрения размерности (физическая интерпретация), микропроцессор аппаратно поддерживает следующие основные типы данных:

- Байт.
- Слово.
- Двойное слово.
- Учетверное слово.

Размерности простых типов данных

Байт						
8 бит						
Слово						
16 бит						
Двойное слово					Старшее слово	
32 бит					31	23
Учетверенное слово	Старшее двойное слово				Младше	
64 бита	63	55	47	39	31	23

Байт — 8 последовательно расположенных битов, пронумерованных от 0 до 7, при этом бит 0 является самым младшим значащим битом;

Слово — последовательность из 2 байт, имеющих последовательные адреса. Размер слова — 16 бит; биты в слове нумеруются от 0 до 15. Байт, содержащий

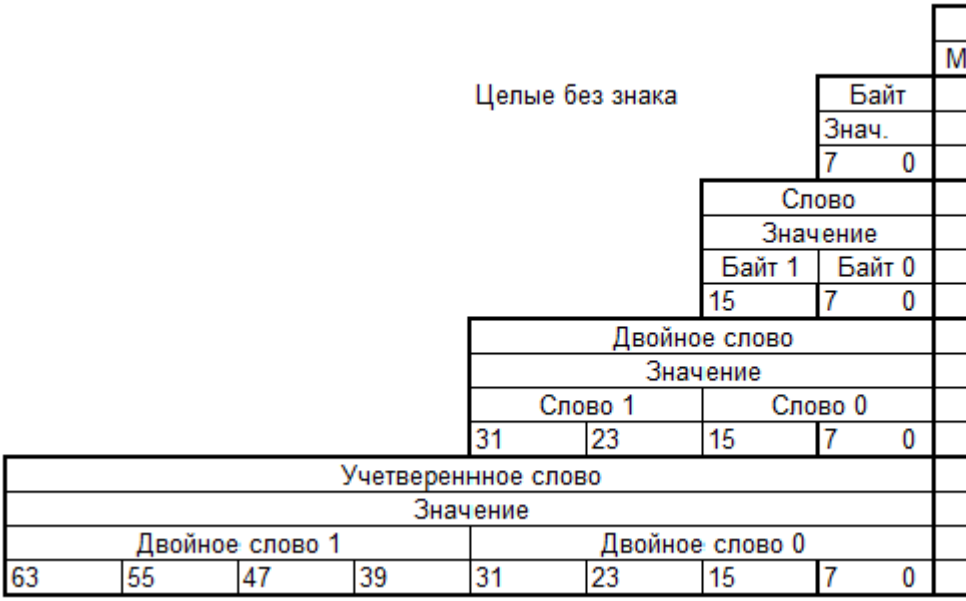
нулевой бит, называется младшим байтом, а байт, содержащий 15-й бит - старшим байтом. Микропроцессоры Intel имеют важную особенность — **младший байт всегда хранится по меньшему адресу**. Адресом слова считается адрес его младшего байта. Адрес старшего байта может быть использован для доступа к старшей половине слова.

Двойное слово — последовательность из 4 байт (32 бита), расположенных по последовательным адресам. Нумерация этих бит производится от 0 до 31. Слово, содержащее нулевой бит, называется младшим словом, а слово, содержащее 31-й бит, - старшим словом. Младшее слово хранится по меньшему адресу. Адресом двойного слова считается адрес его младшего слова. Адрес старшего слова может быть использован для доступа к старшей половине двойного слова.

Учетверенное слово — последовательность из 8 байт (64 бита), расположенных по последовательным адресам. Нумерация бит производится от 0 до 63. Двойное слово, содержащее нулевой бит, называется младшим двойным словом, а двойное слово, содержащее 63-й бит, — старшим двойным словом. Младшее двойное слово хранится по меньшему адресу. Адресом учетверенного слова считается адрес его младшего двойного слова. Адрес старшего двойного слова может быть использован для доступа к старшей половине учетверенного слова.

Кроме трактовки типов данных с точки зрения их разрядности, микропроцессор на уровне команд поддерживает логическую интерпретацию этих типов.

Двоичные числа. Целый тип без знака — двоичное значение без знака, размером 8, 16, 32 и 64 бита.



Целый тип со знаком — двоичное значениесо знаком в старшем бите, размеры 8, 16, 32 и 64 бита. Ноль в знаковом бите в операндах соответствует положительному числу, а единица — отрицательному. Отрицательные числа представляются в дополнительном коде.

									Мини
				Целые со знаком			Байт		-(2 в сте
							Знак	Знач.	-12
							7	6 0	
						Слово			-(2 в сте
						Знак	Значение		-32
						Байт 1		Байт 0	
						15	14	7 0	
				Двойное слово					-(2 в сте
				Знак	Значение				–2 147 4
				Слово 1			Слово 0		
				31	30	23	15	7 0	
Учетвереннное слово									-(2 в сте
Знак	Значение								
Двойное слово 1				Двойное слово 0					
63	62	55	47	39	31	23	15	7 0	

BCD – двоично-десятичное представление. Оно использует двочное представление десятичных цифр 0 – 9. Одна десятичная цифра в двоичном коде требует 4 бита (полбайта). Различают подтипы:

- Неупакованный. Неупакованные десятичные числа хранятся как байтовые значения без знака по одной цифре в каждом байте. Значение цифры определяется младшим полубайтом. Старший полубайт содержит 0000.
- Упакованный. Каждая цифра хранится в своем полубайте. Цифра в старшем полубайте (биты 4–7) является старшей

BCD - двоично-десятичное представление

Неупакованный тип								Старший по
Цифра 0-9								Биты 0000
7	6	5	4	3	2	1	0	
Упакованный тип								Старший по
Цифра 0-9				Цифра 0-9				Цифра 0-9
7	6	5	4	3	2	1	0	

Представление многозначного BCD

Неупакованный тип																											
Байт 2								Байт 1								Байт 0											
Биты 0000				BCD				Биты 0000				BCD				Биты 0000				BCD							
23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0				
Упакованный тип																											
Байт 2								Байт 1								Байт 0											
BCD				BCD				BCD				BCD				BCD				BCD							
23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0				

Указатели на память. Различаются указатели на память двух типов:

- ближнего типа (NEAR)— 32-разрядный логический адрес, представляющий собой относительное смещение в байтах от начала сегмента. Эти указатели могут также использоваться в сплошной (плоской) модели памяти, где сегментные составляющие одинаковы;
- дальнего типа (FAR) — 48-разрядный логический адрес, состоящий из двух частей: 16-разрядной сегментной части — селектора, и 32-разрядного смещения.

Указатели на память

Указатель ближнего типа			
Смещение			
31	23	15	7

Указатель дальнего типа					
Селектор		Смещение			
47	39	31	23	15	7

Цепочка.

Цепочка представляет собой некоторый непрерывный набор байтов, слов или двойных слов максимальной длины до 4 Гбайт. Цепочка может содержать набор байтов, слов и двойных слов.

Цепочка

Цепочка байтов				
Байт N	Байт ...	Байт 2	Байт 1	Байт

Цепочка слов							
Слово N		Слово ...		Слово 1		Слово 0	

Цепочка двойных слов			
Двойное слово N	Двойное слово ...	Двойное слово 1	Двойное слово

Битовое поле. Представляет собой непрерывную последовательность бит, в которой каждый бит является независимым и может рассматриваться как отдельная переменная. Битовое поле может начинаться с любого бита любого байта и содержать до 32 бит.

Битовые поля

Последовательность байтов																							
Байт 1																Байт							
31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8

Битовое поле произвольной длины															
14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	

Непосредственные данные и данные простого типа являются **элементарными**, или **базовыми**; работа с ними поддерживается на уровне системы команд микропроцессора. Используя данные этих типов, можно формализовать и запрограммировать практически любую задачу. Но насколько это будет удобно — вот вопрос.

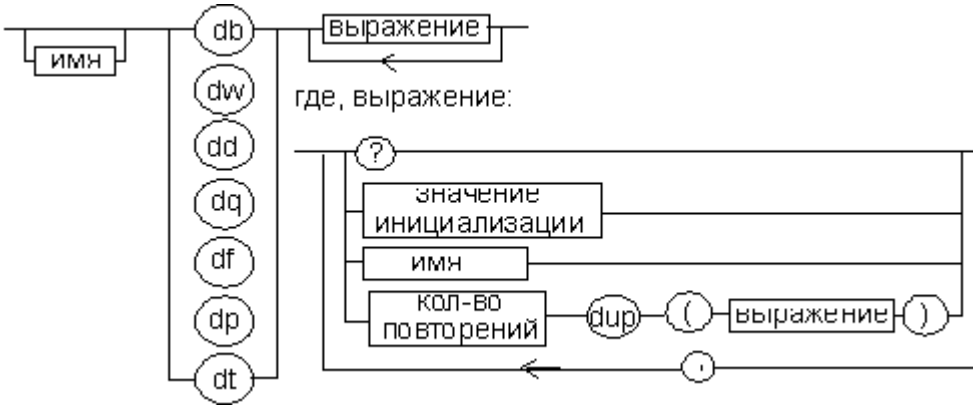
Обработка информации, в общем случае, процесс очень сложный. Это косвенно подтверждает популярность языков высокого уровня. Одно из несомненных достоинств языков высокого уровня — поддержка развитых структур данных.

При их использовании программист освобождается от решения конкретных проблем, связанных с представлением числовых или символьных данных, и получает возможность оперировать информацией, структура которой в большей степени отражает особенности предметной области решаемой задачи. В то же самое время, чем выше уровень такой абстракции данных от конкретного их представления в компьютере, тем большая нагрузка ложится на компилятор с целью создания действительно эффективного кода. Ведь нам уже известно, что в конечном итоге все написанное на языке высокого уровня в компьютере будет представлено на уровне машинных команд, работающих только с базовыми типами данных. Таким образом, самая эффективная программа — программа, написанная в машинных кодах, но писать сегодня большую программу в машинных кодах — занятие не имеющее слишком большого смысла.

Директивы резервирования и инициализации данных TASM и MASM. В TASM и MASM определены несколько директив резервирования и инициализации данных. Машинного эквивалента этим директивам нет.

Очень важно уяснить себе порядок размещения данных в памяти. Он напрямую связан с логикой работы микропроцессора с данными. Микропроцессоры Intel требуют следования данных в памяти по принципу: **младший байт по младшему адресу.**

Директивы резервирования и инициализации данных простых типов имеют формат.



На рисунке использованы следующие обозначения:

- **?** показывает, что содержимое поля не определено, то есть при задании директивы с таким значением выражения содержимое выделенного участка физической памяти изменяться не будет. Фактически, создается неинициализированная переменная;
- **Значение инициализации** — значение элемента данных, которое будет занесено в память после загрузки программы. Фактически, создается инициализированная переменная, в качестве которой могут выступать константы, строки символов, константные и адресные выражения в зависимости от типа данных. Подробная информация приведена в приложении 1;
- **Имя** — некоторое символическое имя метки или ячейки памяти в сегменте данных, используемое в программе.
- **Выражение с оператором повторения dup**. В памяти резервируется под значение после dup столько раз, сколько определено в количестве повторений.

Директивы резервирования памяти ассемблеров TAS

Ассемблер TASM		Ассемблер TASM		
Имя	Аббревиатура	Имя	Аббревиатура	Название
db	Define Byte	BYTE		Байт
		SBYTE	S (Sign) + BYTE	Байт со знаком
dw	Define Word	WORD		Слово
		SWORD	S (Sign) + WORD	Слово со знаком
dd	Define Double word	DWORD	D (Double) + WORD	Двойное слово
		SDWORD	S (Sign) + DWORD	Двойное слово со знаком
df	Define Far word	FWORD	F (Far) + WORD	Слово в 6 байт (указатель даль
dp	Define Pointer			Слово в 6 байт (указатель)
dq	Define Quarter word	QWORD	Q (Quarter) + WORD	Учетверенное слово
dt	Define Ten bytes	TBYTE	T (Ten) + BYTE	Слово в 10 байт
		REAL4		Короткое вещественное с ПТ (4
		REAL8		Короткое вещественное с ПТ (8
		REAL10		Расширенное вещественное с

Например, в переменной типа DWORD можно сохранить любое 32-разрядное целое значение. Однако на некоторые типы накладываются более жесткие ограничения. Например, переменной типа REAL4 можно присвоить только вещественную константу.

Размер резервируемой памяти определяет диапазон представимых данных:

- Для чисел без знака от 0 до $2^{\text{Биты}} - 1$.

- Для чисел со знаком от $-2^{Биты-1}$. до $2^{Биты-1} – 1$.
- Для строк это Биты/8. Каждый символ в одном байте.

Перечисленные типы данных относятся к целочисленным значениям, за исключением последних трех. При описании этих трех типов используется аббревиатура "IEEE", которая означает, что эти типы данных соответствуют стандарту представления вещественных чисел, принятому отделением информатики Института инженеров по электротехнике и электронике (IEEE).

Любой переменной, объявленной с помощью директив описания простых типов данных, ассемблер присваивает три атрибута:

- Сегмент (**seg**) — адрес начала сегмента, содержащего переменную;
- Смещение (**offset**) в байтах от начала сегмента с переменной;
- Тип (**type**) — определяет количество памяти, выделяемой переменной в соответствии с директивой объявления переменной.

Получить и использовать значение этих атрибутов в программе можно с помощью рассмотренных нами операторов ассемблера **seg**, **offset** и **type**.

Операторы определения данных. С помощью оператора определения данных в программе резервируется область памяти соответствующей длины для размещения переменной. При необходимости этой переменной можно назначить имя. Операторы определения данных используются в программе на ассемблере для создания переменных, типы которых перечислены в таблице выше. Синтаксис оператора следующий:

[имя] директива инициализатор [,инициализатор]..

Инициализаторы. При определении данных должен быть указан хотя бы один инициализатор, даже если переменной не назначается какого-то конкретного значения (в этом случае значение инициализатора равно ?). Все дополнительные инициализаторы перечисляются через запятую. Для целочисленных типов данных инициализатор является целочисленной константой либо выражением, значение которого соответствует размеру определяемых данных (BYTE, WORD, и т.д.).

Независимо от используемого формата чисел, все инициализаторы автоматически преобразовываются ассемблером в двоичную форму. Другими словами, в результате компиляции инициализаторов 00110010b, 32h и 50d будет получено одинаковое двоичное значение.

Определение переменных типа BYTE и SBYTE. Директивы BYTE (определяет беззнаковый байт) и SBYTE (определяет знаковый байт) используются в операторах определения данных, с помощью которых в программе выделяется память под одну или несколько знаковых или беззнаковых переменных длиной 8 битов. Каждый инициализатор должен быть либо 8-разрядным целочисленным выражением или символьной константой. Например:

```
value1    BYTE    'A'     ; Символьная константа
value2    BYTE    0       ; Наименьшее беззнаковое байтовое значение
value3    BYTE    255     ; Наибольшее беззнаковое байтовое значение
value4    SBYTE   -128    ; Наименьшее знаковое байтовое значение
value5    SBYTE   + 127   ; Наибольшее знаковое байтовое значение
```

Для наглядности мы выделили ключевые слова BYTE и SBYTE прописными буквами, но вы с тем же успехом можете записать их и строчными буквами.

Чтобы оставить переменную неинициализированной (т.е. при выделении под нее памяти не присваивать ей никакого значения), вместо инициализатора используется знак вопроса. Такая форма записи предполагает, что значение данной переменной будет назначено во время выполнения программы с помощью специальных команд процессора. Вот пример:

```
value6    BYTE    ?
```

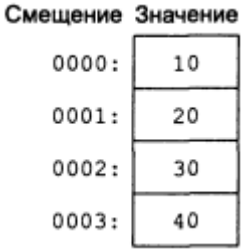
Имена переменных. Имя переменной является меткой, значение которой соответствует смещению данной переменной относительно начала сегмента, в котором она расположена. Например, предположим, что переменная valuel1 расположена в сегменте данных со смещением 0 и занимает один байт памяти. Тогда переменная value2 будет располагаться в том же сегменте со смещением 1:

```
.data
    valuel    BYTE    10h
    value2    BYTE    20h
```

Множественная инициализация. Если в одном и том же операторе определения данных используется несколько инициализаторов, то присвоенная этому оператору метка относится только к первому байту данных. В приведенном ниже примере подразумевается, что метке list соответствует смещение 0. Тогда значение 10 располагается со смещением 0 относительно сегмента данных, значение 2 0 — со смешением 1, 3 0 — со смещением 2 и 4 0 — со смещением 3:

```
.data
    list  BYTE 10,20,30,40
```

На рисунке эта последовательность байтов показана наглядно вместе с соответствующим значением смещения.



Определение строк. Чтобы определить в программе текстовую строку, нужно составляющую ее последовательность символов заключить в кавычки. Чаще всего в программах используются так называемые нуль-завершенные (null-terminated) строки, или строки, оканчивающиеся нулевым байтом, т.е. байтом, значение которого равно двоичному нулю. Этот тип строк используется в таких популярных языках программирования, как C/C++? и Java, а также передается в качестве параметров функциям системы Microsoft Windows. Ниже приведен пример нуль-завершенной строки:

```
Privetstvie  BYTE   "Добрый день!"
```

Каждый символ данной строки занимает один байт памяти.

Определение переменных типа WORD и SWORD.

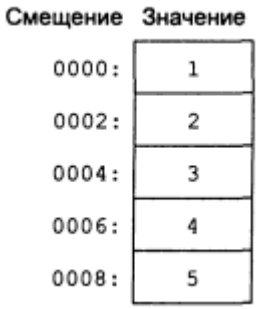
С помощью директив WORD (определить слово) и SWORD (определить слово со знаком) в программах выделяется память для хранения 16-разрядных целых значений. Например:

```
word1     WORD     65535     ; Наибольшее беззнаковое значение
word2     SWORD    -32768    ; Наименьшее знаковое значение
word3     WORD     7         ; Неинициализированное беззнаковое значение
```

Для создания массива 16-разрядных слов можно воспользоваться либо оператором DUP, либо явно перечислить значения каждого элемента массива через запятую. Вот пример массива слов, содержащего определенные значения:

```
myList    WORD    1, 2, 3, 4,.5
```

На рисугке эта последовательность слов показана наглядно вместе с соответствующим значением смещения. Предполагается, что переменная myList располагается со смещением 0. Обратите внимание, что в данном случае значение смещения каждого элемента массива увеличивается на 2 (т.е. на размер элемента массива в байтах).



Определение переменных типа DWORD и SDWORD.

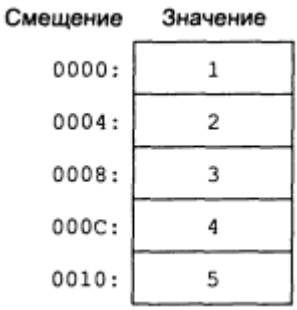
С помощью директив DWORD (определить двойное слово) и SDWORD (определить двойное слово со знаком) в программах выделяется память для хранения 32-разрядных целых значений. Например:

```
val1    DWORD    12345678h     ; Беззнаковое значение
val2    SDWORD   -2147483648   ; Знаковое значение
```

Для создания массива 32-разрядных слов можно воспользоваться либо оператором DUP, либо явно перечислить значения каждого элемента массива через запятую. Вот пример массива слов, содержащего определенные значения:

```
myList    DWORD    1, 2, 3, 4,.5
```

На рисугке эта последовательность слов показана наглядно вместе с соответствующим значением смещения. Предполагается, что переменная myList располагается со смещением 0. Обратите внимание, что в данном случае значение смещения каждого элемента массива увеличивается на 4 (т.е. на размер элемента массива в байтах).



Определение переменных типа QWORD. С помощью директивы QWORD (определить учетверенное слово) в программах выделяется память для хранения 64-разрядных целых значений. Например:

```
quad1    QWORD    1000000000123456789Ah
```

Определение переменных типа TBYTE. С помощью директивы TBYTE (определить 10 байтов) в программах выделяется память для хранения 80-разрядных целых значений. Этот тип данных в основном используется для хранения десятичных упакованных целых чисел (двоично-кодированных целых чисел). Для работы с этими числами используется специальный набор команд математического сопроцессора. Вот пример определения: Например:

```
val1     TBYTE    10000000000123456789Ah
```

Определение переменных вещественного типа. Директива REAL4 определяет в программе 4-байтовую переменную вещественного типа одинарной точности. Директива REAL8 определяет 8-байтовую переменную вещественного типа двойной точности, а REAL10 — 10-байтовую переменную вещественного типа расширенной точности. После каждой из директив необходимо указать один или один или несколько инициализаторов, значение которых должно соответствовать длине выделяемого участка памяти под переменную:

```
rval1    REAL4    -2.1 з
rval2    REAL8    3.2E-260
rval3    REAL10   4.6E+4096
```

В табл.ице перечислены характеристики, такие как количество значащих цифр и диапазоны возможных значений для каждого из трех основных вещественных типов данных.

Тип	Количество значащих десятичных цифр	Диапазон значений
Короткое вещественное	6	$1,18x10^{-38}...3,40x10^{38}$
Длинное вещественное	15	$2,23x10^{-308}...1,79x10^{30}$
Расширенное вещественное	19	$3,37x10^{-4932}...1,18x10^{4}$

Данные сложного типа. MASM и TASM поддерживают следующие сложные типы данных:

- массивы;
- структуры;
- объединения;
- записи.

Разберемся более подробно с тем, как определить данные этих типов в программе и организовать работу с ними.

Массив (MASsiv) - структурированный тип данных, состоящий из некоторого числа элементов **одного** типа.

Специальных средств описания массивов в программах ассемблера, конечно, нет. При необходимости использовать массив в программе его нужно моделировать одним из следующих способов:

- Перечислением элементов массива в поле операндов одной из директив описания данных. При перечислении элементы разделяются запятыми. К примеру:
```
; массив из 5 элементов.Размер каждого элемента 4 байта
mas     dd      1,2,3,4,5
```
- Используя оператор повторения **dup**. К примеру:
```
; массив из 5 нулевых элементов
; Размер каждого элемента 2 байта
mas     dw      5    dup (0)
```
- Используя директивы **label** и **rept**. Пара этих директив может облегчить описание больших массивов в памяти и повысить наглядность такого описания. Директива **rept** относится к макросредствам языка ассемблера и вызывает повторение указанное число раз строк, заключенных между директивой и строкой endm.

- Использование цикла для инициализации значений области памяти, которую можно будет впоследствии трактовать как массив.

Двухмерный массив. С представлением **одномерных** массивов в программе на ассемблере и организацией их обработки все достаточно просто. А как быть если программа должна обрабатывать **двухмерный** массив? Все проблемы возникают по-прежнему из-за того, что специальных средств для описания такого типа данных в ассемблере нет. **Двухмерный** массив нужно моделировать. На описании самих данных это почти никак не отражается — память под массив выделяется с помощью директив резервирования и инициализации памяти.

Непосредственно моделирование обработки массива производится в сегменте кода, где программист, описывая алгоритм обработки ассемблеру, определяет, что некоторую область памяти необходимо трактовать как двухмерный массив. При этом вы вольны в выборе того, как понимать расположение элементов двухмерного массива в памяти: по строкам или по столбцам.

Структура (STRUCture) - это тип данных, состоящий из фиксированного числа элементов **разного** типа.

В приложениях часто возникает необходимость рассматривать некоторую совокупность данных разного типа как некоторый единый тип. Это очень актуально, например, для программ баз данных, где необходимо связывать совокупность данных разного типа с одним объектом. С целью повысить удобство использования языка ассемблера в него также был введен такой тип данных.

Для использования структур в программе необходимо выполнить три действия:

- Задать шаблон структуры. По смыслу это означает определение нового типа данных, который впоследствии можно использовать для определения переменных этого типа.
- Определить экземпляр структуры. Этот этап подразумевает инициализацию конкретной переменной заранее определенной (с помощью шаблона) структурой.
- Организовать обращение к элементам структуры.

Очень важно, чтобы вы с самого начала уяснили, в чем разница между **описанием** структуры в программе и ее **определением**.

- Описать структуру в программе означает лишь указать ее схему или шаблон; память при этом не выделяется. Этот шаблон можно рассматривать

лишь как информацию для транслятора о расположении полей и их значении по умолчанию.

- Определить структуру — значит, дать указание транслятору выделить память и присвоить этой области памяти символическое имя.

Описать структуру в программе можно только один раз, а определить — любое количество раз.

Пример.

```
worker      struc                   ; информация о сотруднике, начало
name        db      30      dup(' ')    ; фамилия, имя, отчество, 30 пустых
полей рахмером 1 байт
sex         db      'м'             ; пол, по умолчанию 'м' — мужской
position    db      30      dup (' ')   ; должность, 30 пустых полей рахмером
1 байт
age         db      2       dup(‘ ’)    ; возраст, 2 пустых поля рахмером 1
байт
standing    db      2       dup(‘ ’)    ; стаж, 2 пустых поля рахмером 1 байт
salary      db      4       dup(‘ ’)    ; оклад в рублях, 4 пустых поля рахме-
ром 1 байт
birthdate   db      8       dup(‘ ’)    ; дата рождения, 8 пустых полей рахме-
ром 1 байт
worker      ends                    ; информация о сотруднике, конец
```

Объединение (UNION) - тип данных, позволяющий трактовать одну и ту же область памяти как имеющую **разные типы и имена**.

Представим ситуацию, когда мы используем некоторую область памяти для размещения некоторого объекта программы (переменной, массива или структуры). Вдруг после некоторого этапа работы у нас отпала надобность в использовании этих данных. Обычно память останется занятой до конца работы программы. Конечно, в принципе, ее можно было бы использовать для хранения других переменных, но при этом без принятия специальных мер нельзя изменить тип и имя. Неплохо было бы иметь возможность переопределить эту область памяти для объекта с другим типом и именем. Язык ассемблера предоставляет такую возможность в виде специального типа данных, называемого **объединением**.

Пример структуры с вложенным объединением.

```
pnt struc                   ; структура pnt, содержащая вложенное объединение
```

```
    union                   ; описание вложенного в структуру объединения
        offs_16     dw ?    ; объект размером 2 байта
        offs_32     dd ?    ; объект размером 4 байта
    ends                    ; конец описания объединения
segm        dw ?
ends                        ; конец описания структуры
```

Запись (RECORD) - структурный тип данных, состоящий из фиксированного числа элементов длиной от одного до нескольких бит. При описании записи для каждого элемента указывается его длина в битах и, что необязательно, некоторое значение. Суммарный размер записи определяется суммой размеров ее полей и не может быть более 8, 16 или 32 бит. Если суммарный размер записи меньше указанных значений, то все поля записи “прижимаются” к младшим разрядам.

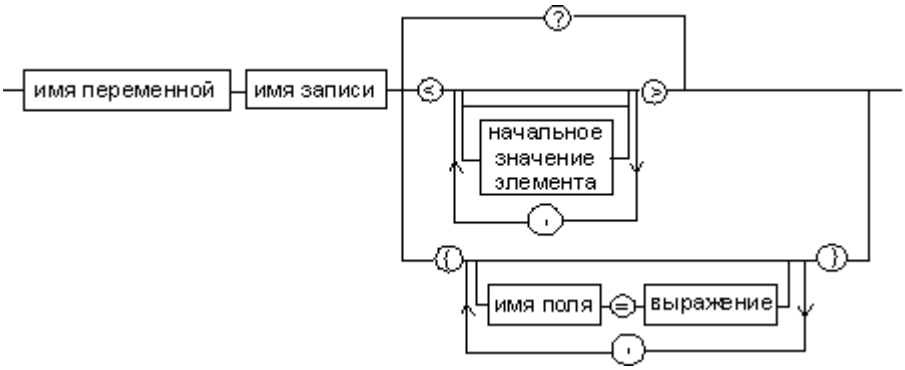
Подумаем, зачем тратить под некоторый программный индикатор со значением “включено-выключено” целых 8 разрядов, если вполне хватает одного? А если таких индикаторов несколько, то расход оперативной памяти может стать весьма ощутимым. Когда мы знакомились с логическими командами, то говорили, что их можно применять для решения подобной проблемы. Но это не совсем эффективно, так как велика вероятность ошибок, особенно при составлении битовых масок.

TASM предоставляет нам специальный тип данных, использование которого помогает решить проблему работы с битами более эффективно. Речь идет о специальном типе данных **— записях**.

Использование записей в программе, так же, как и структур, организуется в три этапа:

- Задание шаблона записи, то есть определение набора битовых полей, их длин и, при необходимости, инициализация полей.
- Определение экземпляра записи. Так же, как и для структур, этот этап подразумевает инициализацию конкретной переменной типом заранее определенной с помощью шаблона записи.
- Организация обращения к элементам записи.

Для использования шаблона записи в программе необходимо определить переменную с типом данной записи, для чего применяется следующая синтаксическая конструкция:



Пример.

```
iotest    record                          ; Создание записи с именем iotest
     i1:1,i2:2=11,i3:1,i4:2=11,i5:2=00    ; Список компонент записи с указанием
числа бит для каждого
```

Целочисленные константы. Целочисленная константа (или целочисленный литерал) состоит из необязательного знака, одной или нескольких цифр и необязательного символа — суффикса (называемого основанием), который показывает, к какой системе счисления относится это число:

[{+ I -.}] цифры [основание]

Значения основания для разных типов чисел

Суффикс	Система счисления
h	Шестнадцатеричная
q или o	Восьмеричная
d или ничего	Десятичная
b	Двоичная
r	Закодированное вещественное число
t	Десятичная (альтернативная форма)
y	Двоичная (альтернативная форма)

Если основание в целочисленной константе не указано, предполагается, что число десятичное. Примеры:

26	Десятичное
26d	Десятичное
11010011b	Двоичное
42q	Восьмеричное
42o	Восьмеричное
1Ah	Шестнадцатеричное
0A3h	Шестнадцатеричное

Если шестнадцатеричная константа начинается с буквы, перед ней должеи ставиться символ нуля (0), чтобы ассемблер не воспринял эту константу как идентификатор. Хотя символ основания может быть и прописной буквой, рекомендуется использовать строчные буквы для унификации записи.

Вещественные константы. Существует два типа вещественнык констант: десятичные и закодированные (шестнадцатеричные). Десятинные вещественные константы состоят из необязательного знака, за которым следует одна или несколько цифр, десятичная точка и еще несколько цифр, выражающих дробную часть числа, а затем показатель степени:

[Знак]Цифры 1. [Цифры 2] [степень]

Ниже приведены определения понятий знаки степень:

- Знак – это плюс (+) или минус (-).
- Цифры 1 – целая часть мантиссы.
- Точка – разделитель целой и дробной частей мантиссы.
- Цифры 2 – дробная часть мантиссы. Может пропускаться.
- степень - E [+ или -]Цифры 3.
- E – символ.
- Цифры 3 – порядок..

Поле знака является необязательным, в котором может находиться математический знак + или -. Примеры правильных вещественных констант: 2, + 3.0, -44.2E+05, 26.E5.

В самом простейшем случае для определения вещественной константы достаточно указать цифру и десятичную точку. Без десятичной точки эту константу компилятор будет считать целой.

Закодированные вещественные константы. Вещественную константу можно также задать и в шестнадцатеричном виде в форме закодированного вещественного числа. Разумеется,для этого нужно знать точный формат представления вещественных чисел в двоичном виде.

Символьные констевнты — это последовательности символов, заключенные в одинарные или двойные кавычки. Ассемблер автоматически заменяет символьную константу на соответствующий ей ASCII-код. Вот несколько примеров: 'A', "d".

Строковые константы. Строковой константой называется последовательность символов, заключенных в одинарные или двойные кавычки. Вместо нее ассемблер автоматически подставляет последовательность ASCII-кодов, соответствующих каждому символу строковой константы. Вот несколько примеров: 'ABC', 'X', "Привет, Вася!"

3.14. Макросредства

Даже для маленьких по объему программ возникают некоторые из перечисленных здесь проблем:

- плохое понимание исходного текста программы, особенно по прошествии некоторого времени после ее написания;
- ограниченность набора команд;
- повторяемость некоторых идентичных или незначительно отличающихся участков программы;
- необходимость включения в каждую программу участков кода, которые уже были использованы в других программах.

Если бы мы писали программу на машинном языке, то данные проблемы были бы принципиально не решаемыми. Но язык ассемблера, являясь символическим аналогом машинного языка, предоставляет для их решения ряд средств.Основной целью, которая при этом преследуется, является повышение удобства написания программ. В общем случае эта цель достигается по нескольким направлениям за счет следующего:

- расширения набора директив;

- введения некоторых дополнительных команд, не имеющих аналогов в системе команд микропроцессора. За примером далеко ходить не нужно — команды setfield и getfield, которые скрывают от программиста рутинные действия и генерируют наиболее эффективный код;
- введения сложных типов данных.

Но это все глобальные направления, по которым развивается сам транслятор от версии к версии. Что же делать программисту для решения его локальной задачи, для облегчения работы в определенной проблемной области? Для этого разработчики компиляторов ассемблера включают в язык и постоянно совершенствуют **аппарат макросредств**. Этот аппарат является очень мощным и важным. В общем случае есть смысл говорить о том, что транслятор ассемблера состоит из двух частей — непосредственно транслятора, формирующего объектный модуль, и макроассемблера.

Если вы знакомы с языком C или C++, то конечно помните широко применяемый в них **механизм препроцессорной обработки**. Он является некоторым аналогом механизма, заложенного в работу макроассемблера. Для тех, кто ничего раньше не слышал об этих механизмах, поясню их суть. Основная идея — использование подстановок, которые замещают определенным образом организованную символьную последовательность другой символьной последова-

тельностью. Создаваемая таким образом последовательность может быть как последовательностью, описывающей данные, так и последовательностью программных кодов. Главное здесь то, что на входе макроассемблера может быть текст программы весьма далекий по виду от программы на языке ассемблера, а на выходе обязательно будет текст на чистом ассемблере, содержащем символические аналоги команд системы машинных команд микропроцессора.

Таким образом, обработка программы на ассемблере с использованием макросредств неявно осуществляется транслятором в две фазы. На первой фазе работает часть компилятора, называемая **макроассемблером**, функции которого на идейном уровне мы описали выше.На второй фазе трансляции работает непосредственно **ассемблер**, задачей которого является формирование объектного кода, содержащего текст исходной программы в машинном виде.

Далее мы обсудим основной набор макросредств, доступных при использовании компилятора TASM. Отметим, что большинство этих средств доступно и в компиляторе с языка ассемблера MASM фирмы Microsoft. Обсуждение начнем с простейших средств и закончим более сложными.

3.15. Директивы

Директивой называется команда, которая выполняется ассемблером во время трансляции исходного кода программы. Директивы ассемблера используются для определения логических сегментов, выбора модели памяти, определения переменных, создания процедур и т.п.

Директивы не генерируют машинных кодов. Они представляют собой инструкции клмпилятору, как осуществлять ассемблирование кода. Ассемблеру необходимо задать следующие данные:

- Используемый процессор.
- Условия ассемблирования.
- Условия управления.
- Правила генерации ошибок.
- Выбор процессора

Выбор разрешенного набора инструкций. Для выбранного типа инструкции процессоров старшего уровня запрещены.

Процессор	Разрешаемые инструкции
.386	80386 только непривилегированные
.386P	80386 включая привилегированные
387	80387 сопроцессор
.486	80486 только непривилегированные
.486P	80486 включая привилегированные
.586	80586 только непривилегированные
.586P	80586 включая привилегированные
.686P	80686 только непривилегированные
.686P	80686 включая привилегированные
.K3D	K3D
.MMX	MMX, SIMD
.XMM	Расширение SIMD для Интернет потоков

Размещение данных. Данные директивы позволяют определять в программе некоторое символическое_имя для данных. Для кажого определения в памяти данных выделяется область, размеры которой в байтах зависят от типа данных. Имя связано с адресом первого байта области. При трансляции ассемблер, встретив в коде имя, будет вместо него подставлять его определение.

Директива ALIGNE. Начальный номер размещения.

```
ALIGNE    номер  ; Задает номер, с которого начинается размещение данных.
```

Директива EVEN. Правило четного адреса.

```
EVEN  ; Делает четным текущий адрес.
```

Директива ORG. Создание счетчика для выражения.

```
ORG   выражение    ; Размещает в памяти счетчик для выражения
```

Директива LABEL. Создает метку с именем для типа.

Имя LABEL тип ; Метка с именем для данного типа. Размещение начиная с текущего адреса.

Директивы размещения данных разных типов.

Для численных данных определены типы ниже. Директива определения размещает и инициализирует задаваемое значение.

имя BYTE значение ; беззнаковый байт (1 байт), синоним DB.
имя SBYTE значение ; знаковый байт (1 байт).
имя WORD значение ; беззнаковое слово (2 байта), синоним DD.
имя SWORD значение ; знаковое слово (2 байта).
имя DWORD значение ; беззнаковое двойное слово (4 байта).
имя SDWORD значение ; знаковое двойное слово (4 байта).
имя FWORD значение ; беззнаковое слово (6 байт) , синоним DF.
имя QWORD значение ; беззнаковое квадрослово (8 байт) , синоним DQ.
имя SQWORD значение ; знаковое квадрослово (8 байт).
имя OWORD значение ; беззнаковое октослово (16 байт).
имя REAL4 значение ; переменная с ПТ с одинарной точностью (4 байта).
имя REAL8 значение ; переменная с ПТ с двойной точностью (8 байт).
имя REAL10 значение ; переменная с ПТ с повышенной точностью (10 байт).
имя TBYTE значение ; переменная с ПТ с повышенной точностью (10 байт) , синоним DT.

Строки. Директивы работы со строками.

Мнемоника	Действие
Имя CATSTR [текст1 [, текст2] …	Тексты объединяются в текст Имя.
Имя INSTR [позиция,] текст1 , текст2	В переменную Имя возвращается позиция первого появления текст2 в текст1. Сравнение начиянается с задаваемой позиции. Если позиция пропущена, то анализ с начала текст1.
Имя SUBSTR текст, позиция [,длина]	Возвращает в строку Имя подстроку заданной длины из текста, начиная с задаваемой позиции. Если длина не задана, то возвращается подстрока до конца текста.

Имя SIZESTR текст	Возвращает в переменную Имя длину текста.
FORC параметр, <строка> инструкции ENDM	Блок, который заменяет в строке символ параметр

Структуры и записи. В MASM определены сложные объекты:

- Записи. Это наборы полей переменной длины.
- Структуры. Это наборы иерархически организованных областей.
- Объединения. Это объекты, состоящие из разнородных объектов.

Директива RECORD. Объявление записи с именем ИмяЗаписи, состоящей из поименованных полей с шириной, определяемой выражением.

Синтакис:

```
    ИмяЗаписи RECORD                ; Объявление записи с именем ИмяЗаписи
        ИмяПоля:Ширина = выражение  ; Объявление поля записи
        ИмяПоля:Ширина = выражение  ; Объявление поля записи
```

Директива STRUCT. Синтакис:

```
    ИмяСтруктуры STRUCT             ; Объявление структуры с именем ИмяСтруктуры
        ДекларацияПоля              ; Объявление поля структуры
        ДекларацияПоля              ; Объявление поля структуры
    ИмяСтруктуры ENDS               ; Конец структуры
```

Директива UNION. Синтакис:

```
    ИмяОбъединения UNION            ; Объявление объединения с именем ИмяОбъединения
        ДекларацияПоля              ; Объявление поля объединения
        …
    ИмяОбъединения ENDS             ; Конец объединения
```

Директива TYPEDEF. Объявление нового типа, который эквивалентен заданному типу.

Синтакис:

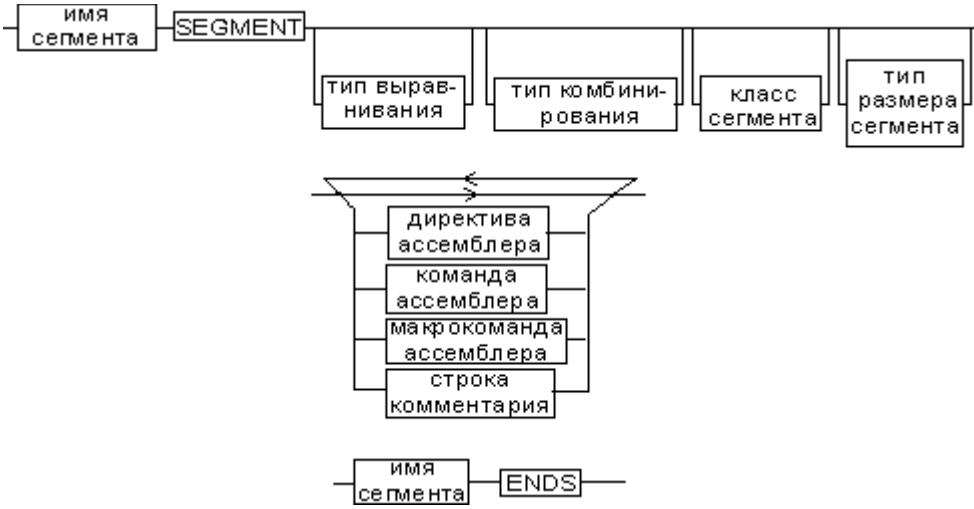
```
    ИмяТипа TYPEDEF тип             ; Объявление типа с именем ИмяТипа, который эквивалентен заданному типу.
```

- Стандартные директивы сегментации

Для архитектуры Фон-Неймана используется одно общее ОЗУ. В нем нужно размещать коды программ, константы, переменные и стек. Сегментация испольуеттся для выделения сегментов под эти клмпоненты. Предусмотрены две группы директив – общие и упрощенные.

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

Директива SEGMENT. Синтаксическое описание сегмента на ассемблере представляет собой конструкцию:



Первое предложение – это определение сегмента и его атрибутов. Последнее предложение – конец сегмента. Внутри размещаются содержательные предложения – директивы, команды, макрокоманды, комментарии.

Имя сегмента. Назначает сегменту символическое имя, определяемое содержанием сегмента (код, данные и др.)

Атрибут выравнивания сегмента (тип выравнивания) сообщает компоновщику о том, что нужно обеспечить размещение начала сегмента на заданной границе. Это важно, поскольку при правильном выравнивании доступ к данным в

процессорах выполняется быстрее. Допустимые значения этого атрибута следующие:

BYTE — выравнивание не выполняется. Сегмент может начинаться с любого адреса памяти;

WORD — сегмент начинается по адресу, кратному 2, то есть последний (младший) значащий бит физического адреса равен 0 (выравнивание на границу слова);

DWORD — сегмент начинается по адресу, кратному 4, то есть два последних (младших) значащих бита равны 0 (выравнивание на границу двойного слова);

PARA — сегмент начинается по адресу, кратному 16, то есть последняя шестнадцатеричная цифра адреса должна быть 0h (выравнивание на границу параграфа);

PAGE — сегмент начинается по адресу, кратному 256, то есть две последние шестнадцатеричные цифры должны быть 00h (выравнивание на границу 256-байтной страницы);

MEMPAGE — сегмент начинается по адресу, кратному 4 Кбайт, то есть три последние шестнадцатеричные цифры должны быть 000h (адрес следующей 4-Кбайтной страницы памяти).

По умолчанию тип выравнивания имеет значение PARA.

Атрибут комбинирования сегментов (комбинаторный тип) сообщает компоновщику, как нужно комбинировать сегменты различных модулей, имеющие одно и то же имя. Значениями атрибута комбинирования сегмента могут быть:

PRIVATE — сегмент не будет объединяться с другими сегментами с тем же именем вне данного модуля;

PUBLIC — заставляет компоновщик соединить все сегменты с одинаковыми именами. Новый объединенный сегмент будет целым и непрерывным. Все адреса (смещения) объектов, а это могут быть, в зависимости от типа сегмента, команды и данные, будут вычисляться относительно начала этого нового сегмента;

COMMON — располагает все сегменты с одним и тем же именем по одному адресу. Все сегменты с данным именем будут перекрываться и совместно использовать память. Размер полученного в результате сегмента будет равен размеру самого большого сегмента;

AT xxxx — располагает сегмент по абсолютному адресу параграфа (параграф — объем памяти, кратный 16; поэтому последняя шестнадцатеричная цифра адреса параграфа равна 0). Абсолютный адрес параграфа задается выражением xxx. Компоновщик располагает сегмент по заданному адресу памяти (это можно использовать, например, для доступа к видеопамяти или области ПЗУ), учитывая атрибут комбинирования. Физически это означает, что сегмент при загрузке в память будет расположен, начиная с этого абсолютного адреса параграфа, но для доступа к нему в соответствующий сегментный регистр должно быть загружено заданное в атрибуте значение. Все метки и адреса в определенном таким образом сегменте отсчитываются относительно заданного абсолютного адреса;

STACK — определение сегмента стека. Заставляет компоновщик соединить все одноименные сегменты и вычислять адреса в этих сегментах относительно регистра ss. Комбинированный тип STACK (стек) аналогичен комбинированному типу PUBLIC, за исключением того, что регистр ss является стандартным сегментным регистром для сегментов стека. Регистр sp устанавливается на конец объединенного сегмента стека. Если не указано ни одного сегмента стека, компоновщик выдаст предупреждение, что стековый сегмент не найден. Если сегмент стека создан, а комбинированный тип STACK не используется, программист должен явно загрузить в регистр ss адрес сегмента (подобно тому, как это делается для регистра ds).

По умолчанию атрибут комбинирования принимает значение PRIVATE.

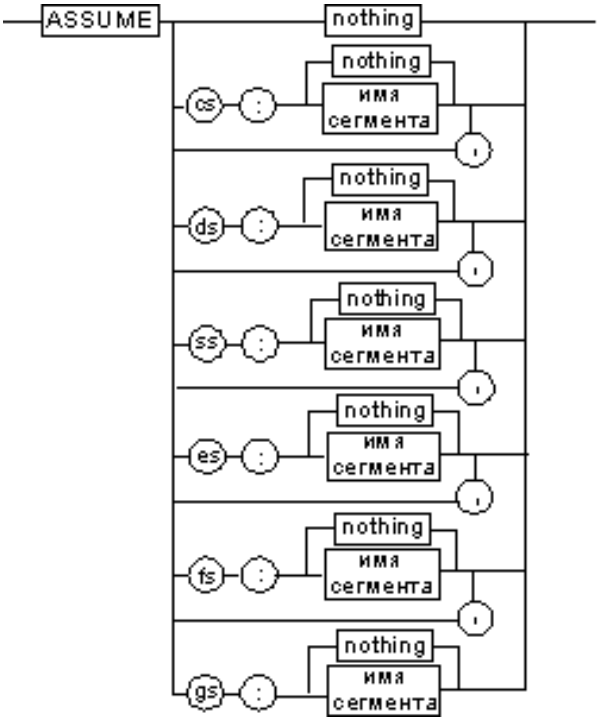
Атрибут класса сегмента (тип класса) — это заключенная в кавычки строка, помогающая компоновщику определить соответствующий порядок следования сегментов при собирании программы из сегментов нескольких модулей. Компоновщик объединяет вместе в памяти все сегменты с одним и тем же именем класса (имя класса, в общем случае, может быть любым, но лучше, если оно будет отражать функциональное назначение сегмента). Типичным примером использования имени класса является объединение в группу всех сегментов кода программы (обычно для этого используется класс “code”). С помощью механизма типизации класса можно группировать также сегменты инициализированных и неинициализированных данных;

Атрибут размера сегмента. Для процессоров i80386 и выше сегменты могут быть 16 или 32-разрядными. Это влияет, прежде всего, на размер сегмента и порядок формирования физического адреса внутри него. Атрибут может принимать следующие значения:

USE16 — это означает, что сегмент допускает 16-разрядную адресацию. При формировании физического адреса может использоваться только 16-разрядное смещение. Соответственно, такой сегмент может содержать до 64 Кбайт кода или данных;

USE32 — сегмент будет 32-разрядным. При формирования физического адреса может использоваться 32-разрядное смещение. Поэтому такой сегмент может содержать до 4 Гбайт кода или данных.

Директива ASSUME. Все сегменты сами по себе равноправны, так как директивы SEGMENT и ENDS не содержат информации о функциональном назначении сегментов. Для того чтобы использовать их как сегменты кода, данных или стека, необходимо предварительно сообщить транслятору об этом, для чего используют специальную директиву ASSUME, имеющую формат, показанный на рисунке. Эта директива сообщает транслятору о том, какой сегмент к какому сегментному регистру привязан. В свою очередь, это позволит транслятору корректно связывать символические имена, определенные в сегментах. Привязка сегментов к сегментным регистрам осуществляется с помощью операндов этой директивы, в которых имя_сегмента должно быть именем сегмента, определенным в исходном тексте программы директивой SEGMENT или ключевым словом nothing. Если в качестве операнда используется только ключевое слово nothing, то предшествующие назначения сегментных регистров аннулируются, причем сразу для всех шести сегментных регистров. Но ключевое слово nothing можно использовать вместо аргумента имя сегмента; в этом случае будет выборочно разрываться связь между сегментом с именем имя сегмента и соответствующим сегментным регистром.



Директива .ALPHA. Сегменты во внутренней таблице упорядочиваются по алфавиту.

Директива .SEQ. Сегменты во внутренней таблице упорядочиваются последовательно. **Директива по умолчанию**..

Директива .GROUP. Добавляет в поименованную группу перечисленные сегменты. Синтакис:

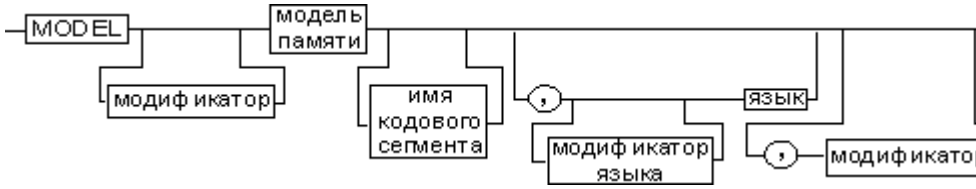
Имя GROUP сегмент [, сегмент […

Упрощенные директивы сегментации. Для простых программ, содержащих по одному сегменту для кода, данных и стека, хотелось бы упростить ее описание. Для этого в трансляторы MASM ввели возможность использования упрощенных директив сегментации. Но здесь возникла проблема, связанная с тем, что необходимо было как-то компенсировать невозможность напрямую управлять размещением и комбинированием сегментов. Для этого совместно с упрощенными

директивами сегментации стали использовать директиву указания модели памяти MODEL, которая частично стала управлять размещением сегментов и выполнять функции директивы ASSUME (поэтому при использовании упрощенных директив сегментации директиву ASSUME можно не использовать). Эта директива связывает сегменты, которые в случае использования упрощенных директив сегментации имеют предопределенные имена, с сегментными регистрами (хотя явно инициализировать ds все равно придется).

Синтаксис той или иной директивы зависит от используемой версии ассемблера и никак не связан с системой команд процессоров Intel. Разные ассемблеры могут генерировать идентичный машинный коддля системы команд процессоров Intel, но они могут поддерживать совершенно разный набор директив.

Директивы модели памяти. Синтаксис директивы MODEL.



Модель памяти. Обязательным параметром директивы MODEL является модель памяти. Этот параметр определяет модель сегментации памяти для программного модуля. В таблице приведены некоторые значения параметра модель памяти директивы MODEL.

Модель	Тип кода	Тип данных	Назначение модели
TINY	near	near	Код и данные объединены в одну группу с именем DGROUP. Используется для создания программ формата .com.
SMALL	near	near	Код занимает один сегмент, данные объединены в одну группу с именем DGROUP. Эту модель обычно используют для большинства программ на ассемблере
MEDIUM	far	near	Код занимает несколько сегментов, по одному на каждый объединяемый программный модуль. Все ссылки на передачу управления — типа far.

			Данные объединены в одной группе; все ссылки на них — типа near.
COMPACT	near	far	Код в одном сегменте; ссылка на данные — типа far.
LARGE	far	far	Код в нескольких сегментах, по одному на каждый объединяемый программный модуль.

Операнды директивы MODEL используют для задания модели памяти, которая определяет набор сегментов программы, размеры сегментов данных и кода, способ связывания сегментов и сегментных регистров.

Предполагается, что программный модуль может иметь только определенные типы сегментов, которые определяются упрощенными директивами описания сегментов. Эти директивы приведены в таблице.

Директива	Назначение
.CODE [имя]	Начало или продолжение сегмента кода (имя по умолчанию _TEXT).
.DATA	Начало или продолжение сегмента инициализированных данных. Также используется для определения данных типа near (имя _DATA).
.CONST	Начало или продолжение сегмента постоянных данных (констант) модуля.
.DATA?	Начало или продолжение сегмента неинициализированных данных. Также используется для определения данных типа near (имя _BSS).
.STACK [размер]	Начало или продолжение сегмента стека модуля. Параметр [размер] задает размер стека. Размер по умолчанию 1024 байта.
.FARDATA [имя]	Начало или продолжение сегмента инициализированных данных типа far (имя FAR_DATA).
.FARDATA? [имя]	Начало или продолжение сегмента неинициализированных данных типа far.

Операнд - имя кодового сегмента.

Наличие в некоторых директивах параметра [имя] говорит о том, что возможно определение нескольких сегментов этого типа. С другой стороны, наличие нескольких видов сегментов данных обусловлено требованием обеспечить совместимость с некоторыми компиляторами языков высокого уровня, которые создают **разные** сегменты данных для инициализированных и неинициализированных данных, а также констант.

При использовании директивы MODEL транслятор делает доступными несколько идентификаторов, к которым можно обращаться во время работы программы, с тем, чтобы получить информацию о тех или иных характеристиках данной модели памяти. Перечислим эти идентификаторы и их значения.

Имя идентификатора	Значение переменной
@code	Физический адрес сегмента кода.
@data	Физический адрес сегмента данных типа near.
@fardata	Физический адрес сегмента данных типа far.
@fardata?	Физический адрес сегмента неинициализированных данных типа far.
@curseg	Физический адрес сегмента неинициализированных данных типа fa.r
@stack	Физический адрес сегмента стека.

Операнд - язык. Применяктся в моделях с вызываемыми процедурами, когда вызывающая программа использует код, созданный в вызываемой программе. Операнд служит для того, чтобы компилятор мог правильно организовать интерфейс (связь) между процедурой на ассемблере и программой на языке высокого уровня. Необходимость такого указания возникает вследствие того, что способы передачи аргументов при вызове процедур различны для разных языков высокого уровня. При вызове подпрограммы аргумент загружаются в стек двумя способами:

- Слева направо.
- Справа налево

После обработки стек очищает одна из двух программ:

- Вызываемая.

- Вызывающая.

В таблице представлены данные о передаче аргументов в языках высокого уровня.

Операнд язык	Язык аргументов	Направление передачи стека	Процедура очистки стека
NOLANGUAGE	Ассемблер	Слева направо	Вызываемая
BASIC	Basic	Слева направо	Вызываемая
PROLOG	Prolog	Справа налево	Вызывающая
FORTRAN	Fortran	Слева направо	Вызываемая
C	C	Справа налево	Вызывающая
C++(CPP)	C++	Справа налево	Вызывающая
PASCAL	Pascal	Слева направо	Вызываемая

Операнд - модификатор языка.

Операнд – модификатор. Позволяет уточнить некоторые особенности использования выбранной модели памяти.

Значение модификатора	Назначение
use16	Сегменты выбранной модели используются как 16-битные (если соответствующей директивой указан процессор i80386 или i80486)
use32	Сегменты выбранной модели используются как 32-битные (если соответствующей директивой указан процессор i80386 или i80486)
dos	Программа будет работать в MS-DOS

Необязательные параметры язык и модификатор языка определяют некоторые особенности вызова процедур. Необходимость в использовании этих параметров появляется при написании и связывании программ на различных языках программирования.

Описанные нами стандартные и упрощенные директивы сегментации не исключают друг друга. Стандартные директивы используются, когда программист желает получить полный контроль над размещением сегментов в памяти и их комбинированием с сегментами других модулей.

Упрощенные директивы целесообразно использовать для простых программ и программ, предназначенных для связывания с программными модулями, написанными на языках высокого уровня. Это позволяет компоновщику эффективно связывать модули разных языков за счет стандартизации связей и управления.

Наблюдатели. Эти директивы регулируют доступность имен.

Мнемоника	Действие
COMM определ1 [, определ2]…	Создание общей переменной COMM с индивидуальными атрибутами, укзанными в определениях.
PUBLIC [язык] имя …	Задание имен переменных, меток или символов, доступных в других модулях программы.
EXTERN [язык] имя : тип…	Задание имен переменных, меток или символов, имя вызова которых – тип.
EXTERNDEF [язык] имя : тип…	Задание имен переменных, меток или символов, имя вызова которых – тип с определением доступности. Если на имя есть ссылки в модуле, то оно трактуется, как EXTERN, если нет, то как PABLIC.

Процедуры. Процедура представляет собой группу команд для решения конкретной подзадачи и обладает средствами получения управления из точки вызова задачи более высокого уровня и возврата управления в эту точку. В простейшем случае программа может состоять из одной процедуры. Другими словами, процедуру можно определить как правильным образом оформленную совокупность команд, которая, будучи однократно описана, при необходимости может быть вызвана в любом месте программы.

Процедура может размещаться в любом месте программы, но так, чтобы на нее случайным образом не попало управление. Если процедуру просто вставить в общий поток команд, то процессор воспримет команды процедуры как часть этогопотока и, соответственно, начнет выполнять эти команды. Учитывая это обстоятельство, есть следующие варианты размещения процедуры в программе:

- в начале программы (до первой исполняемой команды);
- в конце программы (после команды, возвращающей управление операционной системе);
- ii промежуточный вариант — внутри другой процедуры или основной программы (в этом случае необходимо предусмотреть обход процедуры с помощью команды безусловного перехода JMP);

- в другом модуле (библиотеке DLL).

Директива PROC. Создание процедуры. Синтаксис директивы:

имя PROC [тип вызова] [язык] [доступность] [пролог] [список регистров] [, параметр]…
Инструкции
имя ENDP

Аргументы:

- Тип вызова (NEAR – ближний, FAR - дальний)
- Язык, любой доступный (C, C++, PASCAL и др.).
- Доступность (PRIVATE, PUBLIC или EXPORT)
- Пролог. Аргументы запуска пролога процедуры.
- Список регистров, которые пролог резервирует в стек для восстановления после завершения.
- Параметры, ассмблер транслирует их с размещением в стеке.

Как обратиться к процедуре? Так как имя процедуры обладает теми же атрибутами, что и обычная метка в команде перехода, то обратиться к процедуре, в принципе, можно с помощью любой команды перехода. Но есть одно важное свойство, которое можно использовать благодаря специальному механизму вызова процедур.

Суть состоит в возможности сохранения информации о контексте программы в точке вызова процедуры. Под контекстом понимается информация о состоянии программы в точке вызова процедуры. В системе команд процессора есть две команды для работы с контекстом - CALL и RET.

Команда CALL осуществляет вызов процедуры (подпрограммы). Синтаксис команды:

CALL[модификатор] имя_процедуры

Команда CALL передает управление по адресу с символическим именем имя_процедуры, при этом в стеке сохраняется адрес возврата (то есть адрес команды, следующей после команды CALL).

Команда RET считывает адрес возврата из стека и загружает его в регистры CS и EIP/IP, тем самым возвращая управление на команду, следующую в программе за командой CALL Синтаксис команды:

RET [число]

Необязательный параметр [число] обозначает количество элементов, удаляемых из стека при возврате из процедуры.

Для команды CALL актуальна проблема организации ближних и дальних переходов. Это видно из формата команды, где присутствует параметр [модификатор]. Вызов процедуры командой CALL может быть внутрисегментным и межсегментным.

- При внутрисегментном вызове процедура находится в текущем сегменте кода (имеет тип near), и в качестве адреса возврата команда CALL сохраняет только содержимое регистра IP/EIP.
- При межсегментном вызове процедура находится в другом сегменте кода (имеет тип far), и для осуществления возврата команда CALL должна запомнить содержимое обоих регистров (CS и IP/EIP), при этом в стеке сначала запоминается содержимое регистра CS, затем — регистра IP/EIP.

Директива INVOKE. Обеспечивает удобный вызов процедур с параметрами, передающимися через стек. Основная ее задача — сформировать код, который, во-первых, размещает аргументы в стеке, во-вторых, вызывает процедуру и, в третьих, чистит стек после завершения работы процедуры. Вызов процедуры по адресу из выражения**.** Аргументы заносятся в стек или в регистры в зависимости от условий вызова. Синтаксис:

```
INVOKE    ИмяПроцедуры    аргумент1 [ ,аргумент2 ]…
```

Для INVOKE аргумент ИмяПроцедуры не должен быть опережающей ссылкой на адрес. Чтобы исключить подобные ситуации, существует парная для INVOKE директива PROTO, которая должна предшествовать INVOKE

Директива PROTO. Прототипы функции. Эта директива информирует ассемблер о количестве и типах аргументов, которые принимает процедура. Использование данной директивы позволяет ассемблеру выполнять проверку типов. Обычно все директивы PROTO для процедур собираются в начале исходного текста программы либо в отдельном включаемом файле. Синтаксис:

имя PROTO [расстояние] [язык] [, параметр] [: тип]…

Директива PROTO принимает аргументы:

- Аргумент **расстояние** (NEAR, FAR, NEAR16, NEAR32, FAR16 или FAR32) влияет на размер адреса, формируемого ассемблером для вызова процедуры. По умолчанию значение этого параметра определяется, исходя из текущей модели памяти и типа процессора.

- Аргумент **язык** для определения стиля и соглашения по вызову процедуры в качестве значения принимает имя языка (C, C++, PASCAL и др.).
- Аргумент **параметр** представляет собой последовательность перечисленных через запятую параметров процедуры. Исходя из этой информации, при вызове процедуры ассемблер преобразует последовательность параметров в последовательность команд Занесения в стек PUSH с формированием соответствующих адресов параметров процедуры в стеке.
- Аргумент **тип** — один из допустимых ассемблером простых типов данных. В качестве типа может быть указано слово VARARG. Оно предназначено для определения процедур с переменным числом аргументов. Тип VARARG указывается с последним параметром, заданным в директиве PROTO.
- Выражения

Псевдооператоры EQU и =. К простейшим макросредствам языка ассемблера можно отнести псевдооператоры EQU и "=" (равно). Эти псевдооператоры предназначены для присвоения некоторому выражению символического имени или идентификатора. Впоследствии, когда в ходе трансляции этот идентификатор встретится в теле программы, макроассемблер подставит вместо него соответствующее выражение.В качестве выражения могут быть использованы константы, имена меток, символические имена и строки в апострофах. После присвоения этим конструкциям символического имени его можно использовать везде, где требуется размещение данной конструкции.

Синтаксис псевдооператора **EQU**:

```
Имя   EQU     числовое_выражение
Имя   EQU     <строка>
```

Синтаксис псевдооператора **=**:

```
имя   =   числовое_выражение
```

С помощью EQU идентификатору можно ставить в соответствие как числовые выражения, так и текстовые строки, а псевдооператор “=” может использоваться только с числовыми выражениями.

Ассемблер всегда пытается вычислить значение строки, воспринимая ее как выражение. Для того чтобы строка воспринималась именно как текстовая, необходимо заключить ее в угловые скобки: <строка>. Кстати сказать, угловые скобки являются оператором ассемблера, с помощью которого транслятору сооб-

щается, что заключенная в них строка должна трактоваться как текст, даже если в нее входят служебные слова ассемблера или операторы.

Псевдооператор EQU удобно использовать для настройки программы на конкретные условия выполнения, замены сложных в обозначении объектов, многократно используемых в программе, более простыми именами и т. п.

TEXTEQU текст. Назначает тексту имя. Синтаксис псевдооператора **TEXTEQU**:

```
имя TEXTEQU текст
```

Макрокоманды – макросы. Идейно макрокоманда представляет собой дальнейшее развитие механизма замены текста. С помощью макрокоманд в текст программы можно вставлять последовательности строк (которые логически могут быть данными или командами) и даже более того — привязывать их к контексту места вставки.

Макрокоманда представляет собой строку, содержащую некоторое символическое имя — имя макрокоманды, предназначенную для того, чтобы быть замещенной одной или несколькими другими строками. Имя макрокоманды может сопровождаться параметрами.

Обычно программист сам чувствует момент, когда ему нужно использовать макрокоманды в своей программе. Если такая необходимость возникает и нет готового, ранее разработанного варианта нужной макрокоманды, то вначале необходимо задать ее шаблон-описание, который называют макроопределением.

Директива MACRO. Синтаксис макроопределения следующий:

```
имя        MACRO    список_формальных_аргументов
    тело макроопределения
ENDM
```

Где должны располагаться макроопределения? Есть три варианта:

- **В начале исходного текста программы** до сегмента кода и данных с тем, чтобы не ухудшать читабельноть программы. Этот вариант следует применять в случаях, если определяемые вами макрокоманды актуальны только в пределах одной этой программы.
- **В отдельном файле**.Этот вариант подходит при работе над несколькими программами одной проблемной области. Чтобы сделать доступными эти макроопределения в конкретной программе, необходимо в начале исходного текста этой программы записать директиву include имя_файла.

- **В макробиблиотеке**. Если у вас есть универсальные макрокоманды, которые используются практически во всех ваших программах, то их целесообразно записать в так называемую макробиблиотеку. Сделать актуальными макрокоманды из этой библиотеки можно с помощью все той же директивы include.

Функционально макроопределения похожи на процедуры. Сходство их в том, что и те, и другие достаточно один раз где-то описать, а затем вызывать их специальным образом. На этом их сходство заканчивается, и начинаются различия, которые в зависимости от целевой установки можно рассматривать и как достоинства и как недостатки:

- В отличие от процедуры, текст которой неизменен, макроопределение в процессе макрогенерации может меняться в соответствии с набором фактических параметров. При этом коррекции могут подвергаться как операнды команд, так и сами команды. Процедуры в этом отношении объекты менее гибки.
- При каждом вызове макрокоманды ее текст в виде макрорасширения вставляется в программу. При вызове процедуры микропроцессор осуществляет передачу управления на начало процедуры, находящейся в некоторой области памяти в одном экземпляре. Код в этом случае получается более компактным, хотя быстродействие несколько снижается за счет необходимости осуществления переходов.

Макроопределение обрабатывается компилятором особым образом. Для того чтобы использовать описанное макроопределение в нужном месте программы, оно должно быть активизировано с помощью макрокоманды указанием следующей синтаксической конструкции:

```
имя        список_фактических_аргументов
```

Результатом применения данной синтаксической конструкции в исходном тексте программы будет ее замещение строками из конструкции тела макроопределения. Но это не простая замена. Обычно макрокоманда содержит некоторый список аргументов — список_фактических_аргументов, которыми корректируется макроопределение. Места в теле макроопределения, которые будут замещаться фактическими аргументами из макрокоманды, обозначаются с помощью так называемых формальных аргументов. Таким образом, в результате применения макрокоманды в программе формальные аргументы в макроопределении замещаются соответствующими фактическими аргументами; в этом и заключа-

ется учет контекста. Процесс такого замещения называется макрогенерацией, а результатом этого процесса является макрорасширение.

Директива LOCAL. Метки в макросе. В макросе могут использоваться локальные метки. Синтаксис

```
    LOCAL     локал_имя                ; Локальная метки, уникальная для каждого вызова макроса
    LOCAL     метка  [ count :type ]   ; Метка, в стеке создаются count ячеек размером тип
```

Директива EXITM. Завершает макроблок и начинает ассемблирование следующей инструкции за пределами макроблока.

```
    EXITM [ текст ]     ; Завершить макрос, возможен вывод текста.
```

Директива PURG. Стирание макросов из памяти.

```
PURGE Макрос1, макрос2 … ; Стирание перечисленных макросов из памяти
```

Директива GOTO. Переход на ассемблирование макрометки. Прниеняется внутри блоков MACRO, FOR, FORC, REPEAT, и WHILE. Макрометка – директива.

```
    GOTO  [ макрометка ] ;  Переход на ассемблирование макрометки.
```

Директива ENDM. Конец макроса или потовряющего блока.

```
    ENDM  ;  Конец макроса.
```

- Макродирективы - циклы

С помощью макросредств ассемблера можно не только частично изменять входящие в макроопределение строки, но и модифицировать сам набор этих строк и даже порядок их следования. Сделать это можно с помощью набора макродирективов (далее — просто директив). Их можно разделить на две группы:

- директивы повторения WHILE, REPEAT (=REPT), IRP и IRPC. Директивы этой группы предназначены для создания макросов, содержащих тело из нескольких строк.
- директивы управления процессом генерации макрорасширения EXITM и GOTO. Они предназначены для управления процессом формирования макрорасширения из набора строк соответствующего макроопределения. С помощью этих директив можно как исключать отдельные строки из макрорасширения, так и вовсе прекращать процесс генерации. Директивы EXITM

и GOTO обычно используются вместе с условными директивами компиляции, поэтому они будут рассмотрены вместе с ними.

Директива WHILE применяется для повторения определенное количество раз тела цикла. Это цикл с предусловием. Директива имеют следующий синтаксис:

```
WHILE      выражение
    Тело цикла
ENDM
```

При использовании директивы WHILE макрогенератор транслятора будет повторять тело цикла до тех пор, пока значение выражения не станет равно 0. Это значение **вычисляется** перед очередной итерацией (и должно подвергаться изменению внутри тела цикла в процессе макрогенерации).

Директива REPEAT. Директива имеют следующий синтаксис:

```
REPEAT    выражение
    Тело цикла
ENDM
```

Отличие от WHILE состоит в том, что она автоматически уменьшает на 1 значение выражени после каждой итерации.

Директива .WHILE. Отличается от WHILE тем, что итерации управляются проверкой не выражения, а условия. Это цикл с предусловием. Директива имеют следующий синтаксис:

```
.WHILE условие
    Тело цикла
.ENDW
```

При использовании директивы .WHILE макрогенератор транслятора будет повторять тело цикла до тех пор, пока выполняется условие. Условие **проверяется перед** очередной итерацией (и должно подвергаться изменению внутри тела цикла в процессе макрогенерации).

Директива .REPEAT. Это цикл с постусловием. Директива имеют следующий синтаксис:

```
.REPEAT
    Тело цикла
.UNTIL условие
```

При использовании директивы .REPEAT макрогенератор транслятора будет повторять тело цикла до тех пор, пока не выполнится условие. Условие **проверяется после** очередной итерации (и должно подвергаться изменению внутри тела цикла в процессе макрогенерации).

Определены 2 директивы досрочного завершения итераций циклов:

- .BREAK. Завершить цикл.
- .CONTINUE. Завершить итерацию цикла.

Директивы имеют следующий синтаксис:

```
    .BREAK      .IF  условие    ; Завершить цикл, если условие выполняется.
    .CONTINUE   .IF  условие    ; Завершить итерацию цикла, если условие
выполняется.
```

- Директивы компиляции по условию

Данные директивы предназначены для организации выборочной трансляции фрагментов программного кода. Такая выборочная компиляция означает, что в макрорасширение включаются не все строки макроопределения, а только те, которые удовлетворяют определенным условиям. То, какие конкретно условия должны быть проверены, определяется типом условной директивы.

Введение в язык ассемблера этих директив значительно повышает его мощь. Всего имеется 10 типов условных директив компиляции. Их логично попарно объединить в четыре группы:

- Директивы IF и IFE — условная трансляция по результату вычисления логического выражения.
- Директивы IFDEF и IFNDEF — условная трансляция по факту определения символического имени.
- **Директивы IFB и** IFNB — условная трансляция по факту определения фактического аргумента при вызове макрокоманды.
- Директивы IFIDN, IFIDNI, IFDIF и IFDIFI — условная трансляция по результату сравнения строк символов.

Директивы *IF* и IFE. Условные директивы компиляции имеют общий синтаксис:

```
IFxxx   логическое_выражение
    фрагмент1
ELSE
    фрагмент2
ENDIF
```

Если в директиве IF логическое выражение истинно, то в объектный модуль помещается фрагмент1.
Если логическое выражение ложно, то при наличии директивы ELSE в объектный код помещается фрагмент2. Если же директивы ELSE нет, то вся часть программы между директивами IF и ENDIF игнорируется и в объектный модуль ничего не включается..

Директива IFE аналогично директиве IF анализирует значение логического_выражения. Но теперь для включения фрагмент_программы_1 в объектный модуль требуется, чтобы логическое_выражение имело значение “ложь”.

Директивы IFDEF и IFNDEF Синтаксис этих директив следующий:

```
IF(N)DEF      символическое_имя
    фрагмент1
ELSE
    фрагмент2
ENDIF
```

Данные директивы позволяют управлять трансляцией фрагментов программы в зависимости от того, определено или нет в программе некоторое символическое_имя. Директива IFDEF проверяет, описано или нет в программе символическое_имя, и если это так, то в объектный модуль помещается фрагмент1. В противном случае, при наличии директивы ELSE, в объектный код помещается фрагмент2. Если же директивы ELSE нет (и символическое_имя в программе не описано), то вся часть программы между директивами IF и ENDIF игнорируется и в объектный модуль не включается.

Действие IFNDEF обратно IFDEF. Если символического_имени в программе нет, то транслируется фрагмент1. Если оно присутствует, то при наличии ELSE транслируется фрагмент2. Если ELSE отсутствует, а символическое_имя в программе определено, то часть программы, заключенная между IFNDEF и ENDIF, игнорируется.

Директивы IFB и IFNB Синтаксис этих директив следующий:

```
IF(N)B      аргумент
    фрагмент1
ELSE
    фрагмент2
ENDIF
```

Данные директивы используются для проверки фактических параметров, передаваемых в макрос. При вызове макрокоманды они анализируют значение аргумента, и в зависимости от того, равно оно пробелу или нет, транслируется либо фрагмент1, либо фрагмент2. Какой именно фрагмент будет выбран, зависит от кода директивы:

- Директива **IFB** проверяет равенство **аргумента** пробелу. В качестве **аргумента** могут выступать **имя** или **число**.
  Если его значение равно **пробелу** (то есть фактический аргумент при вызове макрокоманды не был задан), то транслируется и помещается в объектный модуль **фрагмент1**. В противном случае, при наличии директивы **ELSE**, в объектный код помещается **фрагмент2**. Если же директивы **ELSE** нет, то при равенстве **аргумента пробелу** вся часть программы между директивами **IFB** и **ENDIF** игнорируется и в объектный модуль не включается.
- Действие **IFNB** обратно **IFB**. Если значение **аргумента** в программе не равно **пробелу**, то транслируется **фрагмент1**.
  В противном случае, при наличии директивы **ELSE**, в объектный код помещается **фрагмент2**. Если же директивы **ELSE** нет, то вся часть программы (при неравенстве **аргумента пробелу**) между директивами **IFNB** и **ENDIF** игнорируется и в объектный модуль не включается.

Директивы IFIDN, IFIDNI, IFDIF и IFDIFI. Эти директивы позволяют не просто проверить наличие или значение аргументов макрокоманды, но и выполнить идентификацию аргументов как строк символов.

Синтаксис этих директив:

```
IFIDN(I)     аргумент1,аргумент2
    фрагмент1
ELSE
    фрагмент2
ENDIF

IFDIF(I) аргумент1,аргумент2
    фрагмент1
ELSE
    фрагмент2
ENDIF
```

В директивах **аргумент1** и **аргумент2** сравниваются, как **строки символов**.

Директива **IFIDN**(I) сравнивает символьные значения **аргумент1** и **аргумент2**. Если результат сравнения положительный (строки совпадают), то **фрагмент1** транслируется и помещается в объектный модуль.В противном случае, при наличии директивы **ELSE**, в объектный код помещается **фрагмент2**. Если же директивы **ELSE** нет, то вся часть программы между директивами **IFIDN**(I) и **ENDIF** игнорируется и в объектный модуль не включается.

Действие **IFDIF**(I) обратно **IFIDN**(I). Если результат сравнения отрицательный (строки не совпадают), транслируется **фрагмент1**. В противном случае все происходит аналогично рассмотренным ранее директивам.

Парность этих директив объясняется тем, что они позволяют учитывать различие строчных и прописных букв. Так, директивы с символом I в конце **IFIDNI** и **IFDIFI** игнорируют это различие, а **IFIDN** и **IFDIF** — учитывают.

Эти директивы удобно применять для проверки фактических аргументов макрокоманд.

Допускается вложенность условных директив компиляции.

Директивы генерации ошибок. В языке есть ряд директив, называемых директивами генерации пользовательской ошибки. Их можно рассматривать и как самостоятельное средство, и как метод, расширяющий возможности директив условной компиляции. Они предназначены для обнаружения различных ошибок в программе, таких как неопределенные метки или пропуск параметров макроса.

Директивы генерации пользовательской ошибки по принципу работы можно разделить на два типа:

- безусловные директивы, генерирующие ошибку трансляции без проверки каких-либо условий;
- условные директивы, генерирующие ошибку трансляции после проверки определенных условий.

Большинство директив генерации ошибок имеют два обозначения, хотя принцип их работы одинаков. Второе название отражает их сходство с директивами условной компиляции. При дальнейшем обсуждении такие парные директивы будут приводиться в скобках.

Безусловная генерация пользовательской ошибки. К безусловным директивам генерации пользовательской ошибки относится только одна директива — это ERR (.ERR). Данная директива, будучи вставлена в текст программы, безус-

ловно приводит к генерации ошибки на этапе трансляции и удалению объектного модуля.

Условная генерация пользовательской ошибки. Набор условий, на которые реагируют директивы условной генерации пользовательской ошибки, такой же, как и у директив условной компиляции. Поэтому и количество этих директив такое же. К их числу относятся следующие директивы.

Директивы .ERRB (ERRIFB) и .ERRNB (ERRIFNB). Синтаксис директив:

```
    .ERRB (ERRIFB) <имя_формал_аргумента>          ; ошибка, если
<имя_формал_аргумента> пропущено;
    .ERRNB (ERRIFNB) <имя_формал_аргумента>        ; ошибка, если
<имя_формало_аргумента> присутствует.
```

Данные директивы применяются для генерации ошибки трансляции в зависимости от того, задан или нет при вызове макрокоманды фактический аргумент, соответствующий формальному аргументу в заголовке макроопределения с именем <имя_формал_аргумента>. По принципу действия эти директивы полностью аналогичны соответствующим директивам условной компиляции IFB и IFNB. Их обычно используют для проверки задания параметров при вызове макроса. Строка имя_формального_аргумента должна быть **заключена в угловые скобки**.

Директивы .ERRDEF (ERRIFDEF) и .ERRNDEF (ERRIFNDEF). Синтаксис директив:

```
    .ERRDEF (ERRIFDEF)    символическое_имя        ; ошибка, если имя
определено до выдачи директивы в программе
    .ERRNDEF(ERRIFNDEF) символическое_имя          ; ошибка, если имя не
определено до момента обработки транслятором
```

Данные директивы генерируют ошибку трансляции в зависимости от того, определено или нет некоторое символическое_имя. в программе. Не забывайте о том, что компилятор TASM по умолчанию формирует объектный модуль за один проход исходного текста программы. Следовательно, директивы .ERRDEF (ERRIFDEF) и .ERRNDEF (ERRIFNDEF) отслеживают факт определения символического_имени только в той части исходного текста, которая находится до этих директив.

Директивы .ERRDIF (ERRIFDIF) и .ERRIDN (ERRIFIDN). Синтаксис директив:

.ERRDIF (ERRIFDIF) <строка_1>,<строка_2> ; ошибка, если две строки посимвольно не совпадают.
.ERRIDN (ERRIFIDN) <строка_1>,<строка_2> ; ошибку, если строки посимвольно идентичны.

Для того чтобы игнорировать различия строчных и прописных букв, существуют аналогичные директивы:

.ERRIFDIFI <строка_1>,<строка_2> ; то же, что и ERRIFDIF, но игнорируется различие строчных и прописных букв
.ERRIFIDNI <строка_1>,<строка_2> ; то же, что и ERRIFIDN, но игнорируется различие строчных и прописных букв

Данные директивы, как и соответствующие им директивы условной компиляции, удобно применять для проверки передаваемых в макрос фактических параметров.

Директивы .ERRE (ERRIFE) и .ERRNZ (ERRIF). Синтаксис директив:

.ERRE (ERRIFE) константное_выражение ; ошибка, если константное_выражение ложно (равно нулю) .ERRNZ(ERRIF) константное_выражение ; ошибка, если константное_выражение истинно (не равно нулю).

Вычисление константного_выражения должно приводить к абсолютному значению и не может содержать компонентов, являющихся ссылками вперед.

Во многих условных директивах в формировании условия участвуют **выражения**. Результат вычисления этого выражения обязательно должен быть **константой**. Хотя его компонентами могут быть и символические параметры, но их сочетание в выражении должно давать абсолютный результат.

Управление листингом. Листинг – это текстовый файл, формируемый ассемблером при трансляции. Команды ниже управляют его содержанием.

Мнемоника	Действие
.CREF	Разрешает листинг в символьном формате из таблиц символов.
.NOCREF [[имя [[, имя]]...]]	Запрещает листинг символов из источников с перечисленными именами.
.LIST	Старт листинга инструкций. Директива по умолчанию.
.NOLIST Синоним .XLIST	Выключает листинг инструкций. Директива по умолчанию.

.LISTIF Синоним .LFCOND	Старт листинга из условных блоков , в которых условие не выполнено.
.NOLISTIF Синоним .SFCOND	Выключает листинг из условных блоков, в которых условие не выполнено.
.LISTMACRO Синоним .XALL	Старт лиситнга инструкций в макросах, которые генерируют коды или данные. Директива по умолчанию.
.NOLISTMACRO Синоним .SALL	Выключает листинг макросов
.LISTMACROALL Синоним .LALL	Старт листинга всех инструкций в макросах.
.LISTALL	Старт листинга всех инструкций. Комбинация .LIST, .LISTIF, LISTMACROALL
PAGE [[[[длина]], ширина]]	Определяет размер страницы листинга программного кода (длина и ширина). Если аргументов нет, то создается пустая страница.
PAGE +	Увеличивает на 1 номер страницы листнга программного кода.
TITLE текст	Определяет заголовок программного листинга.
SUBTITLE текст Синоним SBTTL	Определяет подзаголовок программного листинга.
.TFCOND	Переключает режим листинга из условных блоков.

4. Архитектура RISC

RISC (Reduced Instruction Set Computer) – компьютер с сокращённым набором команд. RISC характеризуется следующими свойствами:

- Фиксированная длина машинных инструкций (например, 32 бита) и простой формат команды.
- Специализированные команды для операций с памятью — чтения или записи. Операции вида «прочитать-изменить-записать» отсутствуют. Любые операции «изменить» выполняются только над содержимым регистров (архитектура load-and-store).
- Большое количество регистров общего назначения (32 и более).
- Отсутствие поддержки операций вида «изменить» над укороченными типами данных — байт, 16-битное слово. Так, например, система команд DEC Alpha содержала только операции над 64-битными словами, и требовала

разработки и последующего вызова процедур для выполнения операций над байтами, 16- и 32-битными словами.
- Отсутствие микропрограмм внутри самого процессора. То, что в CISC процессоре исполняется микропрограммами, в RISC процессоре исполняется как обыкновенный (хотя и помещённый в специальное хранилище) машинный код, не отличающийся принципиально от кода ядра ОС и приложений.

Типичные для RISC решения:

- **Спекулятивное исполнение**. При встрече с командой условного перехода процессор исполняет (или, по крайней мере, читает в кэш инструкций) сразу обе ветви до тех пор, пока не окончится вычисление управляющего выражения перехода. Позволяет отказаться от простоев конвейера при условных переходах.
- **Переименование регистров**. Каждый регистр процессора на самом деле представляет собой несколько параллельных регистров, хранящих несколько версий значения. Используется для реализации спекулятивного исполнения.

RISC процессор имеет повышенное быстродействие за счёт упрощения инструкций, чтобы их декодирование было более простым, а время выполнения — короче. Первые RISC процессоры даже не имели инструкций умножения и деления. Это также облегчает повышение тактовой частоты и делает более эффективной суперскалярность (распараллеливание инструкций между несколькими исполнительными блоками).

В первых архитектурах, причисляемых к RISC, большинство инструкций для упрощения декодирования имеют одинаковую длину и похожую структуру, арифметические операции работают только с регистрами, а работа с памятью идёт через отдельные инструкции загрузки (load) и сохранения (store). Эти свойства и позволили лучше сбалансировать этапы конвейеризации, сделав конвейеры в RISC значительно более эффективными и позволив поднять тактовую частоту.

Фокусирование на простых инструкциях и ведёт к архитектуре RISC, цель которой - сделать инструкции настолько простыми, чтобы они легко конвейеризировались и тратили не более одного такта на каждом шаге конвейера на высоких частотах.

Позднее было отмечено, что наиболее значимая характеристика RISC в разделении инструкций для обработки данных и обращения к памяти - обращение к памяти идёт только через инструкции load и store, а все прочие инструкции ог-

раничены внутренними регистрами. Это упростило архитектуру процессоров: позволило инструкциям иметь фиксированную длину, упростило конвейеры и изолировало логику, имеющую дело с задержками при доступе к памяти, только в двух инструкциях. В результате RISC-архитектуры стали называть также архитектурами load/store.

«Сокращённый набор команд» неверно понимается как минимизация количества инструкций в системе команд. В действительности, инструкций у многих RISC процессоров больше, чем у CISC процессоров. На самом деле сокращён объём (и время) работы для каждой отдельной инструкции - как максимум один цикл доступа к памяти. Сложные инструкции CISC-процессоров могут требовать сотен циклов доступа к памяти для своего выполнения.

Первая система, которая может быть названа RISC системой, это суперкомпьютер CDC 6600, который был создан в 1964 Сеймуром Крем. Позднее появилась шутка, что термин RISC на самом деле расшифровывается как «Really invented by Seymour Cray» («На самом деле придуман Сеймуром Крэем»).

Первая попытка создать RISC процессор на чипе была предпринята в IBM в 1975. Эта работа привела к созданию семейства процессоров IBM 801, которые был выпущен в форме чипа под именем ROMP в 1981. ROMP расшифровывается как Research OPD (Office Product Division) Micro Processor, то есть «Исследовательский МП». Затем последовало несколько исследовательских проектов, в результате одного из которых появилась система POWER.

После того, как процессоры архитектуры x86 были переведены на суперскалярную RISC архитектуру, можно сказать, что подавляющее большинство существующих ныне процессоров основаны на архитектуре RISC.

Архитектура MISC. MISC (Minimum Instruction Set Computer) – компьютер с минимальным набором команд. Дальнейшее развитие идей команды Чака Мура, который полагает, что принцип простоты, изначальный для RISC процессоров, слишком быстро отошёл на задний план. В пылу борьбы за максимальное быстродействие, RISC догнал и перегнал многие CISC процессоры по сложности. Архитектура MISC строится на **стековой** вычислительной модели с ограниченным числом команд.

5. Архитектура VLIW

Архитектуры VLIW (Very Long Instruction Word) используют очень длинное слово команды. Отличаются от суперскалярной архитектуры тем, что решение о распараллеливании принимается **компилятором на этапе генерации кода**, а не

аппаратурой на этапе исполнения. Команды очень длинны и содержат явные инструкции по распараллеливанию нескольких субкоманд на несколько устройств исполнения.

VLIW процессором в его классическом виде является Itanium. Разработка эффективного компилятора для VLIW является сложнейшей задачей. Преимущество VLIW перед суперскалярной архитектурой заключается в том, что компилятор может быть более развитым, нежели устройства управления процессора, и он способен хранить больше контекстной информации для принятия более верных решений по оптимизации.

VLIW - архитектура процессоров с несколькими вычислительными устройствами. Характеризуется тем, что одна инструкция процессора содержит несколько операций, которые должны выполняться параллельно. Задача распределения между ними работы решается **программно**

В суперскалярных процессорах также есть несколько вычислительных модулей, но задача распределения между ними работы решается **аппаратно**. Это сильно усложняет дизайн процессора, и может быть чревато ошибками. В процессорах VLIW задача распределения решается во время компиляции и в инструкциях явно указано, какое вычислительное устройство должно выполнять какую команду.

В отличие от суперскалярных машин, VLIW машины выдают на выполнение фиксированное количество команд, которые сформатированы либо как одна большая команда, либо как пакет команд фиксированного формата. Планирование работы VLIW-машины всегда осуществляется компилятором.

В VLIW машине полную ответственность за формирование пакета команд, которые могут выдаваться одновременно, несет компилятор, а аппаратура в динамике не принимает никаких решений относительно выдачи нескольких команд. Использование VLIW приводит в большинстве случаев к быстрому заполнению небольшого объема внутрикристальной памяти командами NOP (no operation), которые предназначены для тех устройств, которые не будут задействованы в текущем цикле.

В существующих VLIW разработках был найден большой недостаток, который был устранен делением длинных слов на более мелкие, параллельно поступающие к каждому устройству. Обработка множества команд независимыми устройствами одновременно является главной особенностью суперскалярной процессорной архитектуры.

VLIW можно считать логическим продолжением идеологии RISC, расширяющей её на архитектуры с несколькими вычислительными модулями. Так же, как в RISC, в инструкции явно указывается, что именно должен делать каждый модуль процессора. Из-за этого длина инструкции может достигать 128 или даже 256 бит.

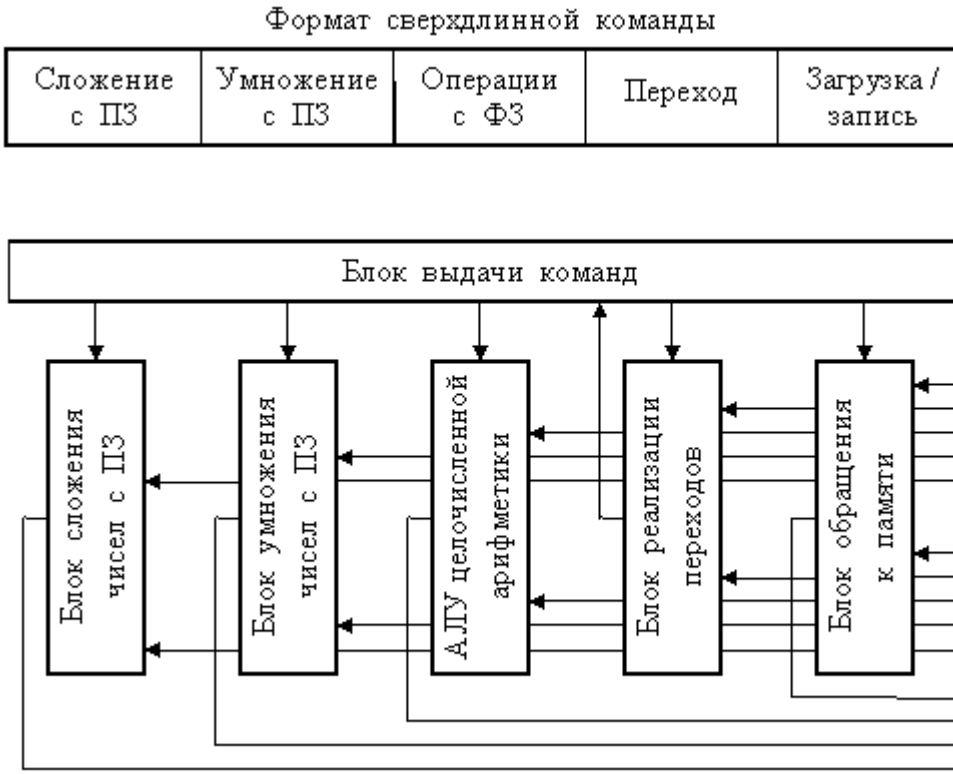
5.1. Архитектура вычислительных систем со сверхдлинными командами

Архитектура с командными словами сверхбольшой длины или со сверхдлинными командами (VLIW - Very Long Instruction Word) известна с начала 80-х годов. VLIW – это набор команд, организованных наподобие горизонтальной микрокоманды в микропрограммном устройстве управления.

Идея VLIW базируется на том, что задача эффективного планирования параллельного выполнения нескольких команд возлагается на «разумный» компилятор. Такой компилятор вначале исследует исходную программу с целью обнаружить все команды, которые могут быть выполнены одновременно без возникновения конфликтов. В процессе анализа компилятор может даже частично имитировать выполнение рассматриваемой программы. На следующем этапе компилятор пытается объединить такие команды в пакеты, каждый из которых рассматривается как одна сверхдлинная команда. Объединение нес-кольких 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, т.е. обеспечивается одновременное исполнение всех составляющих сверхдлинной команды.

Длина сверхдлинной команды обычно составляет от 256 до 1024 бит. Такая **метакоманда** содержит несколько полей (по числу образующих ее простых команд), каждое из которых описывает операцию для конкретного функционального блока. На рисунке показан возможный формат сверхдлинной команды и взаимосвязь между ее полями и ФБ, реализующими отдельные операции.



Каждое поле сверхдлинной команды отображается на свой функциональный блок, что позволяет получить максимальную отдачу от аппаратуры блока исполнения команд.

VLIW-архитектуру можно рассматривать как статическую суперскалярную архитектуру. Распараллеливание кода производится на этапе компиляции, а не динамически во время исполнения. То, что в выполняемой сверхдлинной команде исключена возможность конфликтов, позволяет предельно упростить аппаратуру VLIW-процессора и добиться более высокого быстродействия.

В качестве простых команд, образующих сверхдлинную, обычно используются команды RISC-типа, поэтому архитектуру VLIW иногда называют пост-RISC-архитектурой. Максимальное число полей в сверхдлинной команде равно

числу вычислительных устройств и обычно колеблется в диапазоне от 3 до 20. Все вычислительные устройства имеют доступ к данным, хранящимся в едином многопортовом регистровом файле. Отсутствие сложных аппаратных механизмов, характерных для суперскалярных процессоров (предсказание переходов, внеочередное исполнение и т.д.) дает значительный выигрыш в быстродействии и возможность более эффективно использовать площадь кристалла. Подавляющее большинство цифровых сигнальных процессоров и мультимедийных процессоров с производительностью более 1 млрд операций/с базируется на VLIW-архитектуре. Серьезная проблема VLIW – усложнение регистрового файла и связей этого файла с вычислительными устройствами.

5.2. Архитектура IA-64

Дальнейшим развитием идеи VLIW стала новая архитектура IA-64 – совместная разработка фирм Intel и Hewlett-Packard (IA – это аббревиатура от Intel Architecture). В IA-64 реализован новый подход, известный как вычисления с явным параллелизмом команд (EPIC, Explicitly Parallel Instruction Computing) и являющийся усовершенствованным вариантом технологии VLIW. Первым представителем данной стратегии стал микропроцессор Itanium компании Intel. Корпорация Hewlett-Packard также реализует данный подход в своих разработках.

В архитектуре IA-64 предполагается наличие в процессоре 128 64-разрядных регистров общего назначения (РОН) и 128 80-разрядных регистров с плавающей запятой. Кроме того, процессор IA-64 содержит 64 однобитовых регистра предикатов.

Формат команд в архитектуре IA-64:

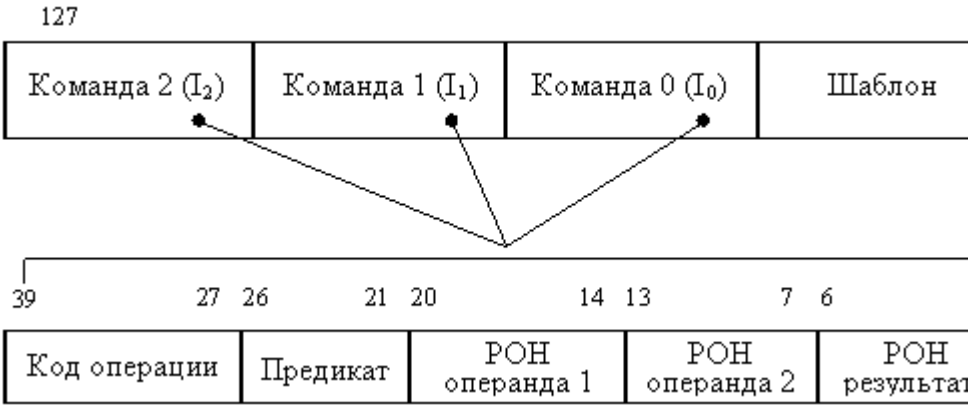


Рис. 10.2. Формат сверхдлинной команды в архитектуре IA-64

Команды упаковываются (группируются) компилятором в сверхдлинную команду – связку (bundle) длиною в 128 разрядов. Связка содержит три команды и шаблон, в котором указываются зависимости между командами (можно ли с командой I0 запустить параллельно I1, или же I1 должна выполняться только после I0), а также между другими связками (можно ли с командой I2 из связки S0 запустить параллельно команду I3 из связки S1).

Перечислим все варианты составления связки из трех команд:

- I0 ‖ I1 ‖ I2 – все команды исполняются параллельно;
- I0 & I1 ‖ I2 – сначала I0, затем исполняются параллельно I1 и I2;
- I0 ‖ I1 & I2 – параллельно обрабатываются I0 и I1, после них – I2;
- I0 & I1 & I2 – команды исполняются в последовательности I0, I1, I2 .

Одна связка, состоящая из трех команд, соответствует набору из трех функциональных блоков процессора. Процессоры IA-64 могут содержать разное количество таких блоков, оставаясь при этом совместимыми по коду. Благодаря тому что в шаблоне указана зависимость и между связками, процессору с N одинаковыми блоками из трех ФБ будет соответствовать сверхдлинная команда из N × 3 команд (N связок). Тем самым обеспечивается масштабируемость IA-64.

Поле каждой из трех команд в связке состоит из пяти полей:

- 13-разрядного поля кода операции;

- 6-разрядного поля предикатов, хранящего номер одного из 64 регистров предиката;
- 7-разрядного поля первого операнда (первого источника), где указывается номер регистра общего назначения или регистра с плавающей запятой, в котором содержится первый операнд;
- 7-разрядного поля второго операнда (второго источника), где указывается номер регистра общего назначения или регистра с плавающей запятой, в котором содержится второй операнд;
- 7-разрядного поля результата (приемника), где указывается номер регистра общего назначения или регистра с плавающей запятой, куда дол-жен быть занесен результат выполнения команды.

Предикация – это способ обработки условных ветвлений. Если в исходной программе встречается условное ветвление (по статистике через каждые 6 команд), то команды из разных ветвей помечаются разными регистрами предиката (команды имеют для этого соответствующие поля), далее они выполняются совместно, но их результаты не записываются, пока значения регистров предиката (РП) не определены. Когда вычисляется условие ветвления, РП, соответствующий «правильной» ветви, устанавливается в 1, а другой – в 0. Перед записью результатов процессор проверяет поле предиката и записывает результаты только тех команд, поле предиката которых указывает на РП с единичным значением.

Предикаты формируются как результат сравнения значений, хранящихся в двух регистрах. Результат сравнения («Истина» или «Ложь») заносится в один из РП, но одновременно с этим во второй РП записывается инверсное значение полученного результата. Такой механизм позволяет процессору более эффективно выполнять конструкции типа IF-THEN-ELSE.

Логика выдачи команд на исполнение сложнее, чем в традиционных процессорах типа VLIW, но намного проще, чем у суперскалярных процессоров с неупорядоченной выдачей. Особенностями архитектуры EPIC являются:

- большое количество регистров;
- масштабируемость архитектуры до большого количества функциональных блоков, т.е. наследственно масштабируемая система команд (ISIS - Inherently Scaleable Instruction Set);
- 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.

Преимущества технологии VLIW. Использование компилятора позволяет устранить зависимость между командами до того, как они будут реально выполняться, в отличие от суперскалярных процессоров, где такие зависимости приходится обнаруживать и устранять «на лету». Отсутствие зависимостей между командами в коде, сформированном компилятором, ведет к упрощению аппаратных средств процессора и за счет этого к существенному подъёму его быстродействия. Наличие множества функциональных блоков дает возмож-ность выполнять несколько команд параллельно.

Недостатки технологии VLIW. Требуется новое поколение компиляторов, способных проанализировать программу, найти в ней независимые команды, связать такие команды в строки длиной от 256 до 1024 бит, обеспечить их параллельное выполнение. Компилятор должен учитывать конкретные детали аппаратных средств. При определенных ситуациях программа оказывается недостаточно гибкой.

Основные сферы применения. VLIW-процессоры пока еще мало распространены. Основными сферами применения технологии VLIW являются цифровые сигнальные процессоры и вычислительные системы, ориентированные на архитектуру IA-64. Наиболее известной была VLIW-система фирмы Multiflow Computer, Inc. В России VLIW-концепция была реализована в суперкомпьютере Эльбрус 3-1 и получила дальнейшее развитие в его последователе – Эльбрус-2000 (E2k). К VLIW относится и семейство сигнальных процессоров TMS320C6x фирмы Texas Instruments. В начале 2000 года фирма Transmeta заявила процессор Crusoe, представляющий собой программно-аппаратный комплекс. В нем команды микропроцессоров серии x86 транслируются в слова VLIW длиной 64 или 128 бит. Оттранслированные команды хранятся в кэш-памяти, а трансляция при многократном их использовании производится только один раз. Ядро процессора исполняет элементы кода в строгой последовательности.

5.3. Itanium

В чем заключается революционность Itanium?

Сама Intel по праву называет Itanium «самой значительной новой разработкой Intel в области микропроцессорной архитектуры с момента выпуска процессора

i386 в 1985 году». Революционность состоит в отказе от давно морально устаревшей системы команд x86 (она в ходу с 1978 года) и в радикальном переходе к новой архитектуре, свободной от «пережитков прошлого».

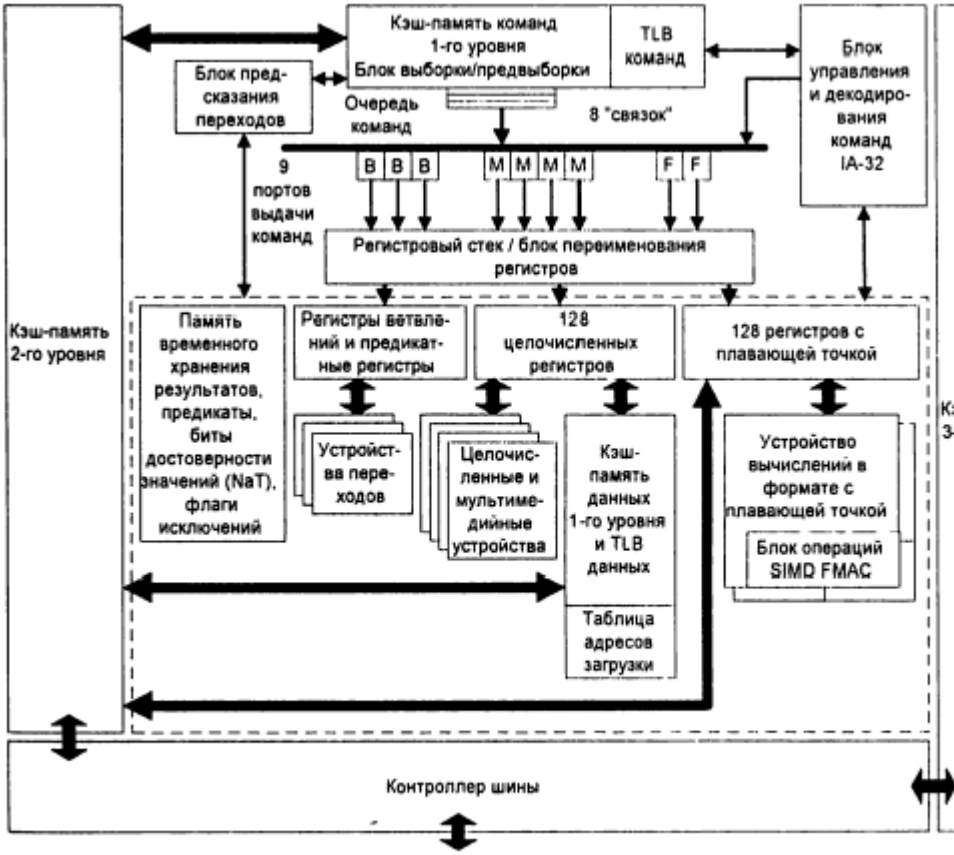
Революционной в Itanium является не только 64-разрядность (у 32-разрядных процессоров теоретическое ограничение объема адресуемой оперативной памяти составляет 4 Гбайт, а у 64-битных — несколько терабайт), но и явный параллелизм EPIC (Explicity Parallel Instruction Computing).

По сути, архитектура IA-64 (именно так называется технология, по которой проектируются процессоры Itanium, McKinley и последующие) впитала в себя все лучшие идеи: VLIW (Very Long Instruction Word, архитектура с длинными командами), устранение ветвлений, улучшенный механизм предварительной подачи данных и пр.

Среди характеристик первого процессора архитектуры IA-64 можно также отметить увеличенное адресное пространство, обнаружение и исправление ошибок. В принципе, о силиконовой составляющей процессора Itanium давно уже известно многое.

Думаю, что к моменту выхода эта информация будет разниться только в деталях: станет известна точная частота (Intel может, к примеру, выдать не обещанные 800, а все 1000 МГц) и будет определена ценовая политика. Картридж Itanium предназначен для установки в Slot M — комбинированный процессорный разъем, сочетающий достоинства как Socket, так и Slot.

Сигнальная матрично-штырьковая часть разведена с силовой частью, по которой подается питание, с тем чтобы исключить помехи. На обратной стороне процессорного картриджа расположена массивная теплоотводная пластина, позволяющая равномерно распределять по всей поверхности процессора ватты, излучаемые в воздух. К тому же сильно нагревающиеся блоки процессора тоже размещены равномерно. Согласно предварительным данным Itanium поддерживает частоту шины памяти 266 МГц. Архитектура Itanium подразумевает использование 2 или 4 Мбайт кэш-памяти третьего уровня. Статическая кэш-память новой конструкции размещена на одной плате с ядром процессора и работает с ним на одинаковой тактовой частоте.



Структура микропроцессора Itanium

Процессор Itanium, предназначенный для корпоративных серверов и рабочих станций самого высокого класса, не похож ни на одно из изделий, ранее выпущенных фирмой Intel. Разработчики этой модели отказались, наконец, от 32-разрядного набора команд x86, который неизменно реализовывали микросхемы Intel с тех пор, как в 1985 г. дебютировал процессор 386. Itanium построен на базе абсолютно нового 64-разрядного набора команд, известного под названием IA-64.

В двух словах, набор команд IA-64 обеспечивает возможность работать с 64-разрядными регистрами и 64-разрядными каналами передачи данных. Главное преимущество такой архитектуры состоит в том, что, поскольку она позволяет использовать 64-разрядные (а не 32-разрядные) адреса для обращения к каждой ячейке памяти, совокупное адресуемое пространство памяти составляет 264 бит (т. е. 18 млрд. Гбайт, или 18 Эбайт). Правда, пока что Itanium не выходит на столь высокие показатели, хотя уже сегодня он обеспечивает возможность обращения к 16 Тбайт памяти (для сравнения: 32-разрядные микросхемы ограничены объемом 4 Гбайт). Ну, а если процессор может работать с такими гигантскими объемами памяти, то он способен лучше справляться с обработкой крупных массивов данных при выполнении таких задач, как добыча информации.

Itanium обеспечивает функционирование современных 32-разрядных программ, но в полной мере реализовать заложенные в новой архитектуре преимущества смогут только приложения, специально разработанные для этой платформы. "Приложения большого объема нужно будет перекомпилировать для Itanium, - считает Кевин Круэлл, старший эксперт специализирующейся на рынке ЦП исследовательской фирмы MicroDesign Resources (Саннивейл, шт. Калифорния). - Этот процессор обрабатывает 32-разрядный код просто с черепашьей скоростью".

6. Многоядерные архитектуры

Существуют одно- и многоядерные процессоры с параллельным выполнением некоторых операций, встречаются также системы, в которых несколько процессоров работают над одной задачей параллельно. Рассмотрим сначала одноядерный процессор.

«Процессорное ядро» (как правило, для краткости его называют просто «ядро») — это конкретное воплощение (микро)архитектуры (т.е. архитектуры в «**аппаратном**» смысле), являющееся стандартом для целой серии процессоров. Например, K10 — это микроархитектура, которая лежит в основе многих сегодняшних процессоров AMD: Athlon II, Phenom, Phenom II, Opteron. Микроархитектура задаёт общие принципы: «средний» по длине конвейер, исполнение до трёх команд за такт, предсказание переходов и внеочередное исполнение, и прочие «глобальные» особенности. Ядро — более конкретное воплощение. Например, процессоры микроархитектуры K10 с двумя ядрами, без поддержки многопроцессорности и кэша L3, с шиной HyperTransport частотой в 2 ГГц — это более-менее полное описание ядра Regor для Athlon II.

Можно сказать что «ядро» — это конкретное воплощение определённой микроархитектуры «в кремнии», обладающее (в отличие от самой микроархитектуры) набором строго обусловленных характеристик. Микроархитектура — аморфна, она описывает **общие принципы** построения процессора. Ядро — микроархитектура, «обросшая» всевозможными параметрами и характеристиками. Чрезвычайно редки случаи, когда процессоры сменяли микроархитектуру, сохраняя название. И, наоборот, практически любое наименование процессора хотя бы несколько раз за время своего существования «меняло» ядро. Например, общее название серии процессоров AMD — «Athlon 64» — это одна микроархитектура (K8), но целых 13 ядер — от Sledgehammer (2003) до Huron (2009). Разные ядра, построенные на одной микроархитектуре, могут иметь в том числе разное быстродействие.

За последнее десятилетие удельная производительность процессоров в пересчете на число транзисторов упала на один-два порядка. Дальнейшее развитие полупроводниковых технологий не может компенсировать неэффективность современных процессоров. Показатели быстродействия процессоров (в частности, тактовые частоты) достигли практически граничных показателей, плотность энергии увеличивается пропорционально уменьшению размеров транзисторов, и, соответственно, увеличиваются проблемы с теплоотводом.

Если нельзя использовать все возможности на одном ядре из-за исключительной сложности такого ядра, то следует пойти по пути увеличения числа ядер. Именно так поступили в Sun Microsystems, выпустив 8-ядерный процессор. Niagara. Сторонники EPIC-подхода также склонились к многоядерному решению. На форуме IDF осенью 2004 года Пол Отеллини, генеральный директор Intel, заявил: Мы связываем наше будущее с многоядерными продуктами; мы верим, что это ключевая точка перегиба для всей индустрии.

Можно говорить о двух заметно разнящихся между собой тенденциях в процессе увеличения числа ядер.

- **Мультиядерность** (multi-core). В этом случае предполагается, что ядра являются высокопроизводительными и их относительно немного; сейчас их число — два-четыре. Основных недостатков этого подхода два: первый — высокое энергопотребление, второй — высокая сложность чипа и, как следствие, низкий процент выхода готовой продукции. При производстве 8-ядерного процессора IBM Cell только 20% производимых кристаллов являются годными.
- Другой путь — **многоядерность** (many-core). В таком случае на кристалле собирается на порядок большее число ядер, но имеющих более простую

структуру и потребляющих миливатты мощности. Сейчас количество ядер варьируется от 40 до 200, можно ожидать появления процессоров с тысячами и десятками тысяч ядер.

Многоядерные процессоры, если все сводится к размещению большего числа простых ядер на одной подложке, нельзя воспринимать как решение всех проблем. Многоядерные процессоры чрезвычайно сложно программировать, они могут быть эффективны только на приложениях, обладающих естественной многопоточностью.

IBM предлагает архитектуру Power7, которая приходит на смену Power6, предназначавшейся для Unix-серверов корпорации. Архитектура Power7 является для IBM заметным шагом вперед. От 2 ядерных моделей корпорация переходит к 4, 6 и 8 ядрам, и каждое из них способно выполнять четыре потока команд одновременно. Процессоры Power7 должны выйти в 2010. Они будут изготавливаться по 45-нанометровому технологическому процессу и, как утверждают в IBM, их можно будет устанавливать в нынешних серверах моделей Power 570 и Power 595.

Fujitsu представляет 8 ядерный процессор SPARC64 — новую версию 4-ядерного процессора SPARC64 VII. Процессоры SPARC64 используются в серверах Fujitsu и Sun Microsystems.

AMD применяет в блейд-серверах процессоры Magny-Cours. Это 12-ядерные процессоры, которые объединяют на одной микросхеме два 6 ядерных процессора, соединенных шиной AMD Hyper Transport. Название им взяли у известной гоночной трассы во Франции. Выход Magny-Cours запланирован на 2010.

Intel работает над 8 ядерными двухпоточными процессорами Nehalem-EX, выход которых назначен на 2010. Ожидается появление Tukwila — 4-ядерной версии процессора Itanium.

Но вопрос в том, в какой степени современное программное обеспечение — и, следовательно, конечные пользователи — способно использовать возможности многоядерных процессоров. Приложения должны быть написаны так, чтобы решаемые ими задачи можно было разбивать на подзадачи, выполняемые на нескольких ядрах параллельно.

7. Микроконтроллер AVR от Atmel

7.1. Архитектура AVR от Atmel

AVR - самая обширная производственная линия среди других флэш-микроконтроллеров корпорации Atmel. Atmel представила первый 8-разрядный флэш-микроконтроллер в 1993 году и с тех пор непрерывно совершенствует технологию. Прогресс данной технологии наблюдался в

- снижении удельного энергопотребления (мА/МГц),
- расширении диапазона питающих напряжений (до 1.8 В) для продления ресурса батарейных систем,
- увеличении быстродействия до 16 млн. операций в секунду,
- встраивании эмуляции в реальном масштабе времени,
- реализации функции самопрограммирования,
- совершенствовании и расширении количества периферийных модулей,
- встраивании специализированных устройств (радиочастотный передатчик, USB-контроллер, драйвер ЖКИ, программируемая логика, контроллер DVD, устройства защиты данных) и др.

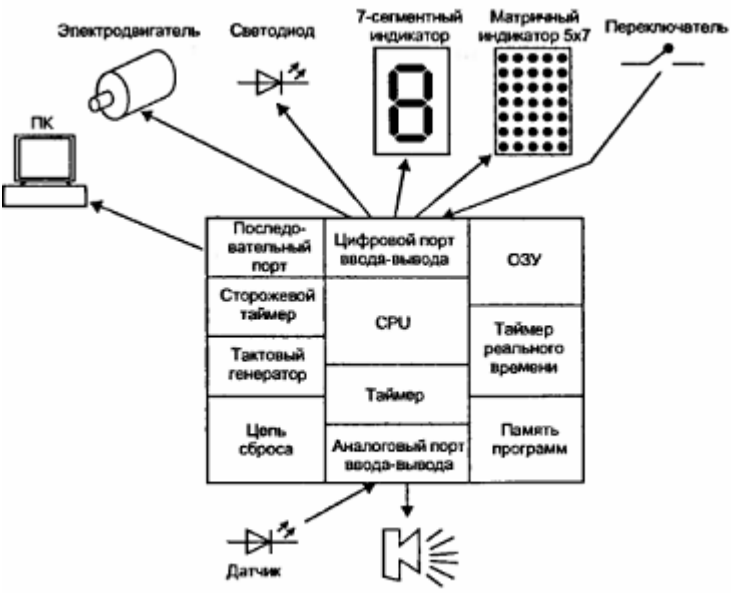
Успех AVR-микроконтроллеров объясняется возможностью простого выполнения проекта с достижением необходимого результата в кратчайшие сроки, чему способствует доступность большого числа инструментальных средств проектирования, поставляемых, как непосредственно корпорацией Atmel, так и сторонними производителями. Ведущие сторонние производители выпускают полный спектр компиляторов, программаторов, ассемблеров, отладчиков, разъемов и адаптеров. Отличительной чертой инструментальных средств от Atmel является их невысокая стоимость.

Другая особенность AVR-микроконтроллеров, которая способствовала их популярности, это использование RISC-архитектуры с мощным набором инструкций, большинство которых выполняются за один машинный цикл.

Это означает, что при равной частоте тактового генератора они обеспечивают производительность в 12 (6) раз больше производительности предшествующих микроконтроллеров на основе CISC-архитектуры (например, MCS51). С другой стороны, в рамках одного приложения с заданным быстродействием, AVR-микроконтроллер может тактироваться в 12 (6) раз меньшей тактовой частотой, обеспечивая равное быстродействие, но при этом потребляя гораздо меньшую мощность. Таким образом, AVR-микроконтроллеры представляют более широкие возможности по оптимизации производительности/энергопотребления, что

особенно важно при разработке приложений с батарейным питанием. Микроконтроллеры обеспечивает производительность до 16 млн. оп. в секунду и поддерживают флэш-память программ различной емкости: 1… 256 кбайт. AVR-архитектура оптимизирована под язык высокого уровня Си, а большинство представителей семейства megaAVR содержат 8-канальный 10-разрядный АЦП, а также совместимый с IEEE 1149.1 интерфейс JTAG или debugWIRE для встроенной отладки. Кроме того, все микроконтроллеры megaAVR с флэш-памятью емкостью 16 кбайт и более могут программироваться через интерфейс JTAG.

Пример использования



Структурная схема микроконтроллера

7.1.1. Основные параметры

- Гарвардская архитектура.
- RISC.
- Количество команд 90…130.
- Число выводов 8…64.
- Частота тактового генератора 10…16 МГц.

- Большинство инструкций выполняются за 1 цикл тактового генератора ТГ.
- Производительность 10…16 MIPS (Millions Instructions per Second).
- Память программ типа FLASH ROM. Перепрограммируется до 1000 раз.
- Память данных ОЗУ (тип EEPROM). Перепрограммируется до 100000 раз.
- 32 регистра общего назначения.
- Есть режимы с пониженным энергопотреблением.
- Отладчик AVR Studio, бесплатный.

7.1.2. Семейства

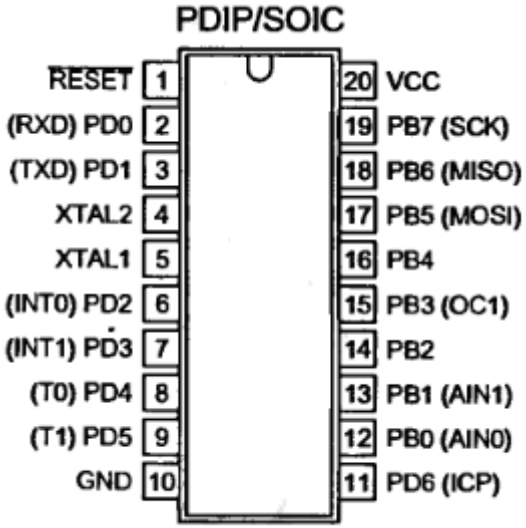
- Tiny AVR – миниатюрные МК. Flash ROM 1…2 Кбайт, ОЗУ типа EEPROM 64 байт
- Classic AVR. Flash ROM 2…8 Кбайт, ОЗУ типа EEPROM 64…512 байт, ОЗУ типа SRAM 128…512 байт
- Mega AVR Flash ROM 2…128 Кбайт, ОЗУ типа EEPROM 64…512 байт, ОЗУ типа SRAM 2…4 Кбайт, 10 разрядный 8-и канальный АЦП, аппаратный умножитель 8*8

Микроконтроллер AT90S2313 выбран как пример для изучения основ. Это современный 8-битный КМОП МК. Он имеет производительность 1 MIPS при частоте ТГ в 1 МГц, так как почти все его команды выполняются за 1 период ТГ.

Используется расширенная RISC архитектура от ARM, 32 регистра общего назначения. Все регистры подключены к арифметико-логическому устройству АЛУ, что дает доступ к 2-м регистрам в течение 1 цикла. Его основные характеристики:

- 2 Кб загружаемой FLASH памяти. Может быть перепрограммирования через интерфейс SPI.
- 128 байт EEPROM для данных.
- 15 линий ввода/вывода общего назначения.
- 2 таймера/счетчика (один 8-разрядный, другой 16-разрядный.).
- Внешние и внутренние прерывания
- Встроенный последовательный порт.
- Программируемый сторожевой таймер со встроенным генератором.
- Последовательный порт SPI для загрузки программ.
- 2 выбираемых программно режима низкого энергопотребления.

7.1.3. Описание выводов



Выводы микроконтроллера AT90S2313

- VCC – питание.
- GND – земля.
- PORT B (PB7…PB0) – 8-разрядный параллельный порт ввода/вывода. Выводы PBO, PB1 являются также положительным (A1N0) и отрицательным (A1N1) входами встроенного аналогового компаратора.
- PORT D (PD6…PD0) – 7-разрядный двунаправленный параллельный порт ввода/вывода с встроенными подтягивающими резисторами. Входы воспринимают ток до 20 мА.
- RESET – вход сброса для перезапуска МК.
- XTAL1 и XTAL2 – вход и выход инвертирующего усилителя, используемые для построения ТГ. К ним можно подключить кварцевый резонатор, задающий частоту ТГ.

7.1.4. Обзор архитектуры

Файл регистров общего назначения РОН. 32 8-разрядных. Регистровый файл занимает адреса $00…$1F, поэтому к регистрам можно обращаться и как ячейкам памяти данных.

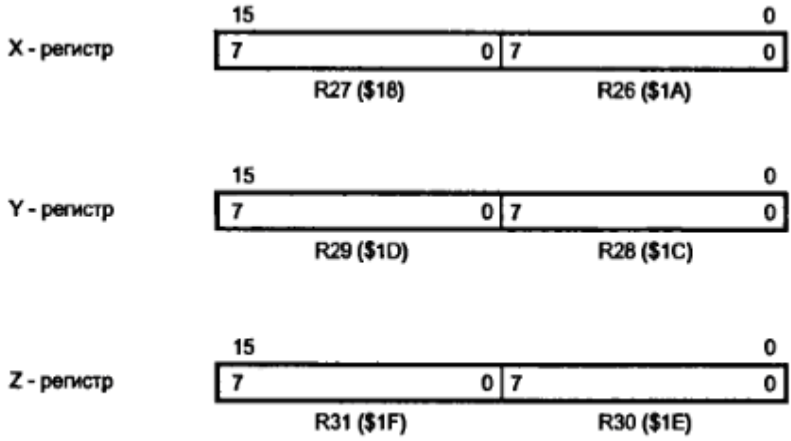
Большинство команд, использующих регистры, могут обращаться к любым РОН. Исключение составляют команды, работающих с константами: SBCI, SUBI, CPI, ANDI, ORI, LDI. Они работают только со второй половиной файла РОН – R16…R31.

Каждому регистру присвоен и адрес в первых 32 ячейках ОЗУ. Это удобно.

Пространство ввода/вывода состоит из 64 адресов $20…$5F.

АЛУ поддерживает арифметические и логические операции.

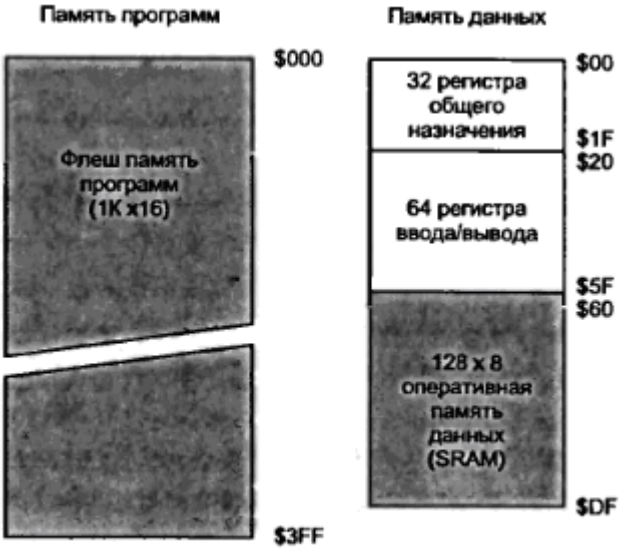
6 РОН (R26…R31) можно использовать как 3 16-разрядных указателя X,Y,Z в адресном пространстве данных. Указатель Z можно использовать для адресации таблиц в памяти программ.



Регистры X, Y, Z

7.1.5. Структура памяти

Использована Гарвардская архитектура – данные и программа в разных устройствах памяти.

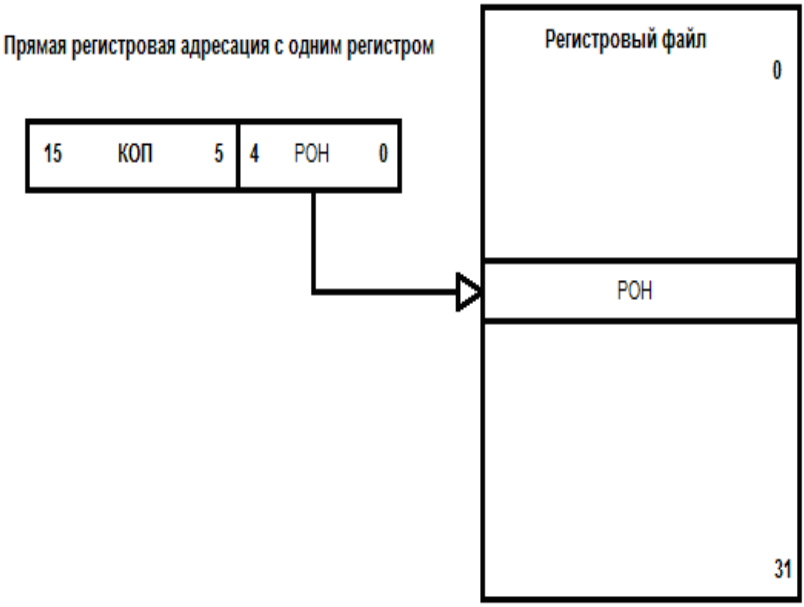


Структура памяти микроконтроллеров AVR

Во время выполнения одной команды следующая команда считывается одновременно из памяти программ.

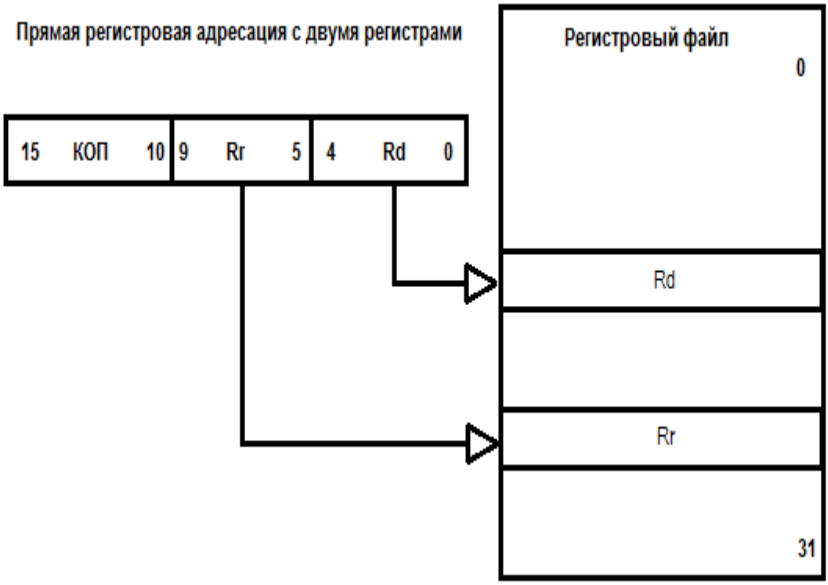
7.1.6. Режимы адресации

Прямая регистровая адресация с одним регистром. В качестве источника используется любой РОН. Результат заносится туда же.

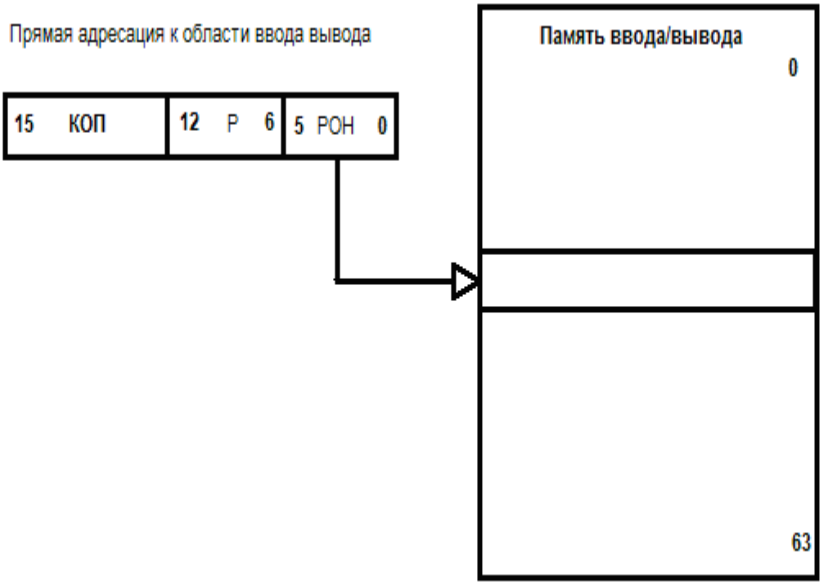


Код команды содержит 1 слово. КОП – код операции, РОН определяет используемый регистр.

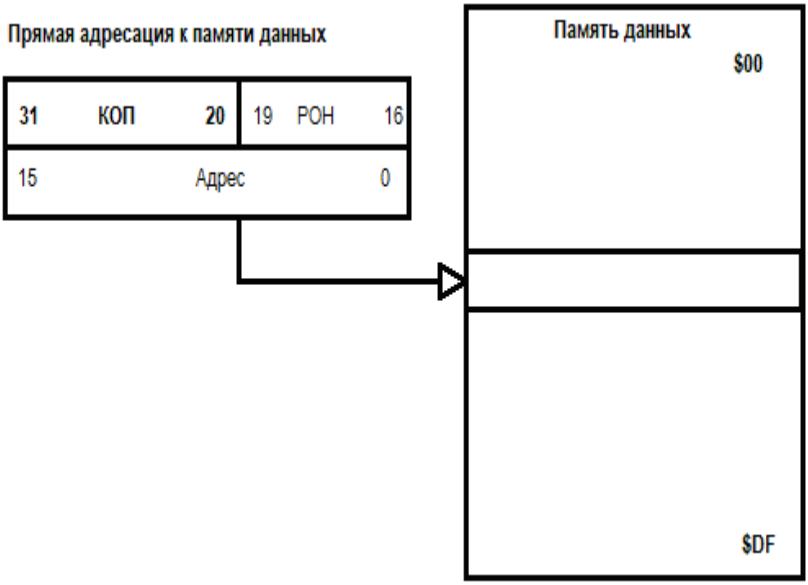
Прямая регистровая адресация с двумя регистрами. Код команды содержит 1 слово: КОП – код операции, Rr – источник данных, Rd – получатель результата.



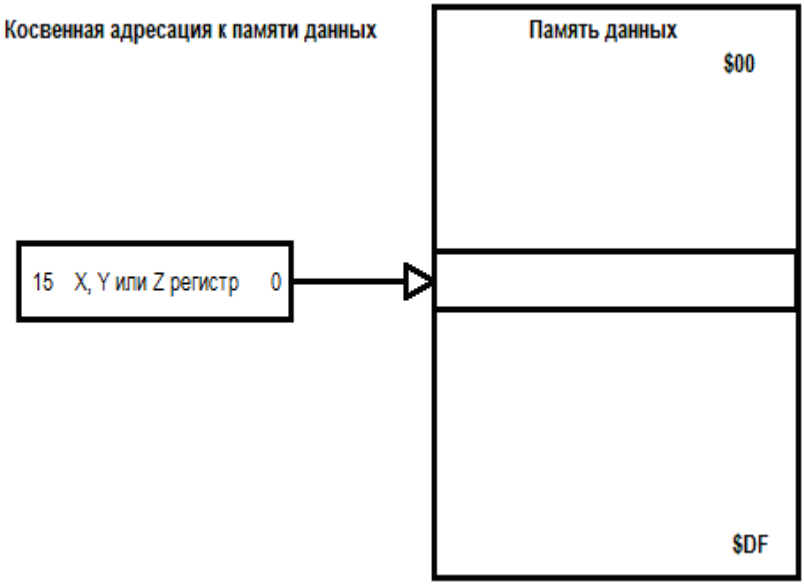
Прямая адресация к области ввода вывода. Операция осуществляется с данными в поле P. Используется регистр РОН, он может быть источником или получателем данных.



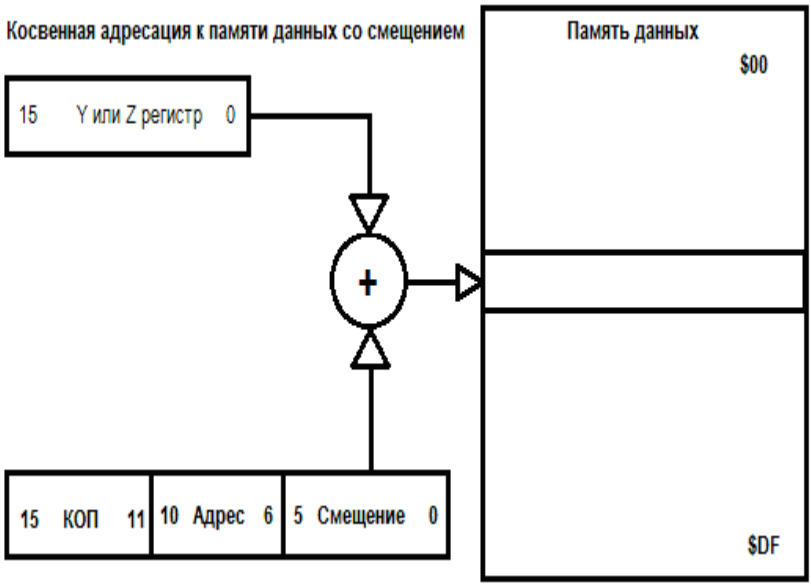
Прямая адресация к памяти данных. Код команды состоит из 2-х слов. В старшем слове команды размещены - код операции КОП и используемый РОН. В младшем слове находится 16-разрядный адрес ячейки памяти данных.



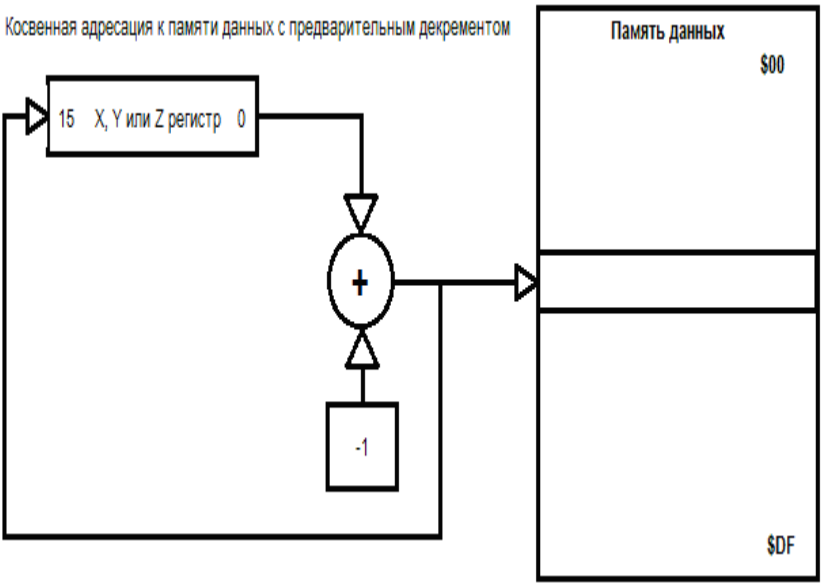
Косвенная адресация к памяти данных. Адрес операнда находится в одном из регистров X, Y, Z.



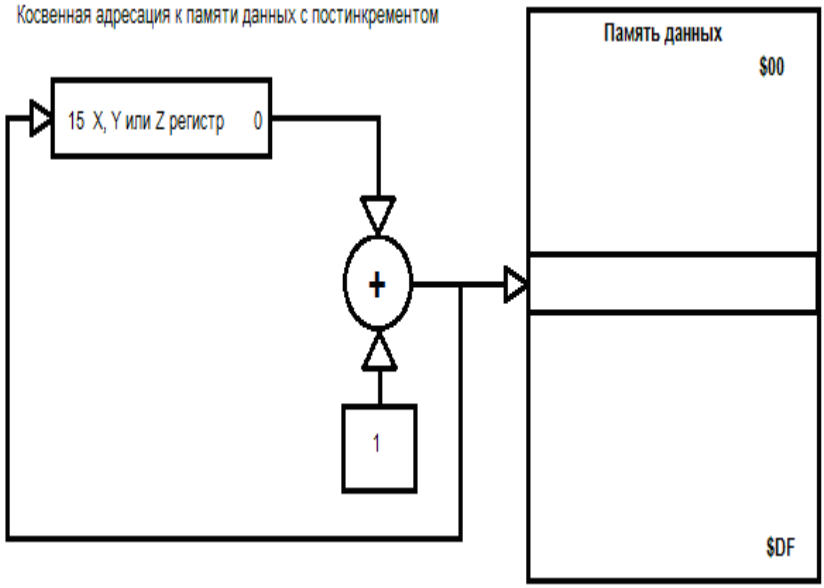
Косвенная адресация к памяти данных со смещением. Адрес операнда определяется как сумма содержимого Z или Y регистра и смещения.



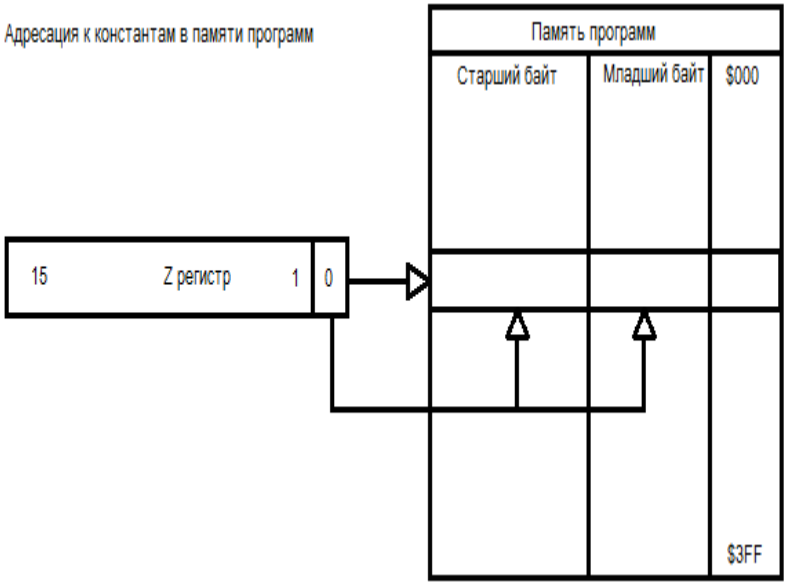
Косвенная адресация к памяти данных с предварительным декрементом. Адрес операнда находится в одном из регистров X, Y или Z. Перед выполнением операции он уменьшается на 1.



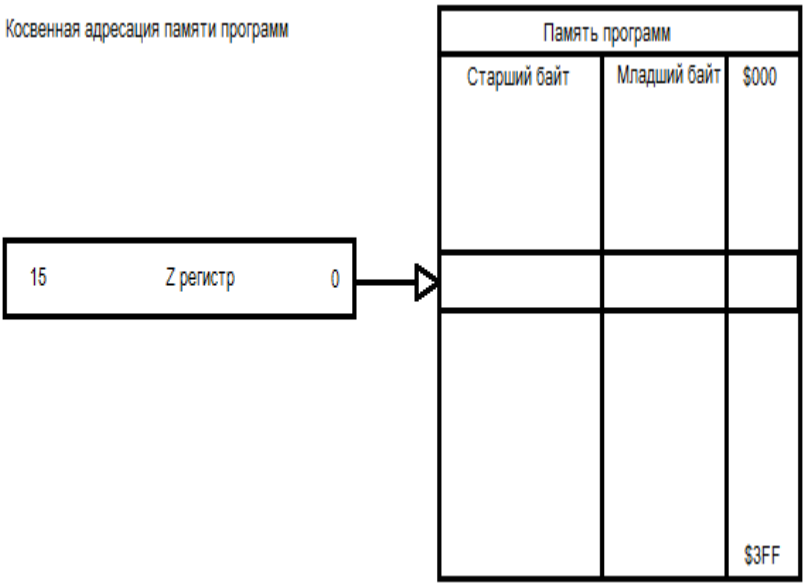
Косвенная адресация к памяти данных с постинкрементом. Адрес операнда находится в одном из регистров X, Y или Z. После выполнение операции он увеличивается на 1.



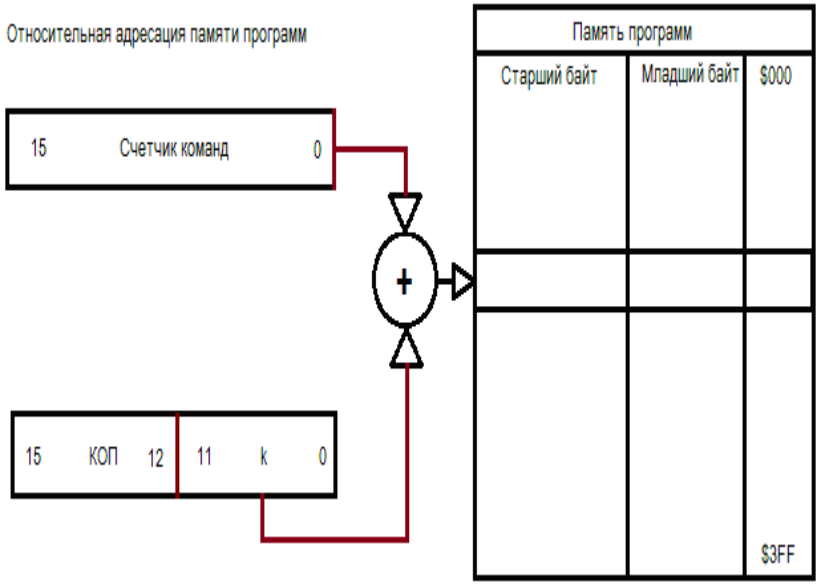
Адресация к константам в памяти программ. Константа в регистре Z. 15 старших битов определяют адрес слова, а младший (0) бит задают младший (если 0) или старший (если 1) байт константы в памяти программ.



Косвенная адресация памяти программ. После операций IJMP или ICALL выполнение программы продолжается с адреса записанного в регистре Z. Его содержимое переносится в счетчик команд.



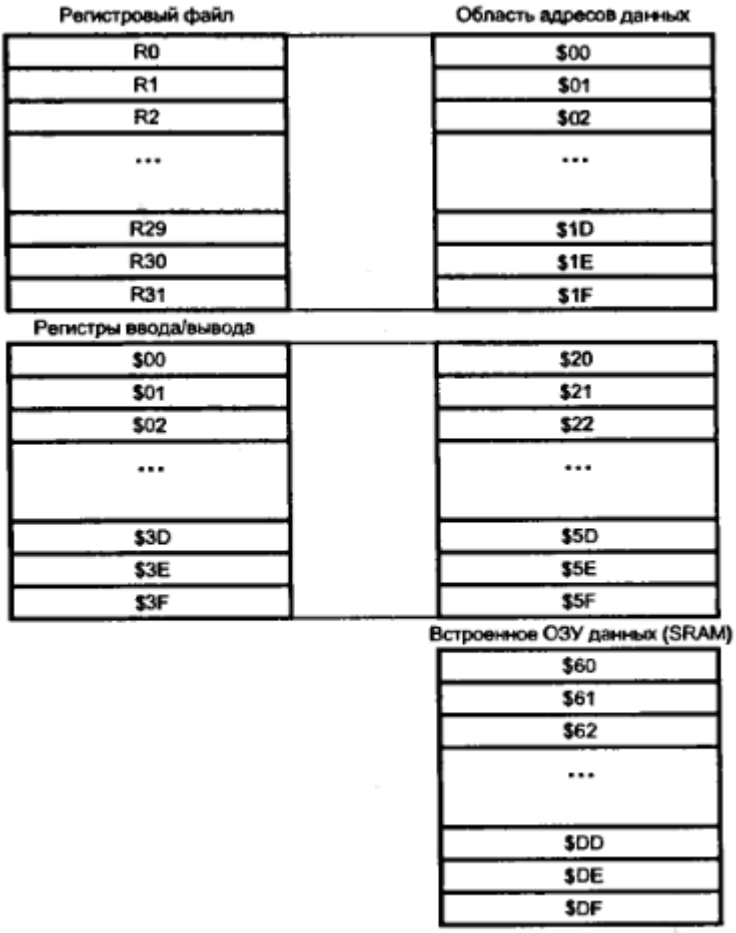
Относительная адресация памяти программ. После операций RJMP или RCALL выполнение программы продолжается с адреса (Счетчик команд)+k+1. Относительный адрес k = -2048…2047



Память программ. Содержит 2 Кб флэш-памяти. Она организована как 1Кх16. Может перепрограммироваться до 1000 раз.. Программный счетчик имеет 10 разрядов и может адресовать 1024 слов памяти программ.

EEPROM память данных. Содержит 128 байтов электрически стираемой энергонезависимой памяти (EEPROM). Организована как отдельная область данных, каждый байт которой может быть прочитан и при необходимости переписан. Выдерживает не менее 100000 циклов записи/стирания. К этой памяти может обращаться программа. Данные а нее можно занести с помощью внешнего программатора.

Оперативная память данных включает 224 ячейки: регистровый файл (32 адреса), память ввода/вывода (64 адреса), оперативная память данных (128 адресов).



Организация памяти данных в микроконтроллере AT90S2313

7.2. Ассемблер

Ассемблер преобразовывает (транслирует) исходные файлы ассемблера в объектные файлы в машинном коде. Эти файлы находятся в общем формате объектного файла (COFF). Исходные файлы могут содержать следующие инструкции ассемблера:

- Директивы Ассемблера.
- Макро директивы.
- Команды ассемблера.
- Формат инструкций исходника

Компилятор работает с исходными файлами, содержащими инструкции, метки и директивы. Инструкции и директивы, как правило, имеют один или несколько операндов.

Строка кода не должна быть длиннее 120 символов.

Строка ассемблерного кода завершается символом Enter. Строка содержит поля, разделенные пробелами. Строка может иметь одну из 4 форм:

- [метка:] директива [операнды] [Комментарий]
- [метка:] инструкция [операнды] [Комментарий]
- Комментарий
- Пустая строка

Позиции, указанные в квадратных скобках, необязательны.

Ассемблер не различает регистр символов.

Любая строка может начинаться с метки, которая является набором символов, заканчивающимся двоеточием. Метки используются для указания места, в которое передаётся управление при переходах, а также для задания имён переменных.

Операнды разделяются запятыми (без пробелов).

Текст после точки с запятой (;) и до конца строки – это комментерий. Он игнорируется компилятором. Комментарий имеет следующую форму:

```
; [Текст]
```

Компилятор **не требует**, чтобы метки, директивы, комментарии или инструкции находились в определённой колонке строки.

Примеры:

```
label:   .EQU var1=100      ; Устанавливает var1=100 (это помеченная директива).
     .EQU var2=200      ; Устанавливает var2=200  (это директива).
; Строка с одним только комментарием
```

7.2.1. Команды ассемблера

Операнды в командах могут быть таких видов:

- Rd: Результирующий (и исходный) регистр в регистровом файле
- Rr: Исходный регистр в регистровом файле
- b: Константа (3 бита), может быть константное выражение
- s: Константа (3 бита), может быть константное выражение
- P: Константа (5-6 бит), может быть константное выражение
- K6; Константа (6 бит), может быть константное выражение
- K8: Константа (8 бит), может быть константное выражение
- k: Константа (размер зависит от инструкции), может быть константное выражение
- q: Константа (6 бит), может быть константное выражение
- Rdl:= R24, R26, R28, R30. Для инструкций ADIW и SBIW
- X,Y,Z: Регистры косвенной адресации (X=R27:R26, Y=R29:R28, Z=R31:R30)

Используемые обозначения:

Символика	Назначение	Комментарий
SREG	Регистр состояния	Адрес $3F
C	Флаг переноса	Разряд 0 в регистре SREG
Z	Флаг нулевого результата	Разряд 1 в регистре SREG
N	Флаг отрицательного результата	Разряд 2 в регистре SREG
V	Флаг переполнения	Разряд 3 в регистре SREG
S	Флаг знака	Разряд 4 в регистре SREG
Il	Флаг половинного переноса	Разряд 5 в регистре SREG
T	Флаг копирования	Разряд 6 в регистре SREG
I	Флаг разрешения прерываний	Разряд 7 в регистре SREG
Rd	Регистр назначения	
Rr	Регистр передающий	
R	Регистр результата	
Kn	n-битная константа	
X, Y, Z	Указатели при косвенной адресации	X = R27:R26 Y = R29:R28 Z = R31:R30
Rdl	Младший байт регистровой пары	Пара Rdh:Rdl
Rdh	Старший байт регистровой пары	Пара Rdh:Rdl
P	Адрес порта ввода/вывода	

q	Смещение при косвенной адресации	
Стек	Область памяти для хранения адреса возврата или промежуточного значения	
SP	Указатель стека	
X	Обозначает разряды, устанавливаемые ассемблером в 0.	

Команды условного перехода с учетом знака.

Условие	Команда	Пояснение
Rd ≥ Rr ?	brge	Переход, если Rd ≥Rr (S=0)
Rd = Rr ?	breq	Переход, если Rd = Rr (Z=1)
Rd < Rr ?	brlt	Переход, если Rd < Rr (S=1)

Команды прямого опроса флагов.

Условие	Команда	Пояснение
C ?	brcs	Переход по переносу (C = 1)
C ?	brcc	Переход, если переноса нет (C = 0)
Z ?	breq	Переход по нулю (Z = 1)
Z ?	brne	Переход по не нулю (Z = 0)
N ?	brmi	Переход по минусу (N = 1)
N ?	brpl	Переход по плюсу(N = 0)
V ?	brvs	Переход по переполнению (V = 1)
V ?	brvc	Переход, если переполнения нет (V = 0)

Арифметические и логические команды.

Мнемоника	Описание	Действие	Циклы	Примеч.
ADD Rd,Rr	Сложить	Rd < Rd+Rr	1	
ADDI Rd,K	Сложить с константой	Rd < Rd+K	1	
ADC Rd,Rr	Сложить с переносом	Rd < Rd+Rr+C	1	
ADCI Rd,Rr	Сложить константу с переносом	Rd < Rd+K+C	1	
ADW Rdl,Rr	Сложить слово с константой	Rdh,l < Rdh,l+K	2	Rdl - парный
SUB Rd,Rr	Вычесть	Rd < Rd-Rr	1	

SUBI Rd,K	Вычесть константу	Rd <- Rd-K	1	
SBC Rd,Rr	Вычесть с переносом	Rd < Rd-Rr-C	1	
SBCI Rd,Rr	Вычесть константу с переносом	Rd < Rd-K-C	1	
SBW Rdl,Rr	Вычесть слово с константой	Rdh,l <- Rdh,l-K	2	Rdl - парный
AND Rd,Rr	Логическое И	Rd < Rd AND Rr	1	
ANDI Rd,K	Логическое И с константой	Rd < Rd AND K	1	
OR Rd,Rr	Логическое ИЛИ	Rd < Rd OR Rr	1	
ORI Rd,K	Логическое ИЛИ с константой	Rd < Rd OR K	1	
EOR Rd,Rr	Исключающее ИЛИ	Rd < Rd XOR Rr	1	
COM Rd	Дополнение до 1	Rd < $FF-Rd	1	
NEG Rd	Дополнение до 2	Rd < $00-Rd	1	
SBR Rd,K	Установить биты	Rd < Rd OR K	1	
CBR Rd,K	Сбросить биты	Rd < Rd AND K	1	
INC Rd	Увеличить на 1	Rd < Rd+1	1	
DEC Rd	Уменьшить на 1	Rd < Rd-1	1	
TST Rd	Проверить на 0 или 1	Rd < Rd AND Rd	1	
CLR Rd	Очистить	Rd < Rd XOR Rd	1	Все 0
ER Rd	Установить	Rd < $FF	1	Все 1

Команды ветвления.

Мнемоника	Описание	Действие	Флаги	Циклы
RJMP k	Относительный переход	PC < PC+k+1	Нет	2
LJMP k	Переход по адресу (Z)	PC < Z	Нет	2
RCALL k	Относительный вызов подпрограммы	PC < PC+k+1	Нет	3
ICALL	Вызов подпрограммы по адресу (Z)	PC < Z	Нет	3
RET	Выход из подпрограммы	PC < STACK	Нет	4
RETI	Выход из прерывания	PC < STACK	I	4
CPSE Rd,Rr	Сравнить, пропуск при равно	If(Rd=Rr) PC<PC+2 или 3	Z,N,V,C,H	1/2
CP Rd,Rr	Сравнить	Rd-Rr	Z,N,V,C,H	1

CPC Rd,Rr	Сравнить с прерыванием	Rd-Rr-C	Z,N,V,C,H	1
CPI Rd,K	Сравнить с константой	Rd-C	Z,N,V,C,H	1
SBRC Rr,b	Пропуск, если в Rr бит b сброшен	If(Rr(b)=0) PC<PC+2 или 3	Нет	1/2
SBRS Rr,b	Пропуск, если в Rr бит b установлен	If(Rr(b)=1) PC<PC+2 или 3	Нет	1/2
SBIC P,b	Пропуск, если в регистре ввода/вывода P бит b сброшен	If(P(b)=0) PC<PC+2 или 3	Нет	1/2
SBIS P,b	Пропуск, если в регистре ввода/вывода P бит b установлен	If(P(b)=1) PC<PC+2 или 3	Нет	1/2
BRBS s,k	Переход, если в SREG установлен флаг s	If(SREG(s)=1) PC<PC+1+k	Нет	1/2
BRBS s,k	Переход, если в SREG сброшен флаг s	If(SREG(s)=0) PC<PC+1+k	Нет	1/2
BREQ k	Переход, если равно	If(Z=1) PC<PC+1+k	Нет	1/2
BRNE k	Переход, если не равно	If(Z=0) PC<PC+1+k	Нет	1/2
BRCS k	Переход, если перенос установлен	If(C=1) PC<PC+1+k	Нет	1/2
BRCC k	Переход, если перенос сброшен	If(C=0) PC<PC+1+k	Нет	1/2
BRSH k	Переход, если равно или больше	If(C=0) PC<PC+1+k	Нет	1/2
BRLO k	Переход, если меньше	If(C=1) PC<PC+1+k	Нет	1/2
BRMI k	Переход, если минус	If(N=1) PC<PC+1+k	Нет	1/2
BRPL k	Переход, если плюс	If(N=0) PC<PC+1+k	Нет	1/2
BRGE k	Переход, если больше или равно	If(N XOR V=0) PC<PC+1+k	Нет	1/2
BRLT k	Переход, если меньше 0	If(N XOR V=1) PC<PC+1+k	Нет	1/2
BRHS k	Переход, если флаг H	If(H=1)	Нет	1/2

	установлен	PC<PC+1+k		
BRHC k	Переход, если флаг H сброшен	If(H=0) PC<PC+1+k	Нет	1/2
BRTS k	Переход, если флаг T установлен	If(H=1) PC<PC+1+k	Нет	1/2
BRTC k	Переход, если флаг T сброшен	If(H=0) PC<PC+1+k	Нет	1/2
BRVS k	Переход, если флаг V установлен	If(H=1) PC<PC+1+k	Нет	1/2
BRVC k	Переход, если флаг V сброшен	If(H=0) PC<PC+1+k	Нет	1/2
BRIE k	Переход, если прерывания азрешены	If(I=1) PC<PC+1+k	Нет	1/2
BRID k	Переход, если прерывания запрещены	If(I=0) PC<PC+1+k	Нет	1/2

Команды пересылок.

Мнемоника	Описание	Действие	Флаги	Циклы
MOV Rd,Rr	Пересылка между регистрами	Rd < Rr	Нет	1
LDI Rd,K	Загрузить константу	Rd < K	Нет	1
LD Rd,X	Загрузить регистр непосредственно	Rd < (X)	Нет	2
LD Rd,X+	Загрузить регистр непосредственно с постинкрементом	Rd < (X) X < X+1	Нет	2
LD Rd,-X	Загрузить регистр непосредственно с предварительным декрементом	X < X-1 Rd < (X)	Нет	2
LD Rd,Y	Загрузить регистр непосредственно	Rd < (Y)	Нет	2
LD Rd,Y+	Загрузить регистр непосредственно с постинкрементом	Rd < (Y) Y < Y+1	Нет	2
LD Rd,-Y	Загрузить регистр непосредственно с предварительным декрементом	Y < Y-1 Rd < (Y)	Нет	2
LDD Rd,Y+q	Загрузить регистр непосредственно по адресу в Y со смещением q	Rd < (Y+k)	Нет	2
LD Rd,Z	Загрузить регистр	Rd < (Z)	Нет	2

	непосредственно			
LD Rd,Z+	Загрузить регистр непосредственно с постинкрементом	Rd < (Z) Z < Z+1	Нет	2
LD Rd,-Z	Загрузить регистр непосредственно с предварительным декрементом	Z < Z-1 Rd < (Z)	Нет	2
LDD Rd,Z+q	Загрузить регистр непосредственно со смещением	Rd < (Z+k)	Нет	2
LDS Rd,k	Загрузить регистр из ОЗУ по адресу (k)	Rd < (k)	Нет	3
ST X,Rr	Сохранить регистр непосредственно по адресу в X	(X) < Rr	Нет	2
ST X+,Rr	Сохранить регистр непосредственно по адресу в X с постинкрементом	(X) < Rr X < X+1	Нет	2
ST -X,Rr	Сохранить регистр непосредственно по адресу в X с предварительным декрементом	X < X-1 (X) < Rr	Нет	2
STD X+q,Rr	Сохранить регистр непосредственно по адресу в X со смещением q	(X+q) < Rr	Нет	2
ST Y,Rr	Сохранить регистр непосредственно по адресу в Y	(Y) < Rr	Нет	2
ST Y+,Rr	Сохранить регистр непосредственно по адресу в Y с постинкрементом	(Y) < Rr Y < Y+1	Нет	2
ST -Y,Rr	Сохранить регистр непосредственно по адресу в Y с предварительным декрементом	Y < Y-1 (Y) < Rr	Нет	2
STD Y+q,Rr	Сохранить регистр непосредственно по адресу в Y со смещением q	(Y+q) < Rr	Нет	2
ST Z,Rr	Сохранить регистр непосредственно по адресу в Z	(Z) < Rr	Нет	2
ST Z+,Rr	Сохранить регистр непосредственно по адресу в Z с постинкрементом	(Z) < Rr Z < Z+1	Нет	2
ST -Z,Rr	Сохранить регистр непосредст-	Z < Z-1	Нет	2

	венно по адресу в Z с предварительным декрементом	(Y) < Rr		
STS k,Rr	Сохранить регистр в ОЗУ по адресу (k)	(k) < Rr	Нет	3
LPM	Загрузка из памяти программ по адресу в Z	R0 < (Z)	Нет	3
IN Rd,P	Ввод в регистр из порта P	Rd < P	Нет	1
OUT P,Rr	Вывод из регистра в порт P	P < Rr	Нет	1
PUSH Rr	Сохранить регистр в стеке	STACK < Rr	Нет	2
POP Rd	Загрузить регистр из стека	Rd < STACK	Нет	2

Команды работы с битами.

Мнемоника	Описание	Действие	Флаги	Циклы
SBI P,b	Установить бит b в регистре ввода/вывода P	I/O (P,b) < 1	Нет	2
CBI P,b	Сбросить бит b в регистре ввода/вывода P	I/O (P,b) < 0	Нет	2
LSL Rd	Логический сдвиг влево на 1 бит в регистре	Rd(n+1) < Rd(n) Rd(0) < 0	Z,C,N,V	1
LSR Rd	Логический сдвиг вправо на 1 бит в регистре	Rd(n) < Rd(n+1) Rd(7) < 0	Z,C,N,V	1
ROL Rd	Сдвиг влево через C на 1 бит в регистре	Rd(0) < C Rd(n+1) < Rd(n) C < Rd(7)	Z,C,N,V	1
ROR Rd	Сдвиг вправо через C на 1 бит в регистре	Rd(7) < C Rd(n) < Rd(n+1) C < Rd(0)	Z,C,N,V	1
LSR Rd	Логический сдвиг вправо на 1 бит в регистре	Rd(n) < Rd(n+1) Rd(7) < 0	Z,C,N,V	1
SWAP Rd	Обмен ниблов (полубайт) в регистре	Rd(3…0) < Rd(7…4)	Нет	1

		Rd(7…4) < Rd(3…0)		
BSET s	Установить флаг s	SREG(s) < 1	SREG(s)	1
BCLEAR s	Сбросить флаг s	SREG(s) < 0	SREG(s)	1
BST Rr,b	Запомнить бит b регистра в флаге T	T < Rr(b)	T	1
BLD Rd,b	Прочитать бит из T в бит b регистра	Rd(b)< T	Нет	1
SEC	Установить перенос	C < 1	C	1
CLC	Сбросить перенос	C < 0	C	1
SEN	Установить флаг N	N < 1	N	1
CLN	Сбросить флаг N	N < 0	N	1
SEZ	Установить флаг Z	Z < 1	Z	1
CLZ	Сбросить флаг Z	Z < 0	Z	1
SES	Установить флаг S	S < 1	S	1
CLS	Сбросить флаг S	S < 0	S	1
SEV	Установить флаг V	V < 1	V	1
CLV	Сбросить флаг V	V < 0	V	1
SET	Установить флаг T	T < 1	T	1
CLT	Сбросить флаг T	T < 0	T	1
SEH	Установить флаг H	H < 1	H	1
CLH	Сбросить флаг H	H < 0	H	1
NOP	Нет операции		Нет	1
SLEEP	Останов		Нет	1
WDR	Сброс сторожевого таймера		Нет	1

7.2.2. Директивы ассемблера

Компилятор поддерживает ряд директив. Директивы не транслируются непосредственно в код. Вместо этого они используются для указания положения в программной памяти, определения макросов, инициализации памяти и т.д. Все директивы предваряются точкой.

Список директив приведён в следующей таблице.

Директива	Описание
BYTE	Зарезервировать байты в ОЗУ

CSEG	Программный сегмент
DB	Определить байты во флэш или EEPROM
DEF	Назначить регистру символическое имя
DEVICE	Определить устройство для которого компилируется программа
DSEG	Сегмент данных
DW	Определить слова во флэш или EEPROM
ENDM, ENDMACRO	Конец макроса
EQU	Установить постоянное выражение
ESEG	Сегмент EEPROM
EXIT	Выйти из файла
INCLUDE	Вложить другой файл
LIST	Включить генерацию листинга
LISTMAC	Включить разворачивание макросов в листинге
MACRO	Начало макроса
NOLIST	Выключить генерацию листинга
ORG	Установить положение в сегменте
SET	Установить переменный символический эквивалент выражения

Байты. BYTE - зарезервировать байты в ОЗУ. Директива BYTE резервирует байты в ОЗУ. Если вы хотите иметь возможность ссылаться на выделенную область памяти, то директива BYTE должна быть предварена меткой. Директива принимает один обязательный параметр, который указывает количество выделяемых байт. Эта директива может использоваться только в сегменте данных(смотреть директивы CSEG и DSEG). Выделенные байты не инициализируются. Синтаксис:

МЕТКА: .BYTE выражение

Пример:

```
.DSEG
var1:   .BYTE 1          ; резервирует 1 байт для var1
table:  .BYTE tab_size   ; резервирует tab_size байт
    .CSEG
    Ldi r30,low(var1)    ; Загружает младший байт регистра Z
    Ldi r31,high(var1)   ; Загружает старший байт регистра Z
    Ld  r1,Z             ; Загружает var1 в регистр 1
```

DB - определить байты во флэш или EEPROM. Директива DB резервирует необходимое количество байт в памяти программ или в EEPROM. Если вы хотите иметь возможность ссылаться на выделенную область памяти, то директива DB должна быть предварена меткой. Директива DB должна иметь хотя бы один параметр. Данная директива может быть размещена только в сегменте программ (CSEG) или в сегменте EEPROM (ESEG).

Параметры, передаваемые директиве - это последовательность выражений разделённых запятыми. Каждое выражение должно быть или числом в диапазоне (-128..255), или в результате вычисления должно давать результат в этом же диапазоне, в противном случае число усекается до байта, причём БЕЗ выдачи предупреждений.

Если директива получает более одного параметра и текущим является программный сегмент, то параметры упаковываются в слова (первый параметр - младший байт). Если число параметров нечётно, то последнее выражение будет усечено до байта и записано как слово со старшим байтом равным нулю, даже если далее идет ещё одна директива DB. Синтаксис:

МЕТКА: .DB список_выражений

Пример:

```
.CSEG
consts: .DB 0, 255, 0b01010101, -128, 0xaa
.ESEG
const2: .DB 1,2,3
```

DW - определить слова во флэш или EEPROM. Директива DW резервирует необходимое количество слов в памяти программ или в EEPROM. Если вы хотите иметь возможность ссылаться на выделенную область памяти, то директива DW должна быть предварена меткой. Директива DW должна иметь хотя бы один параметр. Данная директива может быть размещена только в сегменте программ (CSEG) или в сегменте EEPROM (ESEG).

Параметры, передаваемые директиве, - это последовательность выражений разделённых запятыми. Каждое выражение должно быть или числом в диапазоне (-32768..65535), или в результате вычисления должно давать результат в этом же диапазоне, в противном случае число усекается до слова, причем БЕЗ выдачи предупреждений. Синтаксис:

МЕТКА: .DW expressionlist

Пример:

```
.CSEG
varlist:= .DW 0, 0xffff, 0b1001110001010101, -32768, 65535
.ESEG
eevarlst: .DW 0,0xffff,10
```

Сегменты. DSEG - сегмент данных. Директива DSEG определяет начало сегмента данных. Исходный файл может состоять из нескольких сегментов данных, которые объединяются в один сегмент при компиляции. Сегмент данных обычно состоит только из директив BYTE и меток. Сегменты данных имеют свои собственные побайтные счётчики положения. Директива ORG может быть использована для размещения переменных в необходимом месте ОЗУ. Директива не имеет параметров. Синтаксис:

.DSEG

Пример:

```
.DSEG                   ; Начало сегмента данных
var1:   .BYTE 1             ; зарезервировать 1 байт для var1
table:  .BYTE tab_size; зарезервировать tab_size байт.
.CSEG
    ldi r30,low(var1)       ; Загрузить младший байт регистра Z
    ldi r31,high(var1)      ; Загрузить старший байт регистра Z
    ld r1,Z             ; Загрузить var1 в регистр r1
```

CSEG - программный сегмент. Директива CSEG определяет начало программного сегмента. Исходный файл может состоять из нескольких программных сегментов, которые объединяются в один программный сегмент при компиляции. Программный сегмент является сегментом по умолчанию. Программные сегменты имеют свои собственные счётчики положения, которые считают не побайтно, а по словно. Директива ORG может быть использована для размеще-

ния кода и констант в необходимом месте сегмента. Директива CSEG не имеет параметров. Синтаксис:

.CSEG

Пример:

```
.DSEG           ; Начало сегмента данных
vartab: .BYTE 4     ; Резервирует 4 байта в ОЗУ
.CSEG           ; Начало кодового сегмента
const: .DW 2        ; Разместить константу 0x0002 в памяти программ
    mov r1,r0   ; Выполнить действия
```

ESEG - сегмент EEPROM. Директива ESEG определяет начало сегмента EEPROM. Исходный файл может состоять из нескольких сегментов EEPROM, которые объединяются в один сегмент при компиляции. Сегмент EEPROM обычно состоит только из директив DB, DW и меток. Сегменты EEPROM имеют свои собственные побайтные счётчики положения. Директива ORG может быть использована для размещения переменных в необходимом месте EEPROM. Директива не имеет параметров. Синтаксис:

.ESEG

Пример:

```
.DSEG               ; Начало сегмента данных
var1:   .BYTE 1         ; зарезервировать 1 байт для var1
table:  .BYTE tab_size      ; зарезервировать tab_size байт.
.ESEG
eevar1: .DW 0xffff      ; проинициализировать 1 слово в EEPROM
```

ORG - Установить положение в сегменте. Директива ORG устанавливает счётчик положения равным заданной величине, которая передаётся как параметр. Для сегмента данных она устанавливает счётчик положения в SRAM (ОЗУ), для сегмента программ это программный счётчик, а для сегмента EEPROM это положение в EEPROM. Если директиве предшествует метка (в той же строке) то метка размещается по адресу указанному в параметре директивы. Перед началом компиляции программный счётчик и счётчик EEPROM равны нулю, а счётчик ОЗУ равен 32 (поскольку адреса 0-31 заняты регистрами). Обратите внимание что для ОЗУ и EEPROM используются побайтные счётчики а для программного сегмента - пословный. Синтаксис:

.ORG выражение

Пример:

```
.DSEG           ; Начало сегмента данных
.ORG 0x37       ; Установить адрес SRAM равным 0x37
variable: .BYTE 1   ; Зарезервировать байт по адресу 0x37H
.CSEG
.ORG 0x10       ; Установить программный счётчик равным 0x10
    mov r0,r1   ; Данная команда будет размещена по адресу 0x10
```

DEF - назначить регистру символическое имя. Директива DEF позволяет ссылаться на регистр через некоторое символическое имя. Назначенное имя может использоваться во всей нижеследующей части программы для обращений к данному регистру. Регистр может иметь несколько различных имен. Символическое имя может быть переназначено позднее в программе. Синтаксис:

```
.DEF Символическое_имя = Регистр
```

Пример:

```
.DEF temp=R16
.DEF ior=R0
.CSEG
    ldi temp,0xf0     ; Загрузить 0xf0 в регистр temp (R16)
    in ior,0x3f    ; Прочитать SREG в регистр ior (R0)
    eor temp,ior      ; Регистры temp и ior складываются по исключающему или
```

- Макросы

MACRO - начало макроса. С директивы MACRO начинается определение макроса. В качестве параметра директиве передаётся имя макроса. При встрече имени макроса позднее в тексте программы компилятор заменяет это имя на тело макроса. Макрос может иметь до 10 параметров, к которым в его теле обращаются через @0-@9. При вызове параметры перечисляются через запятые. Определение макроса заканчивается директивой ENDMACRO. По умолчанию в листинг включается только вызов макроса, для разворачивания макроса необходимо использовать директиву LISTMAC. Макрос в листинге показывается знаком +. Синтаксис:

```
.MACRO макроимя
```

Пример:

```
.MACRO SUBI16       ; Начало макроопределения
    subi @1,low(@0)   ; Вычесть младший байт параметра 0 из параметра 1
```

```
    sbci @2,high(@0)  ; Вычесть старший байт параметра 0 из параметра 2
.ENDMACRO             ; Конец макроопределения
.CSEG                 ; Начало программного сегмента
SUBI16 0x1234,r16,r17     ; Вычесть 0x1234 из пары r17:r16
```

ENDMACRO - конец макроса. Директива определяет конец макроопределения, и не принимает никаких параметров. Для информации по определению макросов смотрите директиву MACRO. Синтаксис:

```
.ENDMACRO
```

Пример:

```
.MACRO SUBI16         ; Начало определения макроса
    subi r16,low(@0)  ; Вычесть младший байт первого параметра
    sbci r17,high(@0) ; Вычесть старший байт первого параметра
.ENDMACRO
```

LISTMAC - включить разворачивание макросов в листинге. После директивы LISTMAC компилятор будет показывать в листинге содержимое макроса. По умолчанию в листинге показывается только вызов макроса и передаваемые параметры. Синтаксис:

```
.LISTMAC
```

Пример:

```
.MACRO MACX     ; Определение макроса
    add r0,@0 ; Тело макроса
    eor r1,@1
.ENDMACRO       ; Конец макроопределения
.LISTMAC      ; Включить разворачивание макросов
MACX r2,r1          ; Вызов макроса (в листинге будет показано тело макроса)
```

Выражения. EQU - установить постоянное выражение. Директива EQU присваивает метке значение. Эта метка может позднее использоваться в выражениях. Метка которой присвоено значение данной директивой не может быть переназначена и её значение не может быть изменено. Синтаксис:

```
.EQU метка = выражение
```

Пример:

```
.EQU io_offset = 0x23
.EQU porta = io_offset + 2
```

```
.CSEG           ; Начало сегмента данных
    clr r2      ; Очистить регистр r2
    out porta,r2    ; Записать в порт A
```

SET - Установить переменный символический эквивалент выражения. Директива SET присваивает имени некоторое значение. Это имя позднее может быть использовано в выражениях. Причем в отличие от директивы EQU значение имени может быть изменено другой директивой SET. Синтаксис:

.SET имя = выражение

Пример:

```
.SET io_offset = 0x23
.SET porta  = io_offset + 2
.CSE            ; Начало кодового сегмента
    clr r2      ; Очистить регистр 2
    out porta,r2; Записать в порт A
```

DEF - назначить регистру символическое имя. Директива DEF позволяет ссылаться на регистр через некоторое символическое имя. Назначенное имя может использоваться во всей нижеследующей части программы для обращений к данному регистру. Регистр может иметь несколько различных имен. Символическое имя может быть переназначено позднее в программе. Синтаксис:

.DEF Символическое_имя = Регистр

Пример:

```
.DEF temp=R16
.DEF ior=R0
.CSEG
    ldi temp,0xf0   ; Загрузить 0xf0 в регистр temp (R16)
    in ior,0x3f     ; Прочитать SREG в регистр ior (R0)
    eor temp,ior    ; Регистры temp и ior складываются по исключающему или
```

DEVICE - определить устройство. Директива DEVICE позволяет указать, для какого устройства компилируется программа. При использовании данной директивы компилятор выдаст предупреждение, если будет найдена инструкция, которую не поддерживает данный микроконтроллер. Также будет выдано предупреждение, если программный сегмент, либо сегмент EEPROM превысят размер, допускаемый устройством. Если же директива не используется, то все ин-

струкции считаются допустимыми, и отсутствуют ограничения на размер сегментов. Синтаксис:

.DEVICE AT90S1200 | AT90S2313 | AT90S2323 | AT90S2333 | AT90S2343 | AT90S4414 | AT90S4433 | AT90S4434 | AT90S8515 | AT90S8534 | AT90S8535 | ATtiny11 | ATtiny12 | ATtiny22 | ATmega603 | ATmega103

Пример:

```
.DEVICE AT90S1200   ; Используется AT90S1200
.CSEG
    push r30    ; инструкция вызовет предупреждение, AT90S1200 её не имеет
```

- Файлы

EXIT - выйти из файла. Встретив директиву EXIT, компилятор прекращает компиляцию данного файла. Если директива использована во вложенном файле (см. директиву INCLUDE), то компиляция продолжается со строки следующей после директивы INCLUDE. Если же файл не является вложенным, то компиляция прекращается. Синтаксис:

.EXIT

Пример:

```
.EXIT     ; Выйти из данного файла
```

INCLUDE - вложить другой файл. Встретив директиву INCLUDE компилятор открывает указанный в ней файл, компилирует его пока файл не закончится или не встретится директива EXIT, после этого продолжает компиляцию начального файла со строки следующей за директивой INCLUDE. Вложенный файл может также содержать директивы INCLUDE. Синтаксис:

.INCLUDE "имя_файла"

Пример:

```
; файл iodefs.asm:

 .EQU sreg = 0x3f        ; Регистр статуса
 .EQU sphigh = 0x3e      ; Старший байт указателя стека
 .EQU splow    = 0x3d    ; Младший байт указателя стека
; файл incdemo.asm
.INCLUDE iodefs.asm      ; Вложить определения портов
in r0,sreg               ; Прочитать регистр статуса
```

Листинги. LIST - включить генерацию листинга. Директива LIST указывает компилятору на необходимость создания листинга. Листинг представляет из себя комбинацию ассемблерного кода, адресов и кодов операций. По умолчанию генерация листинга включена, однако данная директива используется совместно с директивой NOLIST для получения листингов отдельных частей исходных файлов. Синтаксис:

.LIST

Пример:
```
.NOLIST              ; Отключить генерацию листинга
.INCLUDE "macro.inc"      ; Вложенные файлы не будут
.INCLUDE "const.def"      ; отображены в листинге
.LIST                ; Включить генерацию листинга
```

NOLIST - выключить генерацию листинга. Директива NOLIST указывает компилятору на необходимость прекращения генерации листинга. Листинг представляет из себя комбинацию ассемблерного кода, адресов и кодов операций. По умолчанию генерация листинга включена, однако может быть отключена данной директивой. Кроме того данная директива может быть использована совместно с директивой LIST для получения листингов отдельных частей исходных файлов. Синтаксис**:**

.NOLIST

Пример:
```
.NOLIST         ; Отключить генерацию листинга
.INCLUDE "macro.inc" ; Вложенные файлы не будут
.INCLUDE "const.def"  ; отображены в листинге
.LIST           ; Включить генерацию листинга
```

7.2.3. Выражения

Компилятор позволяет использовать в программе выражения, которые могут состоять операндов, знаков операций и функций. Все выражения являются 32-битными.

Операнды. В выражениях могут быть использованы следующие операнды:

- Метки, определённые пользователем (дают значение своего положения).
- Переменные, определённые директивой SET.
- Константы, определённые директивой EQU.

- Числа заданные в формате:
  Десятичном (принят по умолчанию): 10, 255.
  Шестнадцатеричном (два варианта записи): 0x0a, $0a, 0xff, $ff.
  Двоичном: 0b00001010, 0b11111111.
  Восьмеричном (начинаются с нуля): 010, 077.
- PC - текущее значение программного счётчика (Program Counter).
- Операторы

Компилятор поддерживает ряд операторов, которые перечислены в таблице (чем выше положение в таблице, тем выше приоритет оператора). Выражения могут заключаться в круглые скобки, такие выражения вычисляются перед выражениями за скобками.

Приор.	Символ	Описание	Пример
14	!	Логическое отрицание. Возвращает 1, если выражение равно 0, и наоборот.	ldi r16, !0xf0 ; В r16 загрузить 0x00
14	~	Побитное отрицание. Возвращает результат, в котором все биты проинвертированы.	ldi r16, ~0xf0 ;В r16 загрузить 0x0f
14	-	Минус. Возвращает арифметическое отрицание выражения.	ldi r16,-2 ;Загрузить –2 (0xfe) в r16
13	*	Умножение. Возвращает результат умножения двух выражений.	ldi r30, label*2
13	/	Деление. Возвращает целую часть результата деления левого выражения на правое.	ldi r30, label/2
12	+	Суммирование. Возвращает сумму двух выражений.	ldi r30, c1+c2
12	-	Вычитание. Возвращает результат вычитания правого выражения из левого.	ldi r17, c1-c2
11	<<	Сдвиг влево. Возвращает левое выражение сдвинутое влево на число бит, указанное справа.	ldi r17, 1<<bitmask ; В r17 загрузить 1, сдвинутую влево на bitmask

11	>>	Сдвиг вправо. Возвращает левое выражение сдвинутое вправо на число бит указанное справа.	ldi r17, c1>>c2 ; В r17 загрузить c1, сдвинутое вправо c2 раз
10	<	Меньше чем. Возвращает 1 если левое выражение меньше чем правое (учитывается знак), и 0 в противном случае.	ori r18, bitmask*(c1<c2)+1
10	<=	Меньше или равно. Возвращает 1 если левое выражение меньше или равно чем правое (учитывается знак), и 0 в противном случае.	ori r18, bitmask*(c1<=c2)+1
10	>	Больше чем. Возвращает 1 если левое выражение больше чем правое (учитывается знак), и 0 в противном случае.	ori r18, bitmask*(c1>c2)+1
10	>=	Больше или равно. Возвращает 1 если левое выражение больше или равно чем правое (учитывается знак), и 0 в противном случае.	ori r18, bitmask*(c1>=c2)+1
9	==	Равно. Возвращает 1 если левое выражение равно правому (учитывается знак), и 0 в противном случае.	andi r19, bitmask*(c1==c2)+1
9	!=	Не равно. Возвращает 1 если левое выражение не равно правому (учитывается знак), и 0 в противном случае.	.SET flag = (c1!=c2)═ ;Установить flag равным 1 или 0
8	&	Побитное И. Возвращает результат побитового И выражений.	ldi r18, High(c1&c2)
7	^	Побитное исключающее ИЛИ. Возвращает результат побитового исключающего ИЛИ выражений.	ldi r18, Low(c1^c2)
6	|	Побитное ИЛИ. Возвращает результат побитового ИЛИ выражении.	ldi r18, Low(c1|c2)

5	&&	Логическое И. Возвращает 1, если оба выражения не равны нулю, и 0 в противном случае.	ldi r18, Low(c1&&c2)
4	||	Логическое ИЛИ. Возвращает 1, если хотя бы одно выражение не равно нулю, и 0 в противном случае.	ldi r18, Low(c1||c2)

Функции. Определены следующие функции:

- LOW(выражение) возвращает младший байт выражения.
- HIGH(выражение) возвращает второй байт выражения.
- BYTE2(выражение) то же что и функция HIGH.
- BYTE3(выражение) возвращает третий байт выражения.
- BYTE4(выражение) возвращает четвёртый байт выражения.
- LWRD(выражение) возвращает биты 0-15 выражения.
- HWRD(выражение) возвращает биты 16-31 выражения.
- PAGE(выражение) возвращает биты 16-21 выражения.
- EXP2(выражение) возвращает 2 в степени (выражение).
- LOG2(выражение) возвращает целую часть log2(выражение).

7.3. ИСР AVR Studio

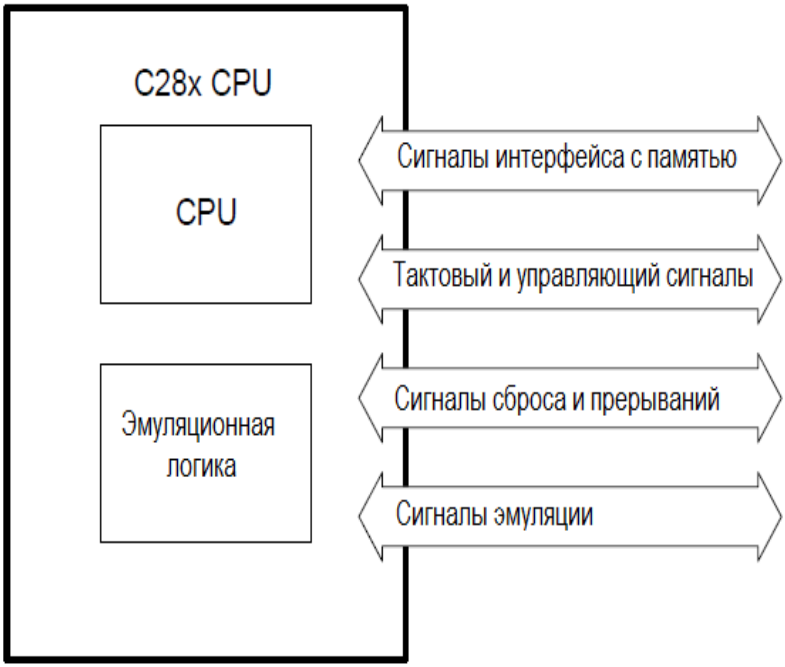
8. Микроконтроллеры C28x

8.1. Архитектура C28x

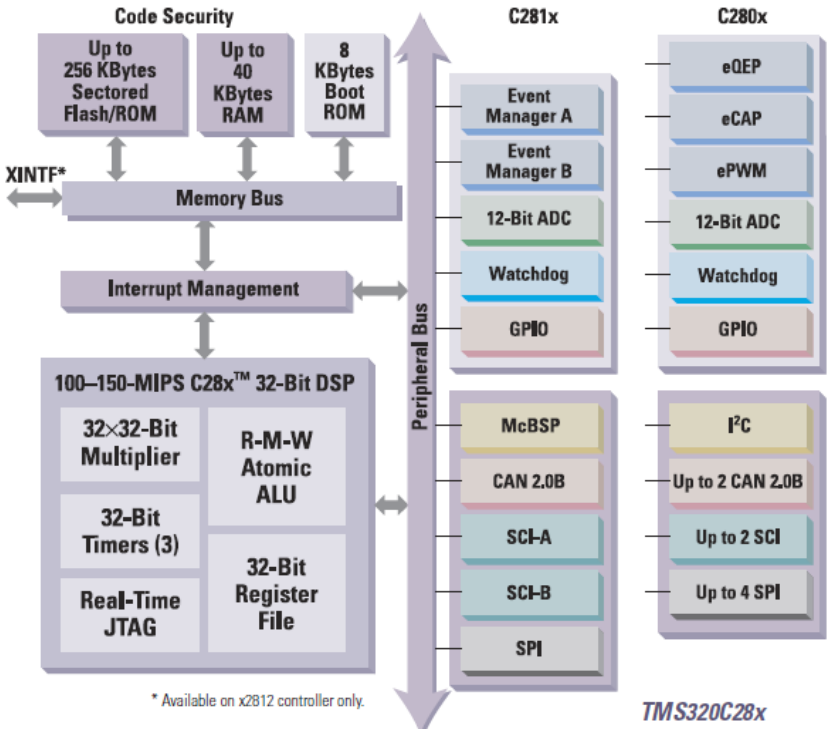
Это дешевый 32-разрядный цифровой сигнальный контроллер (DSC – Digital Signal Controller) с фиксированной точкой. В нем использована система команд RISC (с интуитивно понятными одноцикловыми командами), модифицированная гарвардская архитектура и циклическая адресация.

Высокоуровневая концептуальная модель включает:

- ЦП CPU для генерации адресов памяти программ и памяти данных, декодирования и исполнения инструкций, выполнения арифметических, логических и сдвиговых операций, и контроля передачи данных между регистрами CPU, памятью данных и программ.
- Эмуляционную логику, которая наблюдает за и управляет функциональностью DSC, а также выполняет тестирование.



- Блок схема C28x



Использует гарвардскую архитектуру с раздельными шинами программ и данных. Система команд – RISC (Reduced Instructions Set Computing). Содержит компоненты:

- Ядро C28x – 32-битный DSP, производительность 100-150 MIPS (миллионов инструкций в секунду).
- ПЗУ загрузчика (Boot ROM)
- ОЗУ до 40 Кбайт (RAM).
- Память программ ПЗУ до 256 Кбайт, либо перезагружаемое (Flash), либо программируемое на заводе (ROM).
- Блок управления прерываниями (Interrupt Management).
- Шина данных (Memory Bus).Периферийная шина (Peripheral Bus).
- 2 блока управления событиями (Event Manager A, B).
- Встроенный 12-разрядный аналого-цифровой преобразователь (ADC).

- Сторожевой таймер (Watchdog). Он для защиты от сбоев делает сброс программы через определенные интервалы времени.
- Многоканальный буферизированный интерфейс с последовательным портом McBSP (Multichannel Buffered Serial Port).
- Интерфейс с коммуникационным портом I^2C. Используется всего 2 линии – данные и синхронизация. К интерфейсу подключается любой уникально адресуемый источник.
- Интерфейсы с сетевым контроллером CAN (Control Area Network). Связывает множество клиентов общей сетью, использует буферы, фильтры данных и поддерживает работу с приоритетами.
- Масштабируемый когерентный интерфейс с коммуникационным портом SCI (Scalable Coherent Interface), обмен 16-битными словами.
- Последовательный периферийный интерфейс SPI (Serial Peripherial Interface). Надежный обмен данными по индивидуальным для каждого направления линиям.

Ядро C28x содержит:

- Умножитель 32x32 (Multiplier).
- 32-битные таймеры.
- Встроенный модуль отладки (JTAG реального времени).
- Атомарное АЛУ, выполняющее одновременно считывание, перемножение и запись результата.
- Регистровый файл с комплектом регистров.

В DSC используются регистры (находятся или в регистровом файле, или в других модулях):

- ACC – аккумулятор, 32 бита. Находится в АЛУ.
- AH – старшая часть ACC, 16 бит.
- AL – младшая часть ACC, 16 бит.
- P – произведение, 32 бита. Находится в умножителе.
- PH – старшая часть P, 16 бит.
- PL – младшая часть P, 16 бит.
- XT – регистр сомножителя, 32 бит Находится в умножителе.
- XAR0 – XAR7 – дополнительные регистры, 32 бита. Находятся в регистровом файле.
- AR – старшая часть XAR, 16 бит.
- PC - программный счетчик, 22 бита. Находится в ядре.

- RPC – возврат программного счетчика, 22 бита. Находится в ядре.
- DP – указатель на страницу данных, 16 бит. Находится в ядре.
- SP – указатель стека, 16 бит. Находится в ядре.
- STO0 –STO1 – статусные регистры, 16 бит. Находятся в ядре.
- IER – разрешение прерываний, 16 бит. Находится в ядре.
- IFR - флаг прерываний, 16 бит. Находится в ядре.

Главные особенности CPU:

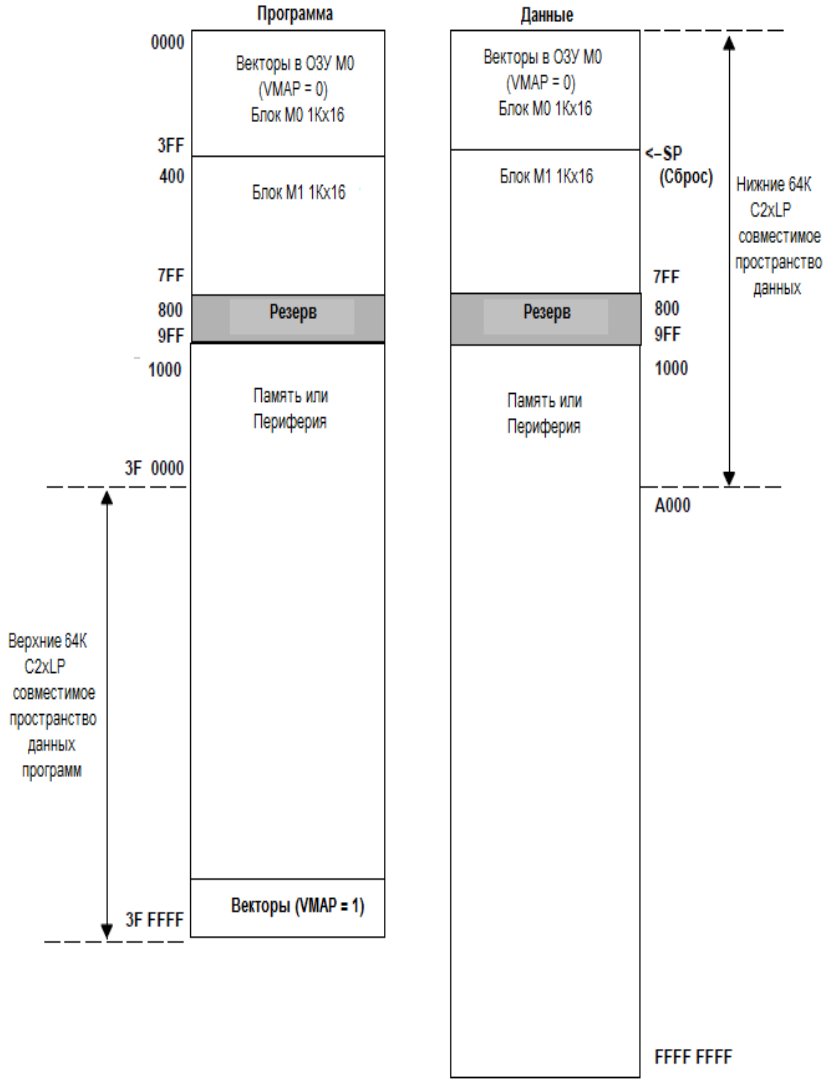
- Поддерживается 8-ступенчатый конвейер, который предотвращает запись или чтение по одному и тому же адресу в неправильном порядке.
- Имеется независимое пространство регистров. Оно не входит в пространство данных. Эти регистры используются как системные (доступны в специальных командах), математические и указатели на данные.
- 32-разрядное арифметико-логическое устройство (АЛУ), выполняет арифметику в дополнительном коде и логические операции.
- Арифметический модуль адресного регистра (Address register arithmetic unit - ARAU). Он работает параллельно с АЛУ, генерирует адреса памяти и увеличивает или уменьшает указатели.
- Циклический сдвигатель (Barrel shifter). Осуществляет сдвиги данных влево или вправо на 1-16 бит.
- Умножитель (Multiplier). Реализован аппаратно, выполняет умножение 32x32 в дополнительном коде с 64-разрядным результатом. Сомножители могут быть как знаковые, так и без знака.

Главные особенности эмуляционной логики:

- Прямой доступ к памяти в отладочном и тестовом режимах (Debug-and-test direct memory access - DT-DMA).
- Запрос данных.
- Счетчик для реализации закладок.
- События отладки. Следующие события вызывают сброс устройства: команды ESTOP0 и ESTOP1, обращение к определенной области памяти программ или памяти данных, запрос от внешнего устройства. Когда событие возникает, CPU переходит в режим ожидания.
- Работа в режиме реального времени.

Главные сигналы CPU:

- Интерфейс с памятью. Они используются для передачи данных между CPU, памятью и периферией, индицируют доступ к памяти программ и памяти данных. Возможен доступ в режимах 16 или 32 бита.
- Тактовый и управляющий сигналы. Обеспечивают тактирование CPU и эмуляционной логики, а также используются при контроле и управлении CPU.
- Сигналы сброса и прерываний. Осуществляют аппаратный сброс или обращение к процедуре обработки прерывания, а также для контроля статуса прерывания.
- Эмуляционные сигналы используются при отладке и тестировании.
- Карта памяти



В C28x используется память из 16-разрядных слов. На кристалле находятся:

Память программ. Адрес задается 22-разрядным регистром программного счетчика (Program Counter - PC). Это позволяет иметь в памяти программ 4M слов (слово = 16 бит). По нижним 2K адресам $0000…$7FF определены 2 блока по 1K: блок векторов прерываний VMAP=0 (адреса $000…$3FF), блок M1 для кода программы (адреса $400…$7FF). Адреса $800…$9FF зарезервированы. По адресам $1000…$3FFFFF расположена область памяти и периферия. Верхние 64K (адреса $3F0000…$3FFFFF) этой области занимает сегмент, совместимый с режимом C2xLP. В нем по верхним адресам имеется блок VMAP=1 векторов прерываний.

Память данных. Использует страничную организацию, номер страницы хранится в 16-разрядном регистре (Data Page Pointer – DP), всего страниц 64K=2^{16}. Размер страницы 64K. Всего память данных может содержать до 4Г=64K*64K слов (слово = 16 бит). Нижние 64K (адреса $0000…$A000) этой области занимает сегмент, совместимый с режимом C2xLP. По нижним 2K адресам $0000…$7FF определены 2 блока по 1K: блок векторов прерываний VMAP=0 (адреса $000…$3FF), блок M1 для данных (адреса $400…$7FF). Адреса $800…$9FF зарезервированы. По адресам $1000…$FFFFFFFF расположена область памяти и периферия.

Память имеет отдельные шины для пространства программ и пространства данных. Это означает, что инструкция может быть вызвана, когда данные уже готовы. Возможен доступ к данным размером 16 и 32 бита. Имеются также инструкции доступа к младшему или старшему байту слова данных. Имеются 3 адресные шины:

- PAB (Program address bus) - шина адреса памяти программ для выборки кода программы.
- DRAB (Data-read address bus) - шина адреса памяти даных для чтения данных.
- DWAB (Data-write address bus) - шина адреса памяти данных для записи данных.

Имеются 3 шины данных:

- PRDB (Program-read data bus). Шина, на которую считывается код из памяти программ.
- DRDB (Data-read data bus). Шина, на которую считываются данные из памяти данных
- DWDB (Data-write data bus). Шина, с которой данные записываются в память данных или в память программ.

Для временного хранения данных предусмотрена стековая память, организованная по правилу «последним пришел первым ушел» (Last In First Out - LIFO). . Доступ к стеку через указатель стека (Stack Pointer – SP). Его содержимое – адрес верхушка стека.

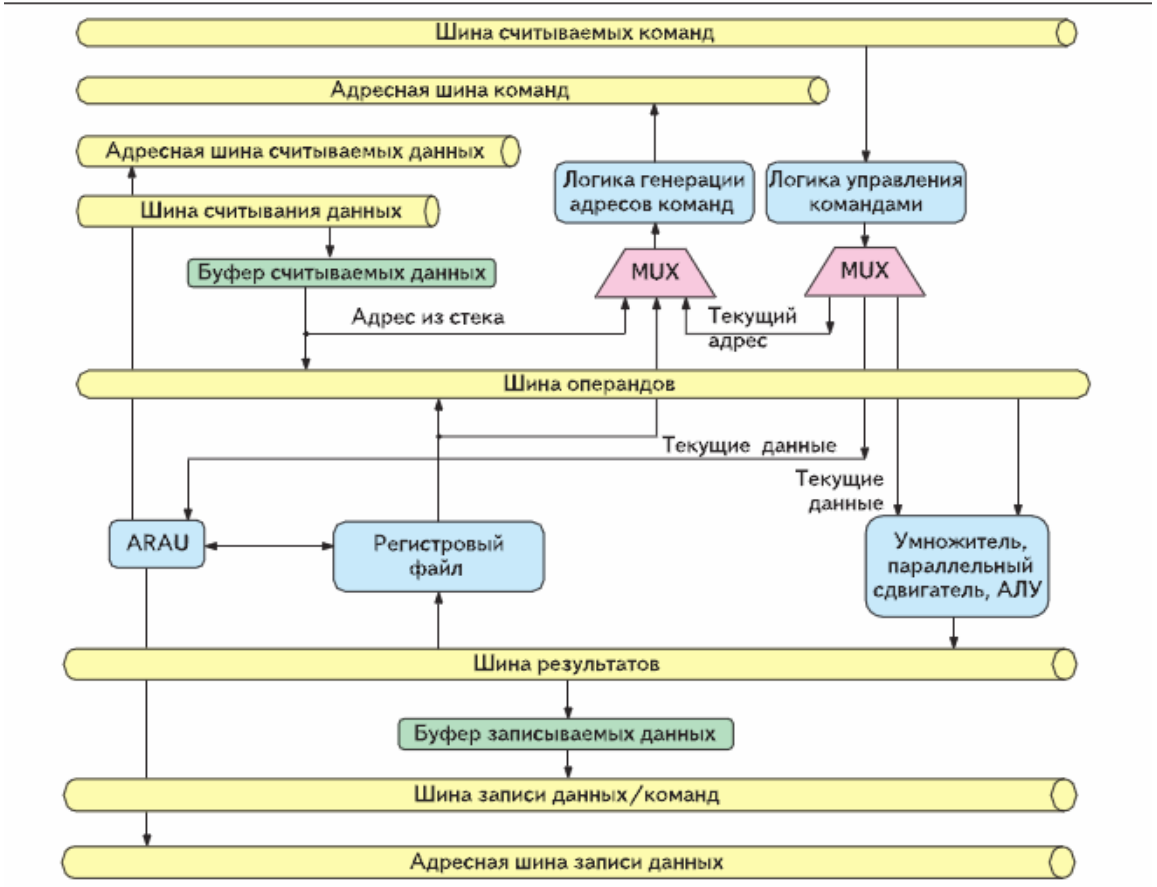
Стек SP размещается в памяти данных. Стек растет в сторону увеличения адреса. Указатель стека SP хранит адрес верхушки стека. SP имеет 16 разрядов, поэтому максимальный адрес в SP равен $FFFF и размер стека не может превышать этого значения. Стек может использовать адреса в диапазоне от $0000 до $FFFF.

Когда 32-разрядное значение сохраняется в стеке, то сначала сохраняется старшая часть, а по следующему адресу – младшая часть.

Методы адресации. Поддерживаются 4 метода адресации:

- **Прямая**. Использует 16-разрядный DP (Data Page) регистр. Его содержимое – начальный адрес сегмента памяти. В инструкциях для адресации в пределах сегмента добавляется поле смещения (6 или 7 бит). Используется для адресации к структурам.
- **Стековая**. Использует 16-разрядный регистр указателя стека (Stack Pointer – SP). Его содержимое – адрес верхушка стека. В инструкциях для адресации в пределах стека добавляется поле смещения (6 бит), которое вычитается из адреса верхушки стека. Используется для адресации к стековой памяти.
- **Косвенная (не прямая).** Использует 32-разрядные вспомогательные регистры XAR0…XAR7 в качестве указателей памяти данных. В инструкциях можно прямо использовать содержимое выбранного регистра с полем смещения (3-битовая константа или содержиое 16-разрядного регистра). Альтернатива - пост-инкремент, пред/пост-декремент для автоматического изменения адреса.
- **Регистровая**. Использует пару регистров, один источник данных, другой приемник.

Концептуальная блок-схема CPU.

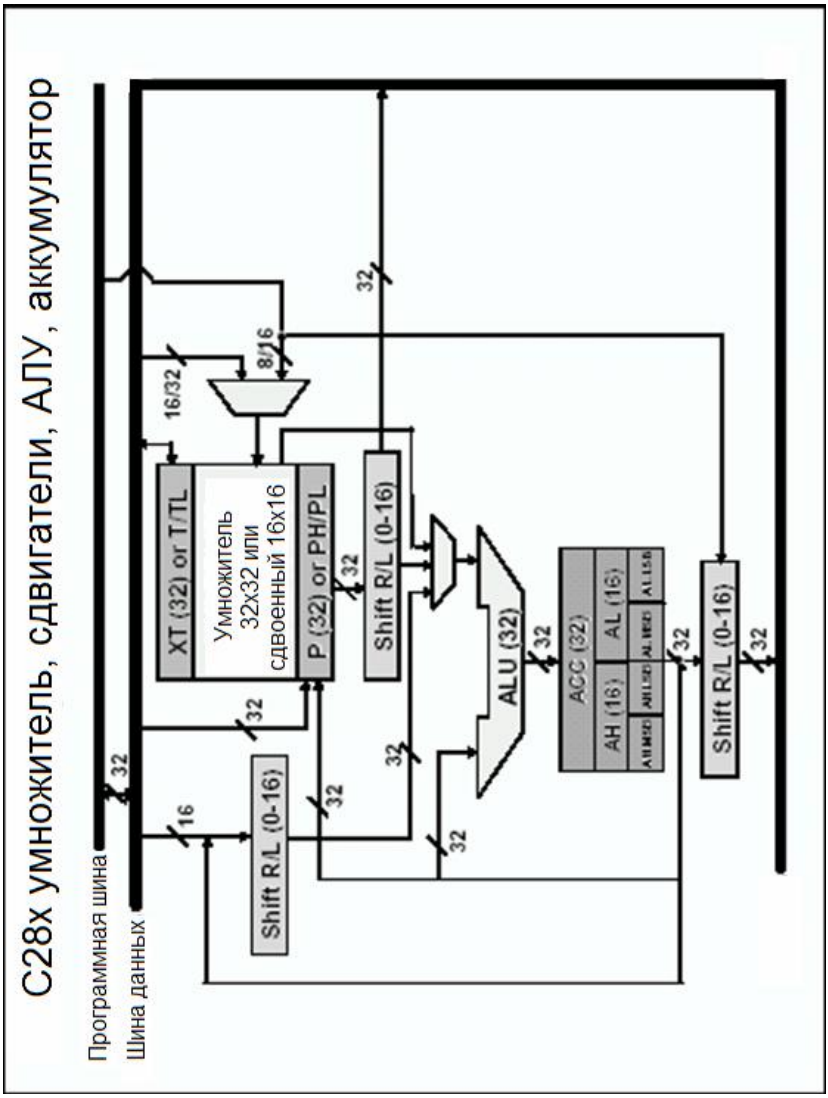


Архитектура ядра C28x

На схеме представлены::

- 32-разрядная шина считываемых команд. На нее считываются команды из памяти программ.
- 22-разряднаяадресная шина команд.
- 32-разрядная адресная шина считываемых данных.
- 32-разрядная шина считываемых данных. На нее считываются данные из памяти данных.
- Буфер считываемых данных. Это регистр для хранения считанных данных. Его выход передается на шину операндов, либо на генератор адреса памяти программ, если данные - это адрес стека. Это первый операнд при выполнении команды.
- Шина операндов. На ней находится первый операнд при выполнении команды.
- Шина результата операции в исполнительном устройстве.
- Буфер записываемых данных. Это регистр данных, записываемых в память данных. Он загружается с шины результатов операции Result Bus.
- Логика управления командами. Декодирует считанную команду и передает данные исполнительным устройствам.
- Логика генерации адресов команд.
- Арифметический модуль адресного регистра ARAU. Вычисляет адреса для регистрового файла и адресной шины считываемых данных.
- Регистровый файл, который содержит набор регистров (системные и общего назначения).
- Исполняющее устройство, включает - умножитель, параллельный сдвигатель, АЛУ. Умножитель реализован аппаратно, выполняет умножение 32x32 или два умножения 16x16.

Исполнительное устройство.



Основные регистры. Определен комплект регистров. Основные регистры:

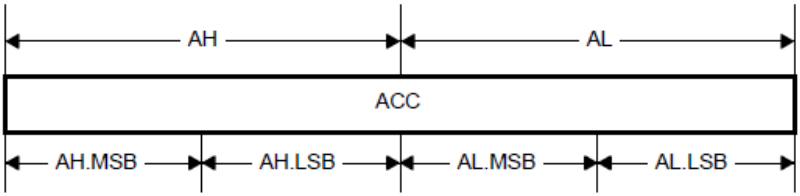
Регистр	Размер бит	Описание

ACC	32	Аккумулятор
AH	16	Старшая часть ACC
AL	16	Младшая часть ACC
XAR0	32	Дополнительный регистр 0
XAR1	32	Дополнительный регистр 1
XAR2	32	Дополнительный регистр 2
XAR3	32	Дополнительный регистр 3
XAR4	32	Дополнительный регистр 4
XAR5	32	Дополнительный регистр 5
XAR6	32	Дополнительный регистр 6
XAR7	32	Дополнительный регистр 7
AR0	16	Младшая часть XAR0
AR1	16	Младшая часть XAR1
AR2	16	Младшая часть XAR2
AR3	16	Младшая часть XAR3
AR4	16	Младшая часть XAR4
AR5	16	Младшая часть XAR5
AR6	16	Младшая часть XAR6
AR7	16	Младшая часть XAR7
DP	16	Указатель на страницу памяти данных
IFR	16	Регистр флагов прерывания
IER	16	Регистр разрешения прерывания
DBGIER	16	Регистр разрешения прерывания при отладке
P	32	Регистр произведения
PH	16	Старшая часть P
PL	16	Младшая часть P
PC	22	Программный счетчик
RPC	22	Возврат значения программного счетчика
SP	16	Указатель стека
ST0	16	Статусный регистр ST0
ST1	16	Статусный регистр ST1
XT	32	Регистр сомножителя
T	16	Старшая часть XT
TL	16	Младшая часть XT

Аккумулятор (ACC). Главный регистр – аккумулятор (ACC). Он хранит результаты операций в АЛУ. Он поддерживает одноцикловые операции перемещения,

сложения, вычитания и сравнения с данными из 32-разрядной памяти данных. Он также принимает 32-разрядный результат от умножителя.

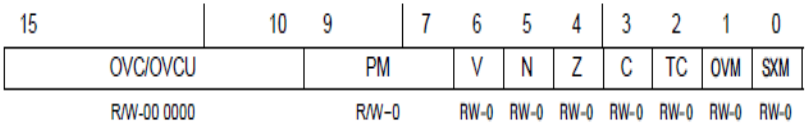
Для ACC возможен доступ и к части его содержимого: AH – старшая часть, AL – младшая часть. Для каждого из них можно получать младший LSB или старший MSB байты.



AH = ACC (31:16)
AH.MSB = ACC (31:24)
AH.LSB = ACC (23:16)

AL = ACC (15:0)
AL.MSB = ACC (15:8)
AL.LSB = ACC (7:0)

Статусный регистр ST0. В нем запоминаются флаги состояния ACC



Флаги (по умолчанию все сброшены):

- Расширение знака (Sign-extension mode - SXM), бит 0. Если SXM=1, то при загрузке 16-разрядного числа со знаком в 32-разрядный ACC оно перед загрузкой расширяется по знаку (пустые биты заполняются битом знака) до 32-разрядного представления.
- Переполнение (Overflow mode -OVM), бит 1. Если OVM = 0 (режим переполнения выключен), то переполнения считаются в поле OVC в ACC. Если OVM = 1 (режим переполнения включен), то переполнения не подсчитываются, в ACC заносятся значения насыщения.
- Флаг теста/контроля (Test/control - TC), бит 2.
- Перенос (Carry - C), бит 3.
- Флаг нуля (Zero flag - Z), бит 4. Если результат операции равен 0, то Z=1.

- Флаг отрицательного значения (Negative - N), бит 5. Если результат операции отрицательное число, то N=1.
- Флаг запрета переполнения (Latched overflow flag - V), бит 6. Если результат операции вызывает переполнения в регистре хранения результата, то V=1 и защелкивается. Если нового переполнения нет, то V не меняется, остается защелкнутым от предыдущего переполнения.
- Поле режима сдвига произведения (Product shift mode bits - PM). Поле в битах 7…9. 3-разрядное значение определяет режим сдвига для значения, получаемого из регистра произведения P. Возможные режимы:.

PM	Результат
PM=000	Сдвиг влево на 1, в младший бит заносится 0. Режим по умолчанию.
PM=001	Нет сдвига.
PM=010, 011, 100, 101, 110, 111	Сдвиг вправо на PM-1 бит, младший бит теряется, в старших битах расширение знака.

- Счетчик переполнения операций с числами со знаком (Overflow counter - OVC). Для чисел без знака OVCU. Поле в битах 10…15. Счетчик 6-разрядный со значениями от -32 до 31. При OVC=0 содержимое ACC не меняется. При OVC>0 содержимое ACC = насыщение по максимуму ($7FFFFFFF), OVC = 0. При OVC<0 содержимое ACC = насыщение по минимуму ($8000000), OVC = 0.
- Статусный регистр ST1

В нем запоминаются флаги состояния разных узлов.

15 … 13	12	11	10	9	8
ARP	XF	M0M1MAP	Reserved	OBJMODE	AMODE
R/W-000	R/W-0	R/W-1	R/W-0	R/W-0	R/W-0

7	6	5	4	3	2	1	0
IDLESTAT	EALLOW	LOOP	SPA	VMAP	PAGE0	DBGM	INTM
R-0	R/W-0	R-0	R/W-0	R/W-1	R/W-0	R/W-1	R/W-1

Флаги (по умолчанию все сброшены):

- Флаг глобальной маски прерывания (Inerrupt global mask - INTM), бит 0. Если INTM=1, то маскированное прерывание запрещено.
- Флаг массирования отладки (Debag enable maskt -DBGM), бит 1. Если DBGM = 1, то отладка запрещена.

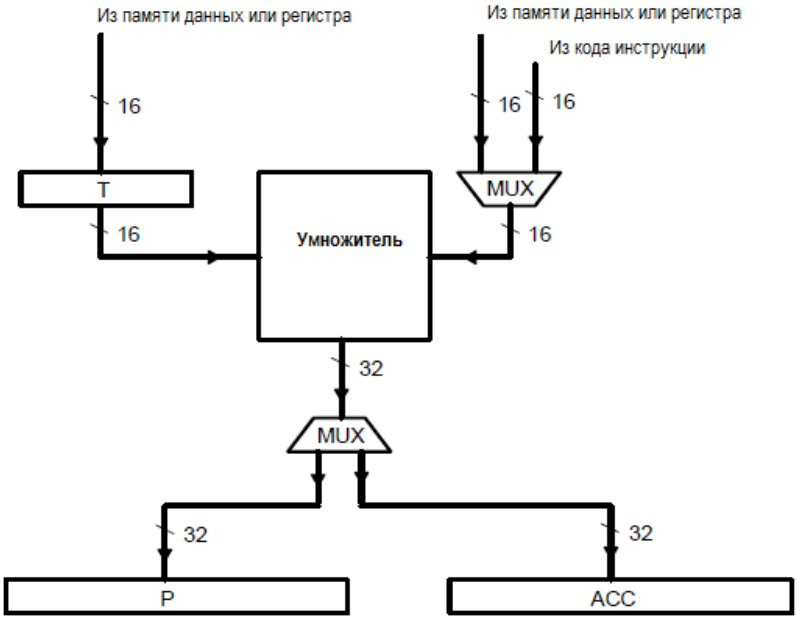
- Флаг режима адресации (PAGE0). Размещен в бите 2. При PAGE0=0 – адресация через стек, при PAGE0=1 – прямая.
- Флаг размещения вектора прерываний (VMAP), бит 3. При VMAP=0 вектор внизу памяти программ (адреса $000000…$000003F), при VMAP=1 вектор вверху памяти программ (адреса $3FFFC0…$3FFFFF).
- Флаг размещения указателя стека (Stack pointer alignment - SPA), бит 4. При SPA=0 указатель стека SP не может занимать четные адреса.
- Статус инструкции цикла (LOOP), бит 5. При LOOP=1 то под операцию LOOP отводится два этапа конвейера.
- Флаг разрешения эмуляции (EALLOW), бит 6. При EALLOW=1 разрешены эмуляция и доступ к защищенным регистрам.
- Флаг режима только чтение (IDLESTAT), бит 7. Устанавливается IDLESTAT=1 при выполнении инструкции IDLE. Флаг сбрасывается командами, отменяющими этот режим.
- Флаг режима адресации (AMODE), бит 8.
- Флаг совместимости режима (OBJMODE), бит 9. При OBJMODE=0 режим C27x, при OBJMODE=1 режим C28x.
- Флаг размещения M0, M1 (M0M1MAP), бит 11. Для C28x.обязательно M0M1MAP=1.
- Статус выходного сигнала XFS (XF), бит 12. При XF=1 совместимость с C2xLP.
- Указатель на дополнительные регистры (Auxiliary register pointer.- ARP). Поле размещено в битах 13-15. В него заносится 3-разрядный номер регистра (0-7).
- Умножитель

C28x содержит встроенный аппаратный умножитель, выполняющий умножение за 1 такт. Возможны:

- умножение16x16
- умножение16x16 с накоплением MAC
- умножение 32x32
- умножение 32x32 с накоплением DMAC

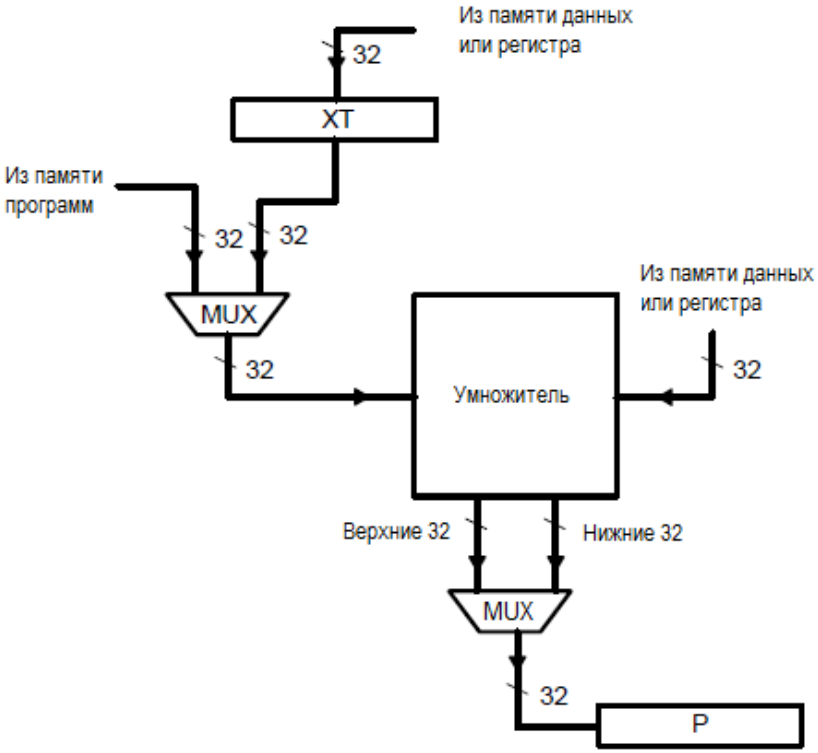
Концептуальная блок-схема умножителя в режиме 16x16. Умножитель получает 16-разрядные сомножители. Первый заносится в регистр T (старшая часть XT)из памяти данных или регистра. Второй извлекается из мультиплексора MUX, который получает значение из памяти данных или регистра, либо из кода инструкции.

32-разрядное произведение через MUX заносится в регистр произведения P (и может использоваться как операнд) и в аккумулятор ACC (для выполнения операций MAC).



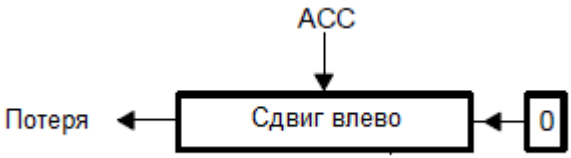
Концептуальная блок-схема умножителя в режиме 32x32. Умножитель получает 32-разрядные сомножители. Первый заносится прямо из памяти данных или регистра Второй извлекается из MUX, который получает значение из памяти программ или регистра XT, который загружается из памяти данных или регистра.

64-разрядное произведение содержит 32-разрядные части (старшую и младшую). Одна из них через MUX и заносится в регистр произведения P.

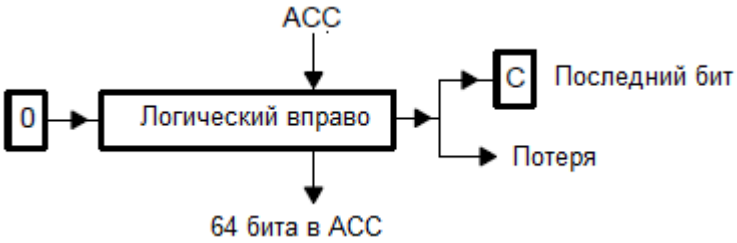


Сдвигатель. Сдвигатель имеет 64 разряда и поддерживает операции сдвига влево и вправо. Входные операнды могут иметь длину 16, 32 или 64 разряда. Возможные сдвиги:

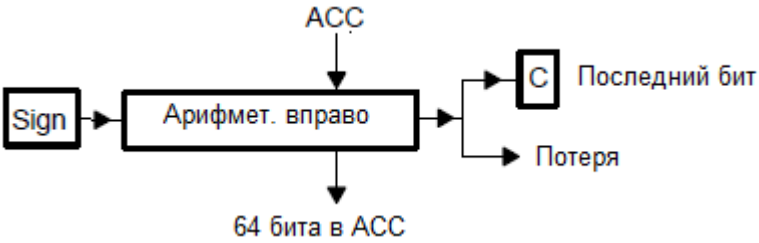
- **Логический влево**. Старшие биты теряются, младшие заполняются нулями.



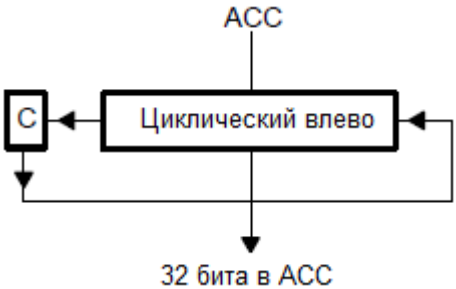
- **Логический вправо**. Младшие биты теряются, старшие заполняются нулями. Последний выдвигаемый бит заносится в бит переноса C.



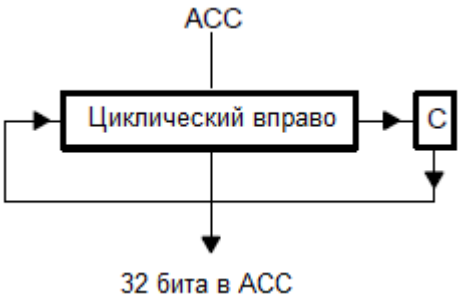
- **Арифметический вправо**. Младшие биты теряются, последний выдвигаемый бит заносится в бит переноса C. В старших расширение бита знака Sign.



- **Циклический влево**. Старшие биты по цепочке переносятся в младшие (в цепочку входит бит переноса C).



- **Циклический вправо**. Младшие биты по цепочке переносятся в старшие (в цепочку входит бит переноса C).



Прерывания и приоритеты. Прерывание это сигнал, генерируемый аппаратно или программно, по которому ЦП останавливает текущую программу и переходит к выполнению подпрограммы обработки прерывания. Обычно прерывания генерируются периферийными или аппаратными блоками, для того чтобы получить данные от ЦП.

Различаются прерывания:

- Маскируемые. Они могут быть блокированы или отменены другими устройствами. Каждому маскируемому прерыванию присваивается приоритетный номер (номер ниже – приоритет выше). Маскируемое прерывание выполняется, если нет запросов прерывания с более высокими приоритетами.
- Немасккируемые. Они не могут быть блокированы или отменены другими устройствами.

Обработка прерывания включает 4 фазы:

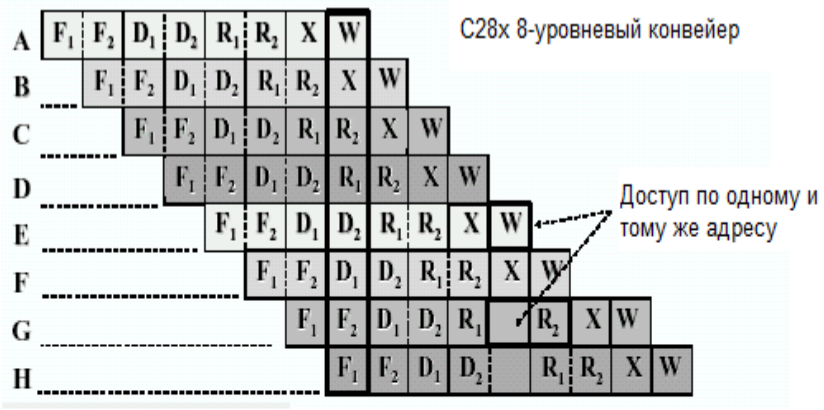
- Прием запроса прерывания.
- Обработка запроса. Если прерывание маскируемое, то нужно проверить условия разрешения прерывания. Немаскируемое прерывание должно выполняться немедленно.
- Подготовка к обработке прерывания. Завершить текущую инструкцию, выполнить операции в конвейере, сохранить контекст программы (значения регистров, текущее состояние счетчика команд). Извлечь из вектора прерывания адрес процедуры обработки и загрузить его в программный счетчик.
- Запустить процедуру обработки прерывания. После завершения происходит возврат в текущую программу.

C28x поддерживает 32 вектора прерывания : 19 приоритетных прерываний с приоритетами 1…19 (19 – нижний), 12 неприоритетных прерывания пользовате-

ля, 1 прерывание при неправильной инструкции. Каждый вектор включает стартовый адрес процедуры обработки прерывания (Interrupt Service Routine - ISR). Он накапливается в двух соседних словах памяти.

Конвейер. Для повышения производительности C28x использует защищенный 8-уровневый конвейер. На каждом уровне конвейера может быть команда для аппаратного блока ЦП (F1, F2, D1, D2, R1, R2, X, W). Конвейер работает независимо от действий программиста. Он управляется командами, заложенными в ЦП.

Конвейер предотвращает возможность записи и чтения по одному и тому же адресу в порядке, не предусмотренном программистом. Если такое происходит, то ЦП вставляет в конвейер пустые операции.



F1: установить адрес инструкции на шине адреса программ
F2: получить инструкцию
D1: декодировать инструкцию
D2: сформировать адреса операндов
R1: установить адрес операнда на шине данных
R2: получить операнд
X: исполнить инструкцию
W: сохранить результат

На рисунке выделен интервал времени, когда одновременно работают 8 блоков ЦП. За счет распараллеливания конвейер позволяет C28x работать на высокой скорости, не прибегая к использованию дорогой высокоскоростной памяти. Спе-

циальное устройство предсказания переходов минимизирует задержку на условных переходах. Специальным образом организованные переходы еще больше увеличивают производительность.

8.2. Архитектура F28x

В нем поддерживаются операции с плавающей точкой.

F28x = C28x + Flash.

Имеются дополнительные регистры:

Регистры R0H – R7H используются для хранения результатов операции с плавающей точкой, 32 бита.

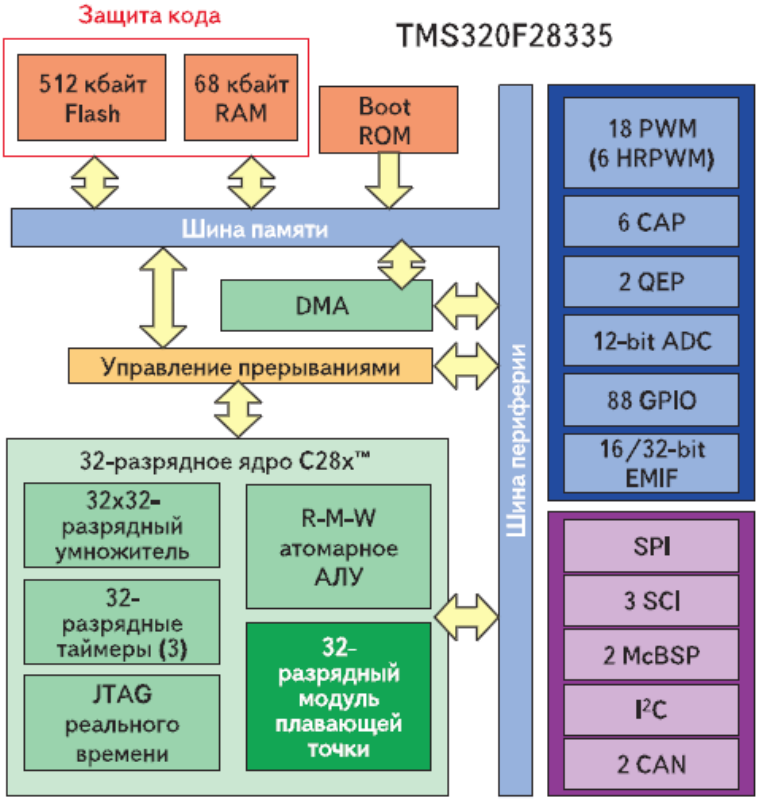
Регистр FPU Status Register (STF) - регистр статуса блока с плавающей точкой, 32 бита. Его структура:

31	30 -10	9	8 - 7	6	5	4	3	2	1	0
SDWS	резерв	RND32	резерв	TF	ZI	NI	ZF	NF	LUF	LVF
		бит округления		Флаг теста	Флаг 0 целого числа	Флаг <0 целого числа	Флаг 0 числа с ПТ	Флаг <0 числа ПТ	Флаг переполнения	Флаг переполнения

Регистр блока повторов (Repeat Block - RB), 32 бита. Его структура:

31	30	29 - 23	22 - 16	15 - 0
RAS – бит прерывания	RA – бит активности блока (0 или 1)	RSize – размер блока повторения	RE – адрес конца	RC – счетчик повторов

Блок-схема.



Функциональная схема контроллера TMS320F28335

Ядро процессора с производительностью 300MFLOPs при частоте 150МГц включает:

- 32x32 аппаратный умножитель.
- 32-разрядные таймеры (3).
- Встроенный модуль отладки (JTAG реального времени).
- Атомарное АЛУ, выполняющее короткие RISC инструкции.
- 32-разрядный модуль умножитель с ПТ.

Подсистема памяти:

- Flash до 512 Кбайт.
- ОЗУ (RAM) 68 Кбайт.
- Загрузочное ПЗУ (ROM).
- Интерфейс EMIF.
- 6 каналов прямого доступа к памяти (DMA).

Периферийный модули:

- 18 модулей ШИМ.
- Высокоскоростной встроенный АЦП.
- 6 модулей захвата (CAP).
- 88 выводов общего назначения.
- 2 канала McBSP с возможностью конфигурирования в режим SPI.
- Порт CAN 2.0b с 32 почтовыми ящиками (mailboxes).
- Интерфейс PC со скоростью 480 кбит/с.
- 2 импульсных квадратурных декодера (QEP).

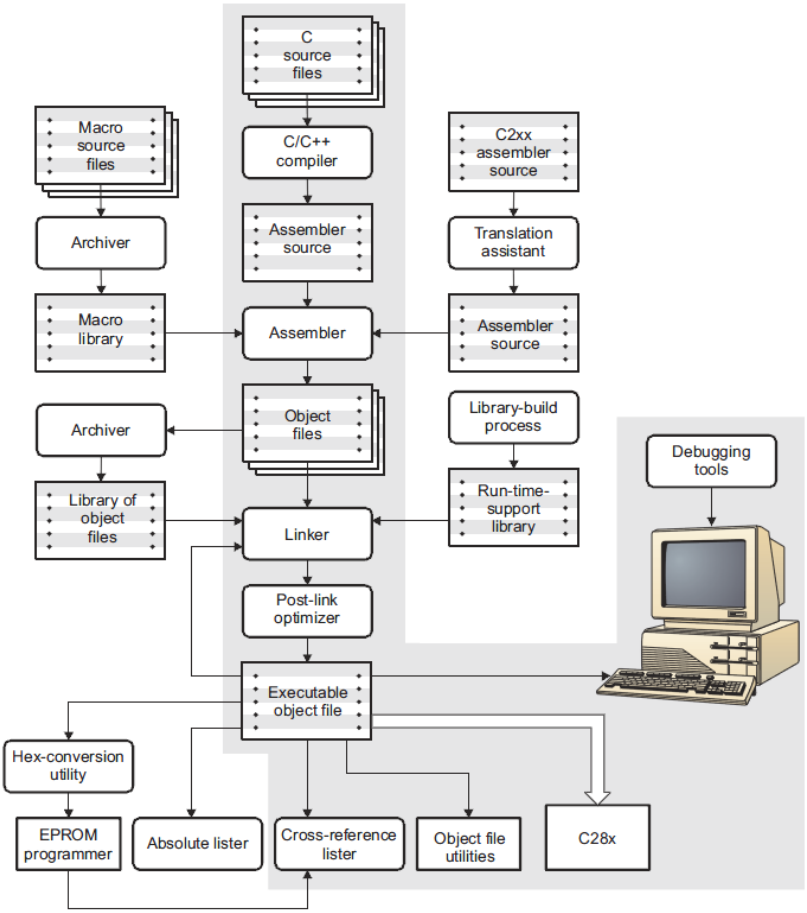
Питание:

- Ядро 1,9 В.
- 3,3 В периферия.

8.3. Инструментальные средства разработки ПО

Основным продуктом процесса разработки является модуль, который может выполняться устройством. Вы можете использовать одно из различных инструментальных средств отладки для обогащения и корректирования вашего программного кода. Доступные продукты включают: программный симулятор с тщательно разработанными инструкциями и синхронизацией, XDS эмулятор.

На рисунке показан процесс разработки программного обеспечения C28. Затенением выделен наиболее общий маршрут разработки; другие части дополнительные. Последние являются периферийными функциями, которые расширяют процесс разработки.



Инструментальные средства:

- **Компилятор C/C++** преобразует C-исходник (C source) в исходник ассемблера (Assembler source). Выход – файл исходника ассемблера.
- **Линейный ассемблер** (Linear Assembler) – исходник ассемблера, который создается программистом без учета возможного распараллеливания команд.

- **Ассемблерный оптимизатор** (Assembly optimizer) переводит линейный ассемблер в высоко параллельный ассемблер. Выход – файл оптимизированного ассемблера (Assembly-optimized file). Программист может делать распараллеливание вручную.
- **Ассемблер** (Assembler) переводит код ассемблера в машинный код. Код ассемблера может содержать инструкции, директивы ассемблера и макро директивы. Код ассемблера может быть получен из файлов исходника ассемблера, библиотек макросов, оптимизированного ассемблера.
- **Компоновщик** (Linker) объединяет объектные файлы в единый выполняемый ООФФ объектный модуль. Компоновщик принимает перемещаемые ООФФ объектные файлы (созданные ассемблером) как вход. Он также принимает элементы библиотеки архиватора и выходные модули, созданные предшествующим запуском компоновщика.
- **Архиватор (Archiver)** позволяет собирать группу файлов в единый архивный файл, называемый библиотекой. Например, Вы можете собрать различные макроопределения в макробиблиотеку. Ассемблер находит библиотеку и использует элементы, которые названы как макроопределения исходного файла. Архиватор позволяет модифицировать библиотеку, удаляя, заменяя, извлекая, или добавляя её элементы.
- **Утилита шестнадцатеричного преобразования** (Hex conversion utility) преобразовывает ООФФ объектный файл в объектный формат стандартных программаторов перепрограммируемого постоянного запоминающего устройства (EPROM).
- **Абсолютный листер** (Absolute lister) использует объектные файлы для реализации абсолютных ссылок, показанных символами списка, их (файлов) определение.
- **Листер перекрестных ссылок** (Cross-reference lister) использует объектные файлы для реализации перекрестных ссылок, показанных символами списка, их (файлов) определение и в результате связывает (формирует) исходные файлы.

8.4. Ассемблер

Ассемблер преобразовывает (транслирует) исходные файлы ассемблера в объектные файлы в машинном коде. Эти файлы находятся в общем формате объектного файла (COFF). Исходные файлы могут содержать следующие элементы ассемблера:

- Директивы Ассемблера.
- Макро директивы.

- Команды ассемблера.

Двухпроходовый ассемблер делает следующее:

- Преобразует операторы исходника в объектный файл.
- Создает листинг исходника (если требуется) и дает Вам возможность управлять им.
- Позволяет Вам сегментировать код по разделам и устанавливает счетчик команд SPC в каждом разделе объектного кода.
- Определяет глобальные символы и ссылки на них, создает перекрестные ссылки на листинг исходника (если требуется).
- Допускает условное ассемблирование.
- Допускает макросы, позволяя создавать макросы в исходнике или в библиотеке.

Вызвать ассемблер можно двумя способами:

- Запустить ASM. Ассемблер запускается с опциями по умолчанию. В частности без опции –L, которая заставляет его формировать файл листинга.
- В командной строке ОС ввести путь к файлу ASM, имя файла и опцию –L. Файл листинга будет сформирован, и допущенные ошибки можно увидеть.

В окне ассемблера нужно задать:

- **Source file** - исходный файл ассемблера. Если Вы не даете расширение, ассемблер использует заданное по умолчанию расширение .asm.
- Формат инструкций исходника

Исходник ассемблера C28x состоит из инструкций, которые могут содержать директивы ассемблера, команды ассемблера, макро-директивы, и комментарии. Инструкция может содержать 5 упорядоченных полей (метка, признак параллельности ||, мнемоника, список операндов, комментарий).

Примеры инструкций:

```
two     .set    2    ; Символ two = 2
label:  MVK     two, A2     ; Запись значения two в регистр A2
.word   016h                ; Инициализация слова значением 016h
```

Ассемблер читает до 200 знаков в строке. Любые знаки свыше 200 усекаются. Операционная часть инструкций (т.е. все, кроме комментариев) должна быть короче 200 знаков для правильной трансляции. Комментарии могут простираться за пределы 200 знаков, но усеченная часть не включается в файл листинга.

Следуйте этим рекомендациям:

- Все инструкции должны начаться с метки, разделителя (пробел или табулятор), звездочки, или точки с запятой.
- Метки не обязательны, если они используется, они должны начаться в столбце 1.
- Один (или больше) разделителей должно отделять каждое поле. Символы табуляции интерпретируются, как пробелы. Вы должны отделить список операндов от предшествующего поля пробелом.
- Комментарии необязательны. Комментарии, которые начинаются в столбце 1, могут начинаться со звездочки или точки с запятой (* или ;). Комментарии, которые начинаются в любом другом столбце, должны начинаться с точки с запятой.
- Мнемоника не может начинаться в столбце 1, иначе она будет интерпретироваться, как метка.

Метки. Они необязательны для всех команд ассемблера и для большинства (но не всех) директив ассемблера. Когда используется, метка должна начаться в столбце 1 инструкции. Метка может содержать до 128 алфавитно-цифровых знаков (A-Z, a-z, 0-9, _ и $). Метки **чувствительны к регистру**, и первый знак не может быть числом. Метка может сопровождаться двоеточием (:).

Если Вы не используете метку, знак в столбце 1 должен быть разделителем, звездочкой, или точкой с запятой.

Признак параллельности. Символы || указывают команды, которые выполняются параллельно с предыдущей командой. Вы можете иметь до 2 команд, выполняющихся параллельно. Следующий пример демонстрирует 2 команды (Inst1…Inst2), выполняющихся параллельно:

```
Inst1
|| Inst2
Inst3
```

Мнемоника. Может содержать

- Мнемоника инструкции (например, ADD, MOV).
- Директива ассемблера (например, .data, .list, .equ).
- Директива макроса (например, .macro, .var, .mexit).
- Вызов макроса (подпрограммы).

Поле операнда. Оно следует за мнемоническим полем и содержит один или большее количество операндов. Поле операнда требуется не для всех команд или директив. Операнд состоит из следующих элементов: символы, константы выражения (комбинация констант и символов). Операнды друг от друга отделяются запятыми (никаких разделителей!).

Комментарий. Может начинаться в любом столбце и простирается до конца исходной строки. Комментарий может содержать любые знаки ASCII, включая пробелы. Комментарии печатаются в листинге программы ассемблера, но не влияют на процесс трансляции.

Исходная инструкция, которая содержит только комментарий, допустима. Такой комментарий – заголовок части кода. Если она начинается в столбце 1, то может начинаться с точки с запятой (;) или звездочки (*).

Комментарии, которые начинаются где-нибудь еще на строке, должны начинаться с точки с запятой. Такой комментарий описывает операцию в строке кода.

Константы. Ассемблер поддерживает каждую константу внутренне, как 32-разрядное число. Константы – не расширяются по знаку. Например, константа 00FFh равна 00FF (в 16-ричной системе) или 255 (в десятичной)..

Ассемблер поддерживает 7 типов констант:

- Двоичное целое. Это строка из 32 двоичных символов (0, 1), которая завершается символом B (или b). Если указано меньше 32 символов, ассемблер дополняет ведущие нули. Например, 0100000b это десятичное 32.
- Восьмеричное целое. Это строка из 11 восьмеричных символов (от 0 до 7), которая завершается символом Q (или q). Например, 10q это десятичное 8. В режиме C2xlp восьмеричное целое не поддерживаются.
- Десятичное целое. Это строка из десятичных символов (от 0 до 9), которая перекрывает диапазон от -2147 483 648 до 4 294 967 295. Например, 25 это десятичное 25.
- 16-ричное целое. Это строка из 8-ми 16-ричных символов (от 0 до 9, A - F), которая завершается символом H (или h). Например, 78h это десятичное 120.
- Символьная константа. Это одиночный символ, помещенный в одиночные кавычки. Внутреннее представление это 8-битный номер символа ASCII. Например, '' – символ с номером 0, 'a' – символ с номером 97.

- Константа ассемблера. Это численное представления символа. Позволяет применять символы в выражениях.
- Константа с плавающей точкой. Это строка из десятичных символов в формате: знак числа (+ или -), значащая часть (целая часть, точка, дробная часть), символ степени E(или e), знак порядка, порядок. Например, +314.59e-2 - это π.

Обратите внимание на различие между символьными константами и символьными строками. Символьная константа представляет одиночное значение; строка - последовательность знаков.

Вы можете также использовать .set директиву, чтобы назначить символическую константу имени регистра. В этом случае, символ становится синонимом регистра:

```
sym    .set    B1
MVK    10, sym
```

Символьные строки. Это строка кодовых знаков, заключенная в **двойные кавычки**. Двойные кавычки, которые являются частью символьной строки, представляются двумя последовательными двойными кавычками. Максимальная длина строки изменяется и определена для каждой директивы, которая требует символьную строку. Знаки представляются внутренне, как кодовые знаки ASCII с 8 битами. Примеры допустимых символьных строк:

“sample program”	определяет строку «sample program» с 14 знаками.
“PLAN “”C”””	определяет строку PLAN “C” с 8 знаками.

Символьные строки используются для следующих целей:

- Имена файлов, как в директиве .copy “имя файла”.
- Имена разделов, как в директиве .sect “имя раздела”.
- Директивы инициализации данных, как в .byte “символьная строка”.
- Операнды директив .string.
- Символы

Используются как метки, константы и символы замены. Имя символа - строка до 200 алфавитно-цифровых знаков (A-Z, a-z, 0-9, $, и _). Первый знак в символе не может быть числом, и символы не могут содержать внутренние пробелы. Символы, которые Вы определяете, **чувствительны к регистру**. Например, ассемблер различает ABC, Abc и abc, как три уникальных символа. Символ допустим только внутри ассемблерной программы.

Метки. Символы, используемые как метки, станут символическими адресами, которые связаны с ячейками памяти в программе. Метки, используемые локально, в пределах файла должны быть уникальны. Мнемонические коды операции и имена директив ассемблера без префикса (.) - допустимые имена меток.

Метки могут также использоваться, как операнды .global, .ref, .def, или .bss директив. Например:

```
        .global  label1
label2: MVK     label2, B3
        MVKH  label2, B3
        B   label1
        NOP    5
```

Локальные метки. Это специальные метки, чьи возможности и сила - временные. Локальная метка может быть определена двумя способами:

- $n, где n - десятичная цифра в диапазоне 0-9. Например, $ 4 и $ 1 являются допустимыми локальными метками..
- имя?, где имя - любое законное имя символа, как описано выше. Ассемблер заменяет вопросительный знак точкой, сопровождаемой уникальным числом. Когда исходный текст расширен, Вы не будете видеть уникальное число в файле листинга. Ваша метка появляется с вопросительным знаком, как это сделано в исходном определении. Вы не можете объявлять эту метку как глобальную.

Нормальные метки должны быть уникальны (они могут быть объявлены только однажды), и они могут использоваться как константы в поле операнда. Локальные метки, однако, могут быть отменены и определены снова. Локальные метки не могут быть определены директивами.

Символические константы. Символам могут быть присвоены постоянные значения. Используя константы, Вы можете сопоставлять имена с постоянными значениями. Директивы .set и .struct/.tag/.endstruct дают Вам возможность присвоить константам символические имена. Символические константы не могут быть переопределены.

Предопределенные символические константы. Ассемблер имеет несколько предопределенных символов, включая следующие типы:

- $, знак доллара, представляет текущее значение счетчика команд раздела (Segment Program Counter - SPC). $ - перемещаемый символ.
- Символы процессоров, включая TMS320C2700, TMS320C2800, TMS320C2800,_FPU32.
- Регистры управления ЦП, включая следующее:

ACC или AH, AL	Регистр аккумулятора и его части
DBGIER	Регистр разрешения прерываний при отладке
DP	Регистр указателя страницы памяти
IER	Регистр разрешения прерываний
IFR	Регистр флагов прерываний
P или PH, PL	Регистр произведения и его части
PC	Программный счетчик
RPC	Счетчик возвратов программы
ST0	Регистр установки прерываний
ST1	Статусный регистр 0
SP	Регистр указателя стека
TH	Старшая часть регистра сомножителя. Синоним T регистра
XARn или ARnH, ARn n = 0-7	Дополнительный регистр с номером n и его части
XT T, TL	Регистр сомножителя и его части

Символы замены. Символы могут быть назначены строковыми значениям (переменным). Это позволяет Вам заменять символьные строки, приравнивая их символическим именам. Символы, которые представляют строки знаков, называются символами замены. Когда ассемблер сталкивается с символом замены, его строковое значение заменяется именем символа. В отличие от символических констант, символы замены могут быть переопределены. Строка может быть назначена символу замены где-нибудь в пределах программы.

Выражения. Выражение - константа, символ, или ряд констант и символов, разделенные арифметическими операторами. 32-разрядные диапазоны допустимых значений выражения: от -2147 483 648 до 2147 483 647 для знаковых значений, от 0 до 4 294 967 295 для значений без знака. Три основных фактора влияют на порядок выполнения выражения:

- **Круглые скобки**. Выражения, включенные в круглые скобки, всегда рассчитываются сначала. 8 / (4 / 2) = 4, но 8 / 4 / 2 = 1. Вы не можете заменять круглые скобки на фигурные скобки ({}) или квадратные скобки ([]).
- **Группы по старшинству**. Операторы, перечисленные ниже, разделены на девять групп по старшинству. Когда круглые скобки не определяют порядок оценки выражения, первой выполняется самая высокая по старшинству операция. 8+4/2=10 (сначала вычислено 4/2).
- **Выполнение слева направо.** Когда круглые скобки и группы по старшинству не определяют порядок оценки выражения, выражения вычисляются слева направо, кроме группы 1, в которой они вычисляются справа налево. 8/4*2=4, но 8/(4*2)=1.

Список операторов, которые могут использоваться в выражениях, в соответствие с группами старшинства.

Группа	Оператор	Описание
1	+ - ~ !	Унарный плюс Унарный минус Дополнение до 1 Логическое НЕ
2	* / %	Умножение Деление Деление по модулю. Результат – остаток от деления
3	+ -	Сложение Вычитание
4	<< >>	Сдвиг влево Сдвиг вправо
5	< < = > > =	Меньше чем Меньше или равно Больше чем Больше или равно
6	= ! =	Равно Не равно
7	&	Поразрядное И
8	^	Поразрядное исключающее ИЛИ (XOR)
9	|	Поразрядное ИЛИ

Внимание: операторы группы 1 вычисляются справа налево. Все другие операторы вычисляются слева направо.

Ассемблер проверяет условия переполнения и антипереполнения, когда арифметические операции выполняются во время трансляции. Он дает предупреждение (Value truncated – значение усечено) всякий раз, когда происходит переполнение или антипереполнение. Ассемблер не проверяет переполнение или антипереполнение при умножении.

Четкие выражения. Некоторые директивы ассемблера требуют четких выражений в качестве операндов. Четкие выражения содержат только символы или разовые константы ассемблера, которые определены прежде, чем с ними сталкиваются в выражении. Значение четкого выражения должно быть абсолютным (без знака). Это - пример четкого выражения:

1000h+X

где X был предварительно определен как абсолютный символ.

Условные выражения. Ассемблер поддерживает условные операторы, которые могут использоваться в любом выражении. Они особенно полезны для условной трансляции. Условные операторы включают следующие:

Оператор	Описание
=	Равно
!=	Не равно
<	Меньше
<=	Меньше или равно
>	Больше
>=	Больше или равно

Условные выражения равны 1, если они истинны, и 0, если ложны и могут использоваться только на операндах эквивалентных типов. Например, абсолютная величина сравнивается с абсолютной величиной, но не с перемещаемым значением.

Законные выражения. За исключением перечисленных ниже случаев в выражениях нет ограничений на использование операторов, констант, внутренне или внешне определенных символов.

Когда выражение содержит более чем один перемещаемый символ или не может быть вычислено во время ассемблирования, ассемблер кодирует и помещает его в объектный файл и оно вычисляется линкером. Если финальное значение выражения требует больше места, чем отведено для него, то Вы получите сообщение линкера об ошибке.

Исключения для законных выражений. При использовании в регистрах режима относительной адресации выражение в квадратных скобках должно быть четким. Например, *+A4[15]

Встроенные математические функции.

Функция	Описание
$acos(x)	Арккосинус в диапазоне [0, π], x [-1, 1]
$asin(x)	Арксинус в диапазоне [-π/2, π/2], x [-1, 1]
$atanx)	Арктангенс в диапазоне [-π/2, π/2]
$atan2(x, y)	Арктангенс от y/x в диапазоне [-π, π]
$ceil(x)	Округление до ближайшего целого, меньшего, чем x с плавающей точкой
$cos(x)	Косинус от x
$cosh(x)	Косинус гиперболический от x
$cvf(n)	Превращает целое в формат с плавающей точкой
$cvi(x)	Превращает число с плавающей точкой в целое
$exp(x)	Экспонента от x
$fabs(x)	Абсолютное значение от x
$floor(x)	Округление до ближайшего целого, большего, чем x с плавающей точкой
$fmod(x, y)	Остаток от деления x/y, знак как для x
$int(x)	Возвращает 1, если x целое, иначе 0
$ldexp(x, n)	Умножение x на 2 в степени n
$log(x)	Натуральный логарифм
$log10(x)	Десятичный логарифм
$max(x, y, ...z)	Наибольшее из списка
$min(x, y, ...z)	Наименьшее из списка
$pow(x, y)	X в степени y
$round(x)	Округление до ближайшего целого
$sgn(x)	Знак числа. 1 для положительного, 0 для отрицательного
$sin(x)	Синус от x
$sinh(x)	Синус гиперболический от x
$sqrt(x)	Квадратный корень из x
$tan(x)	Тангенс от x
$tanh(x)	Тангенс гиперболический от x
$trunc(x)	Ближайшее целое по отношению к 0

Режимы работы ассемблера. Возможны 3 режима:

- -v27 для C27x.
- -v28 для C28x.
- -v28 –C2xlp_src_compatible - для C28x с поддержкой дополнительных инструкций C2xlp.

8.5. Команды ассемблера

Всего определено… команд, разделенных по категориям:

Категория	Кол-во
С регистрами общего назначения XAR0 – XAR7	14
С регистром указателя сегмента памяти данных DP	3
С регистром указателя стека SP	30
С регистрами AX (AH, AL)	38
16-разрядные с регистрами ACC	26
32-разрядные с регистрами ACC	44
64-разрядные с регистрами ACC	9
С регистрами P или XT	21
Перемножение 16x16	20
Перемножение 32x32	13
Прямой доступ к памяти	17
Ввод/вывод	3
Программная память	5
Ветвление, вызов, возврат	31
С регистром прерываний IER	9
С регистрами статуса ST0, ST1	18
Разного назначения	10
Итого	311

8.5.1. Операции с регистрами XAR0-XAR7

Мнемоника	Код операции	XARn				Пример
ADDB XARn, #7bit	1101	1nnn	0CCC	CCCC		XARn = XARn + 0:7bit;
ADRK #8bit	1111	1100	IIII	IIII		XAR(ARP) = XAR(ARP) + 0:8bit;
CMPR 0	0101	0100	0001	1101		Сравнение AR0 с

CMPR 1 CMPR 2 CMPR 3				1001 1000 1100		AR(ARP) на = > < !=
MOV AR6,loc16 MOV AR7,loc16	0101	1110	LLLL	LLLL		AR6/7 = [loc16]; AR6/7H = unchanged;
MOV loc16,ARn	0111	1nnn	LLLL	LLLL		[loc16] = ARn;
MOV XARn,PC	0011	1110	0101	1nnn		XARn = 0:PC;
MOVB AR6,#8bit MOVB AR7,#8bit	1101	0110 0111	CCCC	CCCC		AR6/7 = 0:8bit; AR6/7H = unchanged;

MOVB XAR0…5, #8bit
MOVB XAR6, #8bit
MOVB XAR7, #8bit
	1101
	1011
	1011
		0nnn
		1110
		0110
			CCCC	CCCC		XARn = 0:8bit;

MOVL loc32,XAR0
MOVL loc32,XAR1
MOVL loc32,XAR2
MOVL loc32,XAR3
MOVL loc32,XAR4
MOVL loc32,XAR5
MOVL loc32,XAR6
MOVL loc32,XAR7
	0011
	1011
	1010
	1010
	1010
	1010
	1100
	1100
		1010
		0010
		1010
		0010
		1000
		0000
		0010
		0011
			LLLL	LLLL		[loc32] = XARn;

MOVL XAR0,	1000	1110	LLLL	LLLL		XARn =

loc32 MOVL XAR1, loc32 MOVL XAR2, loc32 MOVL XAR3, loc32 MOVL XAR4, loc32 MOVL XAR5, loc32 MOVL XAR6, loc32 MOVL XAR7, loc32	1000 1000 1000 1000 1000 1100 1100	1011 0110 0010 1010 0011 0100 0101				[loc32];
MOVL XAR0, #22bit MOVL XAR1, #22bit MOVL XAR2, #22bit MOVL XAR3, #22bit MOVL XAR4, #22bit MOVL XAR5, #22bit MOVL XAR6, #22bit MOVL XAR7, #22bit	1000 1000 1000 1000 1000 1000 0111 0111	1110 1011 0110 0010 1010 0011 0100 0101	00CC	CCCC	CCCC CCCC CCCC CCCC	XARn = 0:22bit; В КОП добавлены 16 бит
MOVZ AR0…5n,loc16 MOVZ AR6,loc16 MOVZ AR7,loc16	0101 1000 1000	1nnn 1000 0000	LLLL	LLLL		ARn = [loc16]; ARnH = 0;
SBRK #8bit	1111	1011	CCCC	CCCC		XAR(ARP) = XAR(ARP) −

						0:8bit;
SUBB XARn, #7bit	1101	1nnn	1CCC	CCCC		XARn = XARn − 0:7bit;

8.5.2. Операции загрузки регистра DP

Регистр DP содержит адрес памяти данных.

Мнемоника	Код операции	XARn				Пример
MOV DP,#10bit	1111	10CC	CCCC	CCCC		DP(9:0) = 10bit; DP(15:10) = unchanged;
MOVW DP,#16bit	0111	0110	0001	1111	CCCC CCCC CCCC CCCC	DP(15:0) = 16bit;
MOVZ DP,#10bit	1011	10CC	CCCC	CCCC		DP(9:0) = 10bit; DP(15:10) = 0;

8.5.3. Операции с регистром SP

Мнемоника	Код операции	XARn			Пример
ADDB SP,#7bit	1111	1110	0CCC	CCCC	SP = SP + 0:7bit;
POP ACC	0000	0110	1011	1110	SP −= 2; ACC = [SP];
POP AR1:AR0	0111	0110	0000	0111	SP −= 2; AR0 = [SP]; AR1 = [SP+1]; AR1H:AR0H = без изменений;
POP AR3:AR2	0111	0110	0000	0101	SP −= 2; AR2= [SP]; AR3 = [SP+1]; AR3H:AR2H = без изменений;
POP AR5:AR4	0111	0110	0000	0110	SP −= 2; AR5 = [SP]; AR4= [SP+1];

					AR5H:AR4H = без изменений;
POP AR1H:AR0H	0000	0000	000	0011	SP −= 2; AR0H = [SP]; AR1H = [SP+1]; AR1:AR0 = без изменений;
POP DBGIER	0111	0110	0001	0010	SP −= 1; DBGIER = [SP];
POP DP:ST1	0111	0110	0000	0001	SP −= 2; ST1 = [SP]; DP = [SP+1];
POP DP	0111	0110	0000	0011	SP −= 1; DP = [SP];
POP IFR	0000	0000	0000	0010	SP −= 1; IFR = [SP];
POP loc16	0010	1010	LLLL	LLLL	SP -= 1; [loc16] = [SP];
POP P	0111	0110	0001	0001	SP −= 2; P = [SP];
POP RPC	0000	0000	0000	0111	SP −= 2; RPC = [SP];
POP ST0	0111	0110	0001	0011	SP −= 1; ST0 = [SP];
POP ST1	0111	0110	0000	0000	SP −= 1; ST1 = [SP];
POP T:ST0	0111	0110	0001	0101	SP −= 2; T = [SP]; ST0 = [SP+1]; TL = без изменений;
POP XT	0111	0110			SP −= 2; XT = [SP];
POP XAR0 POP XAR1 POP XAR2 POP XAR3 POP XAR4 POP XAR5	0011 1011 1010 1010 1010 1010	1010 0010 1010 0010 1000 0000	1011	1110	SP −= 2; XARn = [SP];

POP XAR6 POP XAR7	1100 1100	0010 0011			
PUSH ACC	0001	1110	1011	1101	[SP] = ACC; SP += 2;
PUSH AR0:AR1 PUSH AR3:AR2 PUSH AR5:AR4	0111	0110	0000	1101	[SP] = AR0; [SP+1] = AR1; SP += 2;
PUSH AR3:AR2	0111	0110	0000	1111	[SP] = AR2; [SP+1] = AR3; SP += 2;
PUSH AR5:AR4	0111	0110	0000	1100	[SP] = AR4; [SP+1] = AR5; SP += 2;
PUSH AR1H:AR0H	0000	0000	0000	0101	[SP] = AR0H; [SP+1] = AR1H; SP += 2;
PUSH DBGIER	0111	0110	0000	1110	[SP] = DBGIER; SP += 1;
PUSH DP:ST1	0111	0110	0000	0000	[SP] = DP; SP += 1;
PUSH DP	0111	0110	0000	1011	[SP] = ST1; [SP+1] = DP; SP += 2;
PUSH IFR	0111	0110	0000	1010	[SP] = IFR; SP += 1;
PUSH loc16	0010	0010	LLLL	LLLL	[SP] = [loc16]; SP += 1;
PUSH P	0111	0110	0001	1101	[SP] = P; SP += 2;
PUSH RPC	0000	0000	0000	0100	[SP] = RPC; SP += 2;
PUSH ST0	0111	0110	0001	1000	[SP] = ST0; SP += 1;
PUSH ST1	0111	0110	0000	1000	[SP] = ST1; SP += 1;

8.5.4. Операции с регистрами AX (AH, AL)

AX – это части ACC. AH – старшая, AL – младшая. Для них доступны команды с данными не юлее 16 бит.

Мнемоника	Код операции	XARn			Пример
ADD AX,loc16	1001	010A	LLLL	LLLL	
ADD loc16,AX					
ADDB AX,#8bit					
AND AX,loc16,#16bit					
AND AX,loc16					
AND loc16,AX					
ANDB AX,#8bit					
ASR AX,1..16					
ASR AX,T					
CMP AX,loc16					
CMPB AX,#8bit					
FLIP AX					
LSL AX,1..16					
LSL AX,T					
LSR AX,1..16					
LSR AX,T					
MAX AX,loc16					
MIN AX,loc16					
MOV AX,loc16					
MOV loc16,AX					
MOV loc16,AX,COND					
MOVB AX,#8bit					

Мнемоника	Пример
ADD Ист1, Ист2[Сдвиг]	Добавить Ист1 =Ист1 + Ист2
AND Ист1, Ист2[Сдвиг]	Операция И Ист1 =Ист1 & Ист2
ASR Ист1[Сдвиг]	Арифметический сдвиг вправо Ист1 =Ист1 со сдвигом
ADD Ист1, Ист2[Сдвиг]	Добавить

	Ист1 =Ист1 + Ист2

8.5.5. Операции с регистрами ACC

Формат команд:

Команда Ист1, Ист2{Сдвиг)

С содержимым Ист1 и Ист2 выполняется команда. Предварительно содержимое Ист2 сдвигается влево на число бит Сдвиг (необязательно)

В качестве Ист1 или Ист2 могут быть:

- ACC – аккумулятор АЛУ.
- loc16 – 16-разрядная ячейка памяти.
- loc32 – 32 разрядная ячейка памяти.

В качестве Ист2 дополнительно могут быть:

- #8bit – 8-разрядное значение.
- #16bit – 16-разрядное значение.

Примеры команд

Мнемоника	Действие
ADD ACC,loc32 {<< 0..16}	Сложение. К ACC добавить содержимое loc32 и перенос. ACC = ACC + [loc32] + C;

ADD ACC,loc16 {<< 0..16}	Сложение.
	К ACC добавить содержимое loc16.
	
	if(SXM = 1) //расшир. знака разрешено
	ACC = ACC + S:[loc16]<<сдвиг
	else // расшир.знака запрешено
	ACC = ACC + 0:[loc16]<<сдвиг;

ADDB ACC,#8bit	Сложение. К ACC добавить 8-битную константу.

	ACC = ACC + 0:8bit;

SUB ACC,loc16 {<< 0..16}	Вычитание.
	Из ACC вычесть содержимое loc16.
	
	if(SXM = 1) //расшир. знака разрешено
	ACC = ACC - S:[loc16]<<сдвиг
	else // расшир.знака запрешено
	ACC = ACC - 0:[loc16]<<сдвиг;

SUBB ACC,#8bit	Вычитание. Из ACC вычесть 8-битную константу. ACC = ACC - 0:8bit;
AND ACC,loc16	ACC = ACC AND 0:[loc16];
AND ACC,#16bit {<< 0..16}	ACC = ACC AND (0:16bit << сдвиг);
OR ACC,loc16	
OR ACC,#16bit {<< 0..16}	
XOR ACC,loc16	
XOR ACC,#16bit {<< 0..16}	

8.5.6. Операции с регистрами P или XT

Инструкция	Мнемоника	Действие
ADDUL	ADD Ист1, Ист2[Сдвиг]	Добавить Ист1 =Ист1 + Ист2
MAXCL	AND Ист1, Ист2[Сдвиг]	Операция И Ист1 =Ист1 & Ист2
MINCL	AND Ист1, Ист2[Сдвиг]	Операция И Ист1 =Ист1 & Ист2
MOV	ASR Ист1[Сдвиг]	Арифметический сдвиг вправо Ист1 =Ист1 со сдвигом
MOVA	ASR Ист1[Сдвиг]	Арифметический сдвиг вправо Ист1 =Ист1 со сдвигом
MOVAD	ASR Ист1[Сдвиг]	Арифметический сдвиг вправо Ист1 =Ист1 со сдвигом
MOVDL	ASR Ист1[Сдвиг]	Арифметический сдвиг вправо Ист1 =Ист1 со сдвигом

MOVH	ASR Ист1[Сдвиг]	Арифметический сдвиг вправо Ист1 =Ист1 со сдвигом
MOVL	ASR Ист1[Сдвиг]	Арифметический сдвиг вправо Ист1 =Ист1 со сдвигом
MOVP	ASR Ист1[Сдвиг]	Арифметический сдвиг вправо Ист1 =Ист1 со сдвигом
MOVS	ASR Ист1[Сдвиг]	Арифметический сдвиг вправо Ист1 =Ист1 со сдвигом
MOVX	ASR Ист1[Сдвиг]	Арифметический сдвиг вправо Ист1 =Ист1 со сдвигом
SUBUL	ASR Ист1[Сдвиг]	Арифметический сдвиг вправо Ист1 =Ист1 со сдвигом
DMAC	ASR Ист1[Сдвиг]	Арифметический сдвиг вправо Ист1 =Ист1 со сдвигом
MAC	ASR Ист1[Сдвиг]	Арифметический сдвиг вправо Ист1 =Ист1 со сдвигом
MPY	ASR Ист1[Сдвиг]	Арифметический сдвиг вправо Ист1 =Ист1 со сдвигом
MPYA	ASR Ист1[Сдвиг]	Арифметический сдвиг вправо Ист1 =Ист1 со сдвигом
MPYB	ASR Ист1[Сдвиг]	Арифметический сдвиг вправо Ист1 =Ист1 со сдвигом
MPYS	ASR Ист1[Сдвиг]	Арифметический сдвиг вправо Ист1 =Ист1 со сдвигом
MPYU	ASR Ист1[Сдвиг]	Арифметический сдвиг вправо Ист1 =Ист1 со сдвигом
MPYXU	ASR Ист1[Сдвиг]	Арифметический сдвиг вправо Ист1 =Ист1 со сдвигом
SQRA	ASR Ист1[Сдвиг]	Арифметический сдвиг вправо Ист1 =Ист1 со сдвигом
SQRS	ASR Ист1[Сдвиг]	Арифметический сдвиг вправо Ист1 =Ист1 со сдвигом
XMAC	ASR Ист1[Сдвиг]	Арифметический сдвиг вправо Ист1 =Ист1 со сдвигом
XMACD	ASR Ист1[Сдвиг]	Арифметический сдвиг вправо Ист1 =Ист1 со сдвигом
IMACL	ASR Ист1[Сдвиг]	Арифметический сдвиг вправо Ист1 =Ист1 со сдвигом

IMPYAL	ASR Ист1[Сдвиг]	Арифметический сдвиг вправо Ист1 =Ист1 со сдвигом
IMPYL	ASR Ист1[Сдвиг]	Арифметический сдвиг вправо Ист1 =Ист1 со сдвигом
IMPYSL	ASR Ист1[Сдвиг]	Арифметический сдвиг вправо Ист1 =Ист1 со сдвигом
IMPYXUL	ASR Ист1[Сдвиг]	Арифметический сдвиг вправо Ист1 =Ист1 со сдвигом
QMACL	ASR Ист1[Сдвиг]	Арифметический сдвиг вправо Ист1 =Ист1 со сдвигом
QMPYAL	ASR Ист1[Сдвиг]	Арифметический сдвиг вправо Ист1 =Ист1 со сдвигом
QMPYSL	ASR Ист1[Сдвиг]	Арифметический сдвиг вправо Ист1 =Ист1 со сдвигом
QMPYXUL	ASR Ист1[Сдвиг]	Арифметический сдвиг вправо Ист1 =Ист1 со сдвигом

8.5.7. Операции прямого доступа к памяти

Инструкция	Мнемоника	Действие
ADD	ADD Ист1, Ист2[Сдвиг]	Добавить Ист1 =Ист1 + Ист2
AND	AND Ист1, Ист2[Сдвиг]	Операция И Ист1 =Ист1 & Ист2
CMP	AND Ист1, Ист2[Сдвиг]	Операция И Ист1 =Ист1 & Ист2
DEC	AND Ист1, Ист2[Сдвиг]	Операция И Ист1 =Ист1 & Ист2
DMOV	AND Ист1, Ист2[Сдвиг]	Операция И Ист1 =Ист1 & Ист2
INC	AND Ист1, Ист2[Сдвиг]	Операция И Ист1 =Ист1 & Ист2
MOV	AND Ист1, Ист2[Сдвиг]	Операция И Ист1 =Ист1 & Ист2
MOVB	AND Ист1, Ист2[Сдвиг]	Операция И Ист1 =Ист1 & Ист2
OR	AND Ист1, Ист2[Сдвиг]	Операция И Ист1 =Ист1 & Ист2
TBIT	AND Ист1, Ист2[Сдвиг]	Операция И

		Ист1 =Ист1 & Ист2
TCLR	AND Ист1, Ист2[Сдвиг]	Операция И Ист1 =Ист1 & Ист2
TSET	AND Ист1, Ист2[Сдвиг]	Операция И Ист1 =Ист1 & Ист2
XOR	AND Ист1, Ист2[Сдвиг]	Операция И Ист1 =Ист1 & Ист2

8.5.8. Операции ввода вывода

Инструкция	Мнемоника	Действие
IN	ADD Ист1, Ист2[Сдвиг]	Добавить Ист1 =Ист1 + Ист2
OUT	AND Ист1, Ист2[Сдвиг]	Операция И Ист1 =Ист1 & Ист2
UOUT	AND Ист1, Ист2[Сдвиг]	Операция И Ист1 =Ист1 & Ист2

8.5.9. Операции с памятью программ

Инструкция	Мнемоника	Действие
PREAD	ADD Ист1, Ист2[Сдвиг]	Добавить Ист1 =Ист1 + Ист2
PWRITE	AND Ист1, Ист2[Сдвиг]	Операция И Ист1 =Ист1 & Ист2
XWRITE	AND Ист1, Ист2[Сдвиг]	Операция И Ист1 =Ист1 & Ист2
XWRITE	AND Ист1, Ист2[Сдвиг]	Операция И Ист1 =Ист1 & Ист2
XPWRITE	AND Ист1, Ист2[Сдвиг]	Операция И Ист1 =Ист1 & Ист2

8.5.10. Операции ветвления, вызова, возврата

Инструкция	Мнемоника	Действие
B	ADD Ист1, Ист2[Сдвиг]	Добавить Ист1 =Ист1 + Ист2
BANZ	AND Ист1, Ист2[Сдвиг]	Операция И Ист1 =Ист1 & Ист2
BAR	AND Ист1, Ист2[Сдвиг]	Операция И

		Ист1 =Ист1 & Ист2
BF	AND Ист1, Ист2[Сдвиг]	Операция И Ист1 =Ист1 & Ист2
FFC	AND Ист1, Ист2[Сдвиг]	Операция И Ист1 =Ист1 & Ист2
IRET	AND Ист1, Ист2[Сдвиг]	Операция И Ист1 =Ист1 & Ист2
LB	AND Ист1, Ист2[Сдвиг]	Операция И Ист1 =Ист1 & Ист2
LC	AND Ист1, Ист2[Сдвиг]	Операция И Ист1 =Ист1 & Ист2
LCR	AND Ист1, Ист2[Сдвиг]	Операция И Ист1 =Ист1 & Ист2
LOOPZ	AND Ист1, Ист2[Сдвиг]	Операция И Ист1 =Ист1 & Ист2
LOOPNZ	AND Ист1, Ист2[Сдвиг]	Операция И Ист1 =Ист1 & Ист2
LRET	AND Ист1, Ист2[Сдвиг]	Операция И Ист1 =Ист1 & Ист2
LRETE	AND Ист1, Ист2[Сдвиг]	Операция И Ист1 =Ист1 & Ист2
LRETR	AND Ист1, Ист2[Сдвиг]	Операция И Ист1 =Ист1 & Ист2
RPT	AND Ист1, Ист2[Сдвиг]	Операция И Ист1 =Ист1 & Ист2
SB	AND Ист1, Ист2[Сдвиг]	Операция И Ист1 =Ист1 & Ист2
SF	AND Ист1, Ист2[Сдвиг]	Операция И Ист1 =Ист1 & Ист2
SBF	AND Ист1, Ист2[Сдвиг]	Операция И Ист1 =Ист1 & Ист2
XB	AND Ист1, Ист2[Сдвиг]	Операция И Ист1 =Ист1 & Ист2
XBANZ	AND Ист1, Ист2[Сдвиг]	Операция И Ист1 =Ист1 & Ист2
XCALL	AND Ист1, Ист2[Сдвиг]	Операция И Ист1 =Ист1 & Ист2

8.5.11. Математические

Синтаксис мнемоники команды:

Имя Ист1, Ист2[Сдвиг]

Инструкция может использовать 1 или 2 операнда Ист1, Ист2. Ист1 – это источник первого операнда и приемник результата. Ист2 – это источник второго операнда. Для Ист2 может использоваться предварительная операция сдвига на задаваемое число бит.

Операндами могут быть:

- Регистр аккумулятора.ACC и его части (старшая AX, младшая AL).
- Регистр произведения блока умножителя P и его части (старшая PH, младшая PL).
- Константы 16, 8 или 7 бит.
- Дополнительные регистры XAR общего назначения.
- Содержимое памяти с заданным размещением loc16, loc32.

Пример.

ADD XAR0, XAR3 << 3 ; Сложение XAR0 со сдвинутым влево на 3 бита XAR3.

Основные команды:

Мнемоника	Действие
ABORTI	Отменить прерывание
ABS ACC	Абсолютное значение ACC ACC = Abs(ACC)
ADD Ист1, Ист2[Сдвиг]	Добавить Ист1 = Ист1 + Ист2
ADD XARn, #7bit	Добавить константу из 7 бит XARn = XARn + #7bit
ADD Ист1, Ист2[Сдвиг]	Добавить Ист1 = Ист1 + Ист2
AND Ист1, Ист2[Сдвиг]	Операция И Ист1 =Ист1 & Ист2
ASR Ист1[Сдвиг]	Арифметический сдвиг вправо Ист1 =Ист1 со сдвигом
B Смещение, Условие	Условный переход Переход на (Смещение) при выполнении (Условия)

BF Смещение, Условие	Быстрый условный переход Переход на (Смещение) при выполнении (Условия)
BANZ Смещение, Ист2	

8.5.12. Ветвления

Синтаксис мнемоники команды:

Имя Смещение, Условие

Если условие выполняется, то содержимое счетчика команд меняется на величину (Смещения). Мнемоника проверяемых условий:

Условие	Объяснение	Проверяемые флаги
NEQ	не равно	Z=0
EQ	равно	Z=1
GT	больше	Z=0 и N=0
GEQ	больше или равно (не меньше)	N=0
LT	меньше	N=1
LEQ	меньше или равно (не больше)	Z=1 или N=1
HI	выше	C=1 и Z=0
HIS, C	выше или тоже самое	C=1
LO, NC	ниже, переноса нет	C=0
LOS	ниже или тоже самое	C=0 или Z=1
NOV	нет переполнения.	V=0
OV	переполнение	V=1
NTC	бит теста не установлен	TC=0
TC	бит теста установлен.	TC=1
NBIO	входной сигнал BIO нулевой	BIO=0
UNC	безусловный	

8.5.13. Основные инструкции для работы с регистрами

Инструкция.	Мнемоника	Действие
ADDB	ADDB XARn, #7bit #7bit - 7-битовая константа без знака	XARn = XARn + #7bit
ADRK	ADRK XARn, #8bit #7bit - 8-битовая константа без знака	XARn = XARn + #8bit
ADDK	addk (.unit) cst, dst	cst+dst => dst

	.unit = .S1, .S2	
SUB SUBU	sub (.unit) src1, src2, dst .unit = .L1, .L2, S1, S2	src1=src2 => dst
ABS	abs (.unit) src, dst .unit = .L1, .L2	abs(src) => dst
B	b (.unit) label .unit = S1, S2	
CMPEQ	cmpeq (.unit) src1, src2, dst .unit = .L1, .L2	1 => dst при src1=src2
CMPGT CMPGTU	cmpgt (.unit) src1, src2, dst .unit = .L1, .L2	1 => dst при src1>src2
CMPLT CMPLTU	cmplt (.unit) src1, src2, dst .unit = .L1, .L2	1 => dst при src1<src2
MPY MPYU	mpy (.unit) src1, src2, dst .unit = .M1, .M2	src1*src2 => dst
MV	mv (.unit) src, dst .unit = .L1, .L2, S1, .S2, .D1, .D2	src => dst
MVK	mvk (.unit) cst, dst .unit = .S1, .S2	cst => dst
NEG	neg (.unit) src, dst .unit = .L1, .L2, .S1, .S2	=src => dst
NOP	nop	
STB STH STW	stb (.unit) src, *+baseR[offserR] .unit = .D1, .D2	src => baseR[offsetR]
LDB LDH LDW	ldb (.unit) *+baseR[offserR], dst .unit = .D1, .D2	baseR[offsetR] => dst
AND	and (.unit) src1, src2, dst .unit = .L1, .L2, S1, S2	src1 AND src2 => dst
OR	or (.unit) src1, src2, dst .unit = .L1, .L2, .S1, .S2	src1 OR src2 => dst
XOR	xor (.unit) src1, src2, dst .unit = .L1, .L2, S1, S2	src1 XOR src2 => dst
NOT	not (.unit) srt, dst .unit = .L1, .L2, .S1, .S2	
SHL	shl (.unit) src2, src1, dst .unit = .S1, .S2	(src2 на src1) => dst

SHR	shr (.unit) src2, src1, dst .unit = .S1, .S2	(src2 на src1) => dst

8.5.14. Основные команды для работы с вещественными числами

Инструкция	Мнемоника	Действие
ABSSP ABSDP	abssp (.unit) src, dst .unit = .S1, .S2	absdp(src) => dst
ADDSP ADDDP	addsp (.unit) src1, src2, dst .unit = .L1, .L2	src1+src2 => dst
SUBSP SUBDP	subsp (.unit) src1, src2, dst .unit = .L1, .L2	src1+src2 => dst
CMPEQSP CMPEQDP	cmpeqsp (.unit) src1, src2, dst .unit = .S1, .S2	1 => dst при src1=src2
CMPGTSP CMPGTDP	cmpgtsp(.unit) src1, src2, dst .unit = .S1, .S2	1 => dst при src1>src2
CMPLTSP CMPLTDP	cmpltsp (.unit) src1, src2, dst .unit = .S1, .S2	1 => dst при src1<src2
MPYSP MPYDP	mpysp (.unit) src1, src2, dst .unit = .M1, .M2	src1*src2 => dst
SPINT	spint (.unit) src, dst .unit = .L1, .L2	src1 => dst
INTSP	intsp (.unit) src, dst .unit = .L1, .L2	src1 => dst
DPINT	dpint (.unit) src, dst .unit = .L1, .L2	src1 => dst
INTDP	intdp (.unit) src, dst .unit = .L1, .L2	src1 => dst
DPSP	dpsp (.unit) src, dst .unit = .L1, .L2	src1 => dst
SPDP	spdp (.unit) src, dst .unit = .L1, .L2	src1 => dst

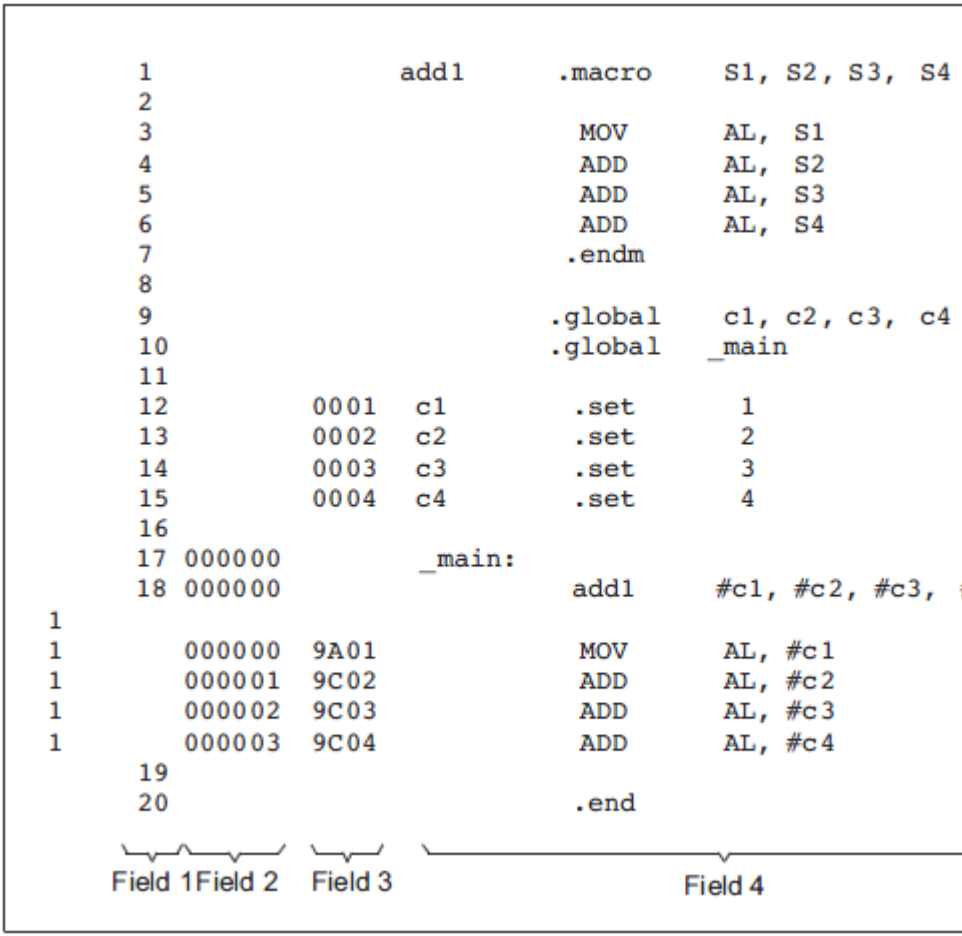
8.6. Листинги программ

Листинг программы показывает исходные инструкции и объектный код, который они производят. Чтобы получить файл листинга, вызовите ассемблер с опцией -L. Листинг печатается постранично. Незаполненную строку и строку заголовка,

имеют наверху каждая страница распечатки. Любой заголовок, определенный .title директивой, печатается в строке заголовка. Номер страницы печатается справа от заголовка. Если Вы не используете .title директиву, печатается имя исходного файла. Ассемблер вставляет незаполненную строку ниже строки заголовка.

Каждая строка в исходном файле создает, по крайней мере, одну строку в файле листинга. Она содержит в порядке слева направо номер исходной инструкции, значение SPC, объектный код, и исходную инструкцию..

Пример показывает листинг ассемблера.



8.7. Формат объектного файла

Ассемблер и компоновщик создают объектные файлы, которые могут выполняться устройством. Формат для этих объектных файлов назван общим объектным файловым форматом (COFF – ООФФ).

ООФФ упрощает модульное программирование при написании программы на языке ассемблера, поскольку он позволяет представлять программу и данные в

виде блоков. Эти блоки известны как разделы. Как ассемблер, так и компоновщик обеспечены директивами, которые позволяют создавать и манипулировать разделами.

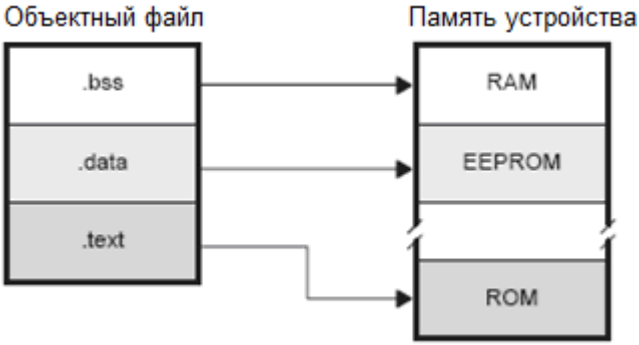
Разделы. Наименьший модуль объектного файла называется разделом. Раздел - блок программы или данных, который занимает непрерывное пространство на карте памяти с другими разделами. Каждый раздел объектного файла является самостоятельным и отличным от других. По умолчанию ООФФ объектных файлов всегда содержит три раздела:

- Раздел текста - .text обычно содержит выполняемый код.
- Раздел данных - .data обычно содержит инициализированные данные.
- Раздел .bss обычно резервируется для неинициализированных переменных.

Кроме того, ассемблер и компоновщик позволяют Вам создавать, называть и связывать поименованные разделы, которые используются подобно разделам .data, .text и .bss. Есть два основных типа разделов:

- **Инициализированные разделы** содержат данные или код. Разделы .text и .data проинициализированы. Именуемые разделы, создаваемые с помощью директивы ассемблера .sect, также проинициализированы;
- **Неинициализированные разделы** резервируют пространство на карте памяти для неинициализированных данных. Раздел .bss неинициализированный. Именуемые разделы, создаваемые с помощью директивы ассемблера .usect, также неинициализированные.

Различные директивы ассемблера позволяют связывать различные части программного кода и данных с соответствующими разделами. Ассемблер формирует эти разделы в течение процесса ассемблирования, создавая объектный файл, организованный так.



В системе имеется целевая платформа (Устройство), это плата с конкретным процессором. Там имеется целевая память, содержащая разделы: ПЗУ (ROM), перепрограммируемое ПЗУ (EEPROM), ОЗУ (RAM). Этим разделам назначается область адресов в карте распределения целевой памяти. Компоновщик должен перемещать в целевую память разделы объектного кода. Поскольку большинство систем содержат различные типы памяти, то применение разделов может помочь использовать целевую память более эффективно. Все разделы независимо переместимые, любой раздел можно разместить в любом блоке целевой памяти. Если у вас нет целевой платформы, то отладчик имитирует ее.

Ассемблер идентифицирует части программы на языке ассемблера, которые принадлежат данному разделу. Ассемблер имеет 5 директив поддержки этой функции:

- .bss
- .usect
- .text
- .data
- .sect

Директивы .bss и .usect создают неинициализированные разделы; директивы .text, .data, и .sect создают инициализированные разделы.

Вы можете создавать подразделы любого раздела, чтобы дать Вам более жесткое управление картой памяти. Подразделы создаются, используя .sect и .usect директивы. Подразделы идентифицируются основным именем раздела и именем подраздела, отделенным двоеточием.

Неинициализированные разделы резервируют пространство в памяти C28x; они обычно распределены в ОЗУ. Эти разделы не имеют никакого фактического содержания в объектном файле, они просто резервируют память. Программа может использовать это пространство во время выполнения для создания и сохранения переменных. Неинициализированные области данных формируются, используя директивы ассемблера .bss и .usect:

- Директива .bss резервирует пространство в .bss разделе. Директива .bss резервирует данное число байтов в .bss разделе. Вы должны определить размер, нет никакого значения по умолчанию.
- Директива .usect резервирует пространство в определенном неинициализированном названном разделе. Директива .usect резервирует данное число байтов в разделе с указанным именем. Вы должны определить размер, нет никакого значения по умолчанию.

Каждый раз, когда Вы вызываете .bss или .usect директиву, ассемблер резервирует дополнительное пространство в .bss или названном разделе. Синтаксис директивы .bss:

.bss символ, размер в словах [, флаг блокировки[, флаг выравнивания [, тип]]

Синтаксис директивы .usect:

символ .usect ”имя раздела”, размер в словах [, флаг блокировки[, флаг выравнивания]

- Символ указывает на первый байт, зарезервированный этим обращением директивы .bss или .usect. Символ соответствует имени переменной, для которой Вы резервируете пространство. На него может ссылаться любой другой раздел, а также он может быть объявлен, как глобальный символ (директивой ассемблера .global).
- Размер в словах – абсолютное выражение.
- Флаг блокировки.
- Флаг выравнивание – необязательный параметр. Он определяет минимальное выравнивание в байтах, требуемое распределяемым пространством. Значение по умолчанию – 1 байт. Значение должно быть степенью числа 2.
- Необязательный тип.

Директивы инициализированных разделов (.text, .data, и .sect) указывают ассемблеру прекратить трансляцию в текущий раздел и начать транслировать в обозначенный раздел. Директивы .bss и .usect, однако, не заканчивают данный раздел и не начинают новый, они просто выходят от текущего раздела временно. Директивы .bss и .usect могут появляться где-нибудь в инициализированном разделе, не воздействуя на его содержание.

Ассемблер обрабатывает неинициализированные подразделы (созданные .usect директивой) тем же самым способом, как неинициализированные разделы.

Инициализированные разделы содержат выполняемый код или инициализированные данные. Содержание этих разделов записывается в объектный файл и помещается в память C28x, когда программа загружается. Каждый инициализированный раздел – независимо перемещаемый и может ссылаться на символы, которые определены в других разделах. Компоновщик автоматически решает эти ссылки к другим разделам. Три директивы предписывают ассемблеру размещать код или данные в раздел. Синтаксис этих директив:

```
.text
.data
.sect  «имя раздела»
```

Когда ассемблер сталкивается с одной из этих директив, он останавливает трансляцию в текущий раздел (действует как команда конца текущего раздела). Затем он транслирует последующий код в обозначенный раздел, пока не обнаружит другую .text, .data, или .sect директиву.

Разделы формируются через итеративный процесс. Например, когда ассемблер первый раз сталкивается с директивой .data, раздел .data - пуст. Инструкции, следующие за первой директивой .data, собираются в .data раздел (пока ассемблер не сталкивается с .text или .sect директивой). Если ассемблер сталкивается с последующими .data директивами, то он прибавляет инструкции после этих директив .data к инструкциям, уже находящимся в разделе .data. Это создает единый раздел .data, который может быть распределен непрерывно в памяти.

Инициализированные подразделы создаются .sect директивой. Ассемблер обрабатывает инициализированные подразделы тем же самым способом, как инициализированные разделы.

Названные разделы - разделы, которые Вы создаете. Вы можете использовать их подобно заданным по умолчанию .text, .data, и .bss разделам, но они транслируются отдельно. Например, повторное использование .text директивы создает единый .text раздел в объектном файле. Во время компоновки этот .text раздел распределяется в памяти как одиночный модуль. Предположим, что имеется часть выполняемого кода (возможно подпрограмма инициализации), которую Вы не хотите размещать вместе с .text. Если Вы размещаете этот сегмент кода в названном разделе, он транслируется отдельно от .text, и Вы можете распределять его в памяти отдельно. Вы можете также транслировать инициализированные данные, который отличны от .data раздела, и Вы можете резервировать пространство для неинициализированных переменных, которые отличны от .bss раздела.

8.8. Директивы ассемблера

Директивы ассемблера поставляют данные программе и управляют процессом трансляции. Директивы ассемблера дают возможность Вам делать следующее:

- Транслировать код и данные в указанные разделы.
- Резервировать пространство в памяти для неинициализированных переменных.
- Управлять видом листинга.
- Инициализировать память.
- Транслировать условные блоки.
- Определять глобальные переменные.
- Определять библиотеки, из которых ассемблер может получить макрокоманды.
- Исследовать информацию о символьной отладке.

Внимание: Метки и комментарии не показаны в синтаксисе.

Любая исходная инструкция, которая содержит директиву, может также содержать метку и комментарий. Метки начинаются в первом столбце (они - единственные элементы, кроме комментариев, которые могут появляться в первом столбце), а комментарии должны начинаться с точки с запятой или звездочки, если комментарий - единственный элемент в строке. Чтобы улучшать разборчивость, метки и комментарии не показываются, как часть синтаксиса директив. В описании части, заключенные в квадратные скобки, могут пропускаться, ассемблер будет их задавать по умолчанию.

8.8.1. Разделы

Мнемоника и синтаксис	Описание
.bss символ, размер в словах [, флаг блокировки [, флаг выравнивания [, тип]]	Резервирует пространство в разделе .bss (неинициализированные данные)
.data	Транслирует в раздел .data (инициализированные данные).
.sect ”имя раздела”	Транслирует в названный (инициализированный) раздел
.text	Транслирует в раздел .text (выполняемый код)
символ .usect ”имя раздела”, размер в словах [, флаг блокировки[, флаг выравнивания]	Резервирует пространство в названном разделе (неинициализированном)

8.8.2. Константы

Мнемоника и синтаксис	Описание
.byte значение1[,..., значениеN]	Заносит в 16-разрядные слова байты из списка. В слово 1 байт в младшую часть.
.char значение1[,..., значениеN]	Заносит в 16-разрядные слова символы из списка. В слово 1 символ в младшую часть.
.string {выражение ”строка”}	Текстовые строки. Заносятся символы строки в младшие байты последовательных слов. То же самое делает .char со списком символов.
.pstring {выраж.1 ”строка1”} [,…,{выраж.N ”строкаN”}]	Аналог .string. Заносятся символы строки в оба байта последовательных слов.
.field значение[, размер]	Инициализирует подполя в константе размером 16 бит. Для каждого подполя задаются значение и размер. При последовательном применении константа заполняется справа налево.

	Применяется для упаковки нескольких значений в одном месте.
.int значение1[,...,значениеN]	Заносит в 16-разрядные слова 16-разрядные целые числа из списка. Есть выравнивание по границам слов.
.longзначение1 [,...,значениеN]	Заносит в 16-разрядные слова 32-разрядные целые числа из списка. На одно значение 2 слова. Есть выравнивание по границам слов.
.xlongзначение1 [,...,значениеN]	То же, что .long. Нет выравнивания по границам слов.
.word значение1 [,...,значениеN]	Заносит в 16-разрядные слова 16-разрядные числа из списка. Есть выравнивание по границам слов.
.float значение1[,...,значениеN]	Заносит в 16-разрядные слова 32-битные константы с ПТ, IEEE с однократной точностью из списка. На одно значение 2 слова. Есть выравнивание по границам слов.
.xfloat значение1 [,...,значениеN]	То же, что .flooat. Нет выравнивания по границам слов.

8.8.3. Выравнивания

Мнемоника и синтаксис	Описание
.align [размер в словах]	Выравнивает SPC на границе, указанной размером в байтах, который должен быть степенью 2; по умолчанию – до границы
.bes [размер в битах]	Резервирует биты в текущем разделе. Метка указывает на конец резервируемого пространства
.space [размер в битах]	Резервирует биты в текущем разделе. Метка указывает на начало резервируемого пространства

8.8.4. Листинг

Мнемоника и синтаксис	Описание
.drlist	Допускает распечатку всех строк директив (по умолчанию).

.drnolist	Подавляет распечатку определенных строк директив.
.fclist	Позволяет распечатку ложного условного блока (по умолчанию).
.fcnolist	Подавляет распечатку ложного условного блока кода.
.length [длина страницы]	Устанавливает длину страницы листинга программы
.list	Повторный запуск распечатки программы
.mlist	Позволяет распечатку макрокоманд и блоков циклов(по умолчанию)
.mnolist	Подавляет распечатку макрокоманд и блоков циклов
.nolist	Останавливает распечатку программы
.option опция1 [, опция2,...]	Выбирает опции листинга; доступны опции - A,B,D,H,L,M,N,O,R,T,W и X
.page	Пропускает страницу в распечатке программы
.sslist	Позволяет расширенный листинг символов замены
.ssnolist (по умолчанию)	Подавляет расширенный листинг символов замены
.tab размер	Устанавливает размер знаков табуляции (в символах)
.title “строка”	Печатает заголовок в начале страницы листинга
.width [ширина страницы]	Устанавливает ширину страницы распечатки программы

8.8.5. Файлы

Мнемоника и синтаксис	Описание
.copy [“]имя файла[“]	Включает исходные инструкции из другого файла
.def символ1 [,...,символN]	Идентифицирует один или более символов, которые определены в текущем модуле и могут использоваться в других модулях
.global символ1 [,...,символN]	Идентифицирует один или более глобальных символов
.include [“]имя файла[“]	Включает исходные инструкции из другого файла
.mlib [“]имя файла[“]	Определяет библиотеку макрокоманд
.ref символ1 [,...,символN]	Идентифицирует один или более символов, используемых в текущем модуле, которые определены в другом модуле

8.8.6. Условная трансляция

Мнемоника и синтаксис	Описание
.break [четкое выражение]	Заканчивает трансляцию .loop, если четкое выражение - истина. При использовании конструкции .loop, конструкция .break - необязательна
.else	Транслирует блок кода, если (.if четкое выражение) является ложным. При использовании конструкции .if, конструкция .else необязательна
.elseif четкое выражение	Транслирует блок, если .if четкое выражение является ложным, а условие .elseif - истинно. При использовании конструкции .if , конструкция .elseif - необязательна
.endif	Заканчивает блок кода .if
.endloop	Заканчивает блок кода .loop
.if четкое выражения	Транслирует блок, если четкое выражение является истинным
.loop [четкое выражение]	Начинает повторяемую трансляцию кодового блока; счетчик цикла определен четким выражением

8.8.7. Структуры

Мнемоника и синтаксис	Описание
.struct	Начало структуры. Это коллекция однотипных данных. Для данных, используемых совместно в исходниках Ассемблера и языка C++.
.cstruct	Начало структуры с выравниванием и размещением. Для данных, используемых совместно в исходниках Ассемблера и языка C++.
.endstruct	Конец структуры.
.union	Начало юниона. Это коллекция разнотипных данных. Для данных, используемых совместно в исходниках Ассемблера и языка C++.
.endunion	Конец юниона.
.tag	Приписывает атрибуты структуры метке

8.8.8. Символы во время трансляции

Мнемоника и синтаксис	Описание
.asg [“]строка знаков[“], символ замены	Назначает строку знаков символу замены
.eval четкое выражение, символ замены	Исполняет арифметику на числовом символе замены.
.label символ	Определяет переместимую во время загрузки метку в разделе.
символ .set значение	Приравнивает значение символу.

8.8.9. Разные директивы

Мнемоника и синтаксис	Описание
.asmfunc	Начало блока кода для функции
. cdecls [options,]"filename" [, "filename2"[, ...]	Согласование дескрипторов для кодов С и ассемблера.
.clink [”имя раздела”]	Допускает условную компоновку для текущего или указанного раздела
.emsg строка	Посылает определяемые пользователем сообщения об ошибке устройству вывода; не производит объектный файл
.end	Заканчивает программу
.endasmfunc	Конец блока кода для функции
.mmsg строка	Посылает определяемые пользователем сообщения устройству вывода
.newblock	Снимает определение локальных меток
.sblock	Создание раздела блокирования
.wmsg строка	Посылает определяемые пользователем предупреждающие сообщения устройству вывода

8.9. Макроязык и макрокоманды

Ассемблер поддерживает макроязык, который дает Вам возможность создать ваши собственные команды. Это особенно полезно, когда программа выполняет частные задачи несколько раз. Макроязык позволяет Вам:

- Определить ваши собственные макрокоманды, и переопределить существующие макрокоманды.
- Упростить длинный или сложный ассемблерный код.
- Обратиться к макробиблиотекам, созданным архиватором.
- Определить условные и повторяемые блоки в пределах макрокоманд.
- Управлять строками в пределах макрокоманд.
- Управлять листингом расширения макрокоманд.

Программы часто содержат подпрограммы, которые выполняются несколько раз. Вместо повтора исходных инструкций подпрограммы, Вы можете определять подпрограмму как макрокоманду, а затем вызывать макрокоманду там, где Вы повторяли бы данную подпрограмму. Это упрощает и сокращает вашу исходную программу.

Если Вы хотите вызвать макрокоманду несколько раз, но каждый раз с различными данными, Вы можете назначить параметры в пределах макрокоманды. Это дает Вам возможность обеспечивать различную информацию макрокоманде каждый раз, когда Вы ее вызываете. Макроязык поддерживает специальный символ, называемый символом замены, который используется для параметров макрокоманды.

Использование макрокоманды - процесс из 3 шагов.

Шаг 1: Определение макрокоманды

Вы должны определить макрокоманды прежде, чем Вы можете использовать их в вашей программе. Имеются два метода для определения макрокоманд:

- Макрокоманды могут быть определены в начале исходного файла или в копируемом / включаемом файле.
- Макрокоманды могут также быть определены в макробиблиотеке. Макробиблиотека является собранием файлов в формате архива, созданном архиватором. Каждый элемент архивного файла (макробиблиотеки) может содержать одно макроопределение, соответствующее имени этого элемента. Вы можете обратиться к макробиблиотеке, используя .mlib директиву.

Шаг 2: Вызов макрокоманды

После того, как Вы определили макрокоманду, вызовите ее, используя имя макрокоманды как мнемонику в тексте исходной программы. Это называется вызовом макрокоманды.

Шаг 3: Расширение макрокоманды

Ассемблер разворачивает ваши макрокоманды, когда исходная программа вызывает их. Во время расширения ассемблер передает переменные аргументы параметрам макрокоманды, заменяет инструкция макровызова определением макрокоманды, затем транслирует исходный код. По умолчанию, макрорасширения печатаются в файле листинга. Вы можете выключить распечатку расширения, используя директиву .mnolist. Для получения дополнительной информации, см. раздел 5.8, Использование директив для форматирования листинга.

Когда ассемблер сталкивается с макроопределением, он размещает имя макрокоманды в таблице кодов операций (opcode table). Это переопределяет, любую предварительно определенную макрокоманду, библиотечный вход, директиву, или мнемонику команды, которые имеют то же самое имя, что и данная макрокоманда. Это позволяет Вам расширить функции директив и команд, а также добавить новые команды.

Определение макрокоманд. Вы можете определять макрокоманду где-либо в вашей программе, но Вы должны определить ее прежде, чем Вы сможете ее использовать. Макрокоманды могут быть определены в начале исходного файла, или в копируемом/включаемом файле, или в макробиблиотеке. Макроопределения могут быть вложенными, и они могут вызывать другие макрокоманды, но все элементы макрокоманды должны быть определены в том же самом файле. Макроопределение - ряд исходных инструкций в следующем формате:

имя макрокоманды **.**macro [параметр 1] [**,** ...**,** параметр n]

- **Имя макрокоманды** называет макрокоманду. Вы должны поместить имя в поле метки исходной инструкции. Только первые 128 знаков имени существенны. Ассемблер размещает макро-имя во внутренней таблице кодов операций, заменяя любую команду или прежнее макроопределение, имеющее то же самое имя.
- **Директива** .**macro** идентифицирует исходную инструкцию, как первую строку макроопределения. Вы должны разместить .macro в поле кода операции.
- **Параметр 1,…, параметр n** - являются необязательными символами замены, которые появляются как операнды директивы .macro.

Пример. Определение, вызов и расширение макрокоманды. Код определяет макрокоманду sadd4 с четырьмя параметрами r1, r2, r3, r4.

```
1       sadd4   .macro  r1,r2,r3,r4

2       sadd4   r1, r2 ,r3, r4
```

```
4           r1 = r1 + r2 + r3 + r4
5           .endm
```

Макровызов: следующий код вызывает макрокоманду sadd4 с четырьмя параметрами:

```
10
11  00000000        sadd4    A0,A1,A2,A3
```

Макробиблиотеки. Один из способов определения макрокоманд - создание макробиблиотеки. Макробиблиотека - собрание файлов, которые содержат макроопределения. Вы должны использовать архиватор, чтобы собрать эти файлы, или элементы, в одном файле (называемом архивом). Каждый элемент макробиблиотеки содержит одно макроопределение. Файлы в макробиблиотеке должны быть не оттранслированными исходными файлами. Имя макрокоманды и имя элемента должны быть одинаковыми, а расширение имени файла с макрокомандой должно быть .asm.

Например:

Макрокоманда	Имя файла в макробиблиотеке
simple	simple.asm
add3	add3.asm

Вы можете обращаться к макробиблиотеке, используя .mlib директиву ассемблера. **Синтаксис**: **.**mlib имя файла

Когда ассемблер сталкивается с .mlib директивой, он открывает названную (в имени файла) библиотеку и создает таблицу содержания библиотеки. Ассемблер вводит имена индивидуальных элементов библиотеки в таблицы кодов операций в качестве библиотечных входов; это переопределяет любые существующие коды операции или макрокоманды, которые имеют то же самое имя. Если одна из этих макрокоманд вызывается, ассемблер извлекает вход библиотеки и загружает его в таблицу макрокоманд.

Ассемблер разворачивает этот библиотечный вход таким же образом, как он разворачивает другие макрокоманды. Извлекаются только те макрокоманды, которые фактически вызываются из библиотеки, и они извлекаются только один раз.

Рекурсивные и вложенные макрокоманды. Макроязык поддерживает рекурсивные и вложенные макровызовы. Это означает, что Вы можете вызывать

другие макрокоманды внутри макроопределения. Вы можете вкладывать макрокоманды глубиной до 32 уровней. Когда Вы используете рекурсивные макрокоманды, Вы вызываете макрокоманду из ее собственного определения (макрокоманда вызывает саму себя).

Когда Вы создаете рекурсивные или вложенные макрокоманды, Вы должны обратить особое внимание на аргументы, которые Вы передаете макропараметрам, потому что ассемблер использует динамический обзор параметров. Это означает, что вызываемая макрокоманда использует окружающую среду макрокоманды, из которой она вызвана.

Следующие директивы могут использоваться с макрокомандами. Директивы .macro, .mexit, .endm и .var допустимы только с макрокомандами; оставшиеся директивы - общие директивы языка ассемблера.

Создание макрокоманд:

Мнемоника и синтаксис	Описание
.endm	Завершает макроопределение
имя .macro [параметр 1] [, ..., параметр n]	Определяет макрокоманду с указанным именем
.mexit	Выполняет переход к .endm
.mlib имя файла	Указывает библиотеку, содержащую макроопределения

Управление символами замены:

Мнемоника и синтаксис	Описание
.asg [“]строка знаков[“]	Назначает знаковую строку символу замены
.eval четкое выражение, символ замены	Исполняет арифметику на числовом символе замены

Условная трансляция:

Мнемоника и синтаксис	Описание
.break [четкое выражение]	Прерывает трансляцию повторяемого блока (необязательная)
.endif	Заканчивает условную трансляцию
.endloop	Заканчивает трансляцию повторяемого блока
.else	Необязательный условный блок
.elseif четкое	Необязательный условный блок

выражение	
.if четкое выражение	Начинает условную трансляцию
.loop [четкое выражение]	Начинает трансляцию повторяемого блока

Создание сообщений во время трансляции:

Мнемоника и синтаксис	Описание
.emsg	Посылает сообщение об ошибке стандартному устройству вывода
.mmsg	Посылает сообщение стандартному устройству вывода
.wmsg	Посылает предупреждение стандартному устройству вывода

Форматирование листинга:

Мнемоника и синтаксис	Описание
.drlist	Разрешает печать всех директив листинга
.fcnolist	Подавляет печать определенных директив листинга
.fclist	Разрешает распечатку ложных условных блоков (по умолчанию)
.fcnolist	Подавляет распечатку ложных условных блоков
.mlist	Разрешает распечатку макрокоманды (по умолчанию)
.mnolist	Подавляет распечатку макрокоманды
.sslist	Разрешает распечатку расширений символов замены
.ssnolist	Подавляет листинг расширений символов замены (по умолчанию)

8.10. Компоновщик

Компоновщик (Linker) создает исполняемые модули, объединяя объектные файлы COFF.

Вызов компоновщика. Общий синтаксис для вызова компоновщика:

cl2000 -v28 --run_linker [опции] имя файла 1 ... имя файла n

Опции - могут появляться где-нибудь в командной строке или в командном файле компоновщика.

Имя файла 1…имя файла n - могут быть объектные файлы, командные файлы компоновщика, или архивные библиотеки. Заданное по умолчанию расширение для всех входных файлов - .obj; любое другое расширение должно быть явно определено. Компоновщик может определять, является ли входной файл объектным или файлом ASCII, который содержит команды компоновщика. Заданное по умолчанию имя файла вывода a.out, если только Вы не используете -o опцию, чтобы назвать выходной файл.

Имеются два метода для вызова компоновщика:

- Определить параметры и имена файлов в командной строке. Этот пример связывает два файла, file1.obj и file2.obj, и создают названный модуль вывода link.out.
  cl2000 -v28 --run_linker file1.obj file2.obj -o link.out
- Ввести команду cl2000 -v28 --run_linker без имен файла или параметров; компоновщик запрашивает их:
  Command files: Командные файлы.
  Object files [.obj]: Объектные файлы.
  Output file []: Выходной файл.
  Options: Опции.

Для командных файлов, введите один или большее количество имен командных файлов компоновщика.

Для объектных файлов, введите один или большее количество имен объектных файлов. Заданное по умолчанию расширение - .obj. Отделите имена файла пробелами или запятыми; если последний знак - запятая, компоновщик предлагает дополнительную строку для имен объектных файлов.

Выходной файл - имя выходного модуля компоновщика. Это имя отменяет любые опции –o, которые Вы вводите. Если нет параметра -o, и Вы не отвечаете на этот запрос, компоновщик создает объектный файл со значением по умолчанию - a.out.

Опции (параметры) - для дополнительных параметров, хотя Вы можете также ввести их в командный файл. Введите их с дефисами, так, как если бы Вы были в командной строке.

8.11. Архиватор

Архиватор позволяет Вам объединять несколько индивидуальных файлов в один файл архива. Например, Вы можете собрать несколько макрокоманд в

макробиблиотеку. Ассемблер ищет библиотеку и использует ее элементы, которые вызываются как макрокоманды исходным файлом. Вы можете также использовать архиватор, чтобы собрать группу объектных файлов в библиотеку объектных модулей. Компоновщик включает в библиотеку элементы, которые решают внешние ссылки в течение компоновки. Архиватор позволяет Вам изменить библиотеку, удаляя, заменяя, извлекая, или добавляя элементы.

Вы можете формировать библиотеки из любого типа файлов. И ассемблер и компоновщик принимают архивные библиотеки в качестве входа; ассемблер может использовать библиотеки, которые содержат индивидуальные исходные файлы, а компоновщик может использовать библиотеки, которые содержат отдельные объектные файлы.

Одно из наиболее полезных приложений архиватора - построение библиотек из объектных модулей. Например, Вы можете записать несколько арифметических подпрограмм, оттранслировать их и использовать архиватор, чтобы собрать объектные файлы в одну логическую группу. Вы можете тогда определить библиотеку объектных модулей как вход компоновщика. Компоновщик ищет эту библиотеку и включает те ее элементы, которые решают внешние ссылки.

Вы можете также использовать архиватор, чтобы формировать макробиблиотеки. Вы можете создавать несколько исходные файлы, каждый из которых содержит одиночную макрокоманду, и использовать архиватор, чтобы собрать эти макрокоманды в одну функциональную группу. Вы можете использовать директиву .mlib в течение трансляции, чтобы определить, какую макробиблиотеку нужно искать для макрокоманд, которые Вы вызываете.

Чтобы вызывать архиватор, введите:

ar2000 [-]команда [параметры] имя библиотеки [имя файла 1 ... имя файла n]

- [-]Команда - сообщает архиватору, как управлять существующими элементами библиотеки и любыми файлами с указанными именами. Перед командой может быть необязательный дефис. Вы должны использовать одну из следующих команд, когда Вы вызываете архиватор, но Вы можете использовать только одну команду в обращении к архиватору. Команды архиватора следующие:
- @ Использует содержание указанного файла вместо ввода в командной строке. Вы можете использовать эту команду, чтобы избежать ограничения на длину командной строки, наложенную операционной системой компьютера. Используйте (;) в начале строки в командном файле, чтобы включить комментарии.

- a Прибавляет указанные файлы к содержимому библиотеки. Эта команда не заменяет существующий элемент, который имеет то же самое имя, как добавленный файл; она просто добавляет новые элементы в конец архива.
- d Удаляет указанные элементы из библиотеки.
- r Заменяет указанные элементы библиотеки. Если Вы не указываете имена файлов, архиватор заменяет элементы библиотеки файлами с тем же самым именем из текущего каталога. Если указанный файл не найден в библиотеке,
- архиватор добавляет его, вместо замены.
- t Печатает оглавление библиотеки. Если Вы определяете имена файлов, только эти файлы печатаются. Если Вы не определяете никаких имен, архиватор перечисляет все элементы указанной библиотеки.
- x Извлекает указанные файлы. Если Вы не определяете имена элементов, архиватор извлекает все элементы библиотеки. Когда архиватор извлекает элемент, он просто копирует этот элемент в текущий каталог; он не удаляет его из библиотеки.

Параметры. В дополнение к одной из команд, Вы можете определять параметры. Чтобы использовать параметры, объедините их с командой; например, чтобы использовать команду a и опцию s, введите –as или as. Дефис - необязательный только для параметров архиватора. Это - параметры архиватора:

- -q (тишина). Подавляет сообщения о состоянии и заголовок.
- -s Печатает список глобальных символов, которые определены в библиотеке. Этот параметр допустим только с командами a, r и d.
- -u Заменяет библиотечные элементы только, если замена имеет более позднюю дату модификации. Вы должны использовать команду r с -u опцией, чтобы определить, какие элементы заменять.
- -v (подробный). Обеспечивает описание “файла за файлом” при создании новой библиотеки из старой библиотеки и ее элементы.

Имя библиотеки. Называет библиотеку архивов, которая будет сформирована или изменена. Если Вы не определите расширение для файла библиотеки, архиватор использует заданное по умолчанию расширение .lib.

Имена файлов. Называют индивидуальные файлы, которыми управляет команда. Эти файлы могут быть существующими элементами библиотеки или новыми файлами, которые будут добавлены в библиотеку. Когда Вы вводите имя файла, Вы должны ввести полное имя файла, включая расширение, если

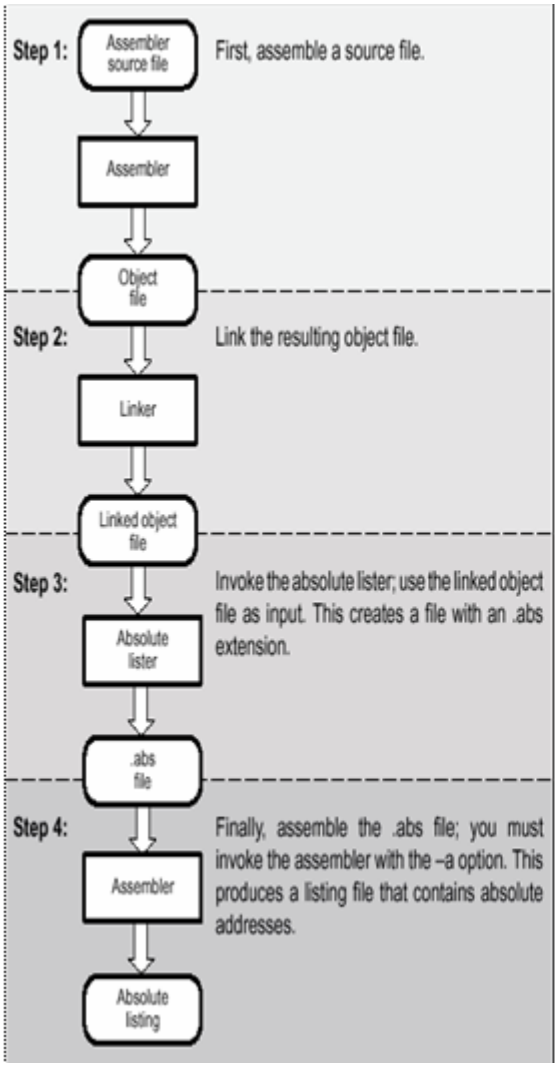
оно есть. Имя файла может иметь длину до 15 знаков; архиватор усекает имена файлов, которые длиннее, чем 15 знаков.

Внимание: для библиотеки возможно (но не желательно) содержать несколько членов с тем же самым именем. Если Вы пытаетесь удалять, заменять, или извлекать элемент, чье имя совпадает с именем другого библиотечного элемента, архиватор удаляет, заменяет, или извлекает первый библиотечный элемент с этим именем.

8.12. Абсолютный листер

Абсолютный листер - средство отладки, которое принимает скомпонованные объектные файлы в качестве входа, и создает файлы *.abs. Эти файлы *.abs можно оттранслировать, чтобы получить листинг, который показывает абсолютные адреса объектного кода. Вручную этот процесс требует длительного времени, однако абсолютный листер делает все автоматически.

Рисунок иллюстрирует шаги, требуемые, чтобы произвести абсолютный листинг.



ШАГ 1: Сначала оттранслируйте исх

ШАГ 2: Скомпонуйте результирующи

ШАГ 3: Вызовите абсолют скомпонованный объектный файл в к файл с расширением .abs.

ШАГ 4: Оттранслируйте файл . ассемблер с опцией –a. Он создаст содержит абсолютные адреса.

Синтаксис вызова абсолютного листера следующий:

abs2000 [-опции] входной файл

abs2000 - команда вызова абсолютного листера.

Опции - указывают опции, которые Вы хотите использовать. Они не чувствительны к регистру и могут появляться где-нибудь в командной строке. Предшествуйте каждую опцию дефисом (-). Возможны следующие опции:

- -e позволяет изменить заданное по умолчанию расширение для файлов:
- -ea [.] расширение для ассемблерных файлов (по умолчанию .asm).
- -ec [.] расширение для файлов на C (по умолчанию .c).
- -eh [.] расширение для файлов заголовка C (по умолчанию .h). Точка в расширении и пробел между опцией и расширением - необязательны.
- -q подавляет заголовок и всю информацию процесса работы.

Входной файл - именует скомпонованный объектный файл. Если Вы опускаете расширение имени входного файла, абсолютный листер считает, что входной файл имеет расширение .out. Если Вы не даете имя входного файла, абсолютный листер запрашивает его у Вас.

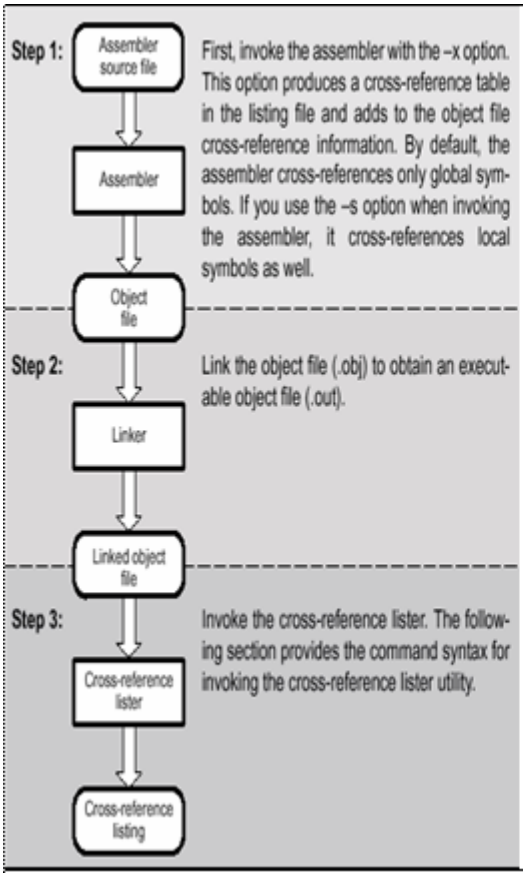
Абсолютный листер создает выходной файл для каждого входного файла, который был скомпонован. Эти файлы имеют то же имя, что и входные файлы, и расширение .abs. Файлы заголовка, однако, не создают соответствующий .abs файл. Транслируйте эти файлы с опцией ассемблера –a, как показано ниже, чтобы получить абсолютный листинг:

```
asm6x -a имя.abs
```

8.13. Листер перекрестных ссылок

Листер перекрестных ссылок - средство отладки. Эта утилита принимает в качестве входа скомпонованные объектные файлы и производит листинг перекрестных ссылок на выходе. Эта распечатка показывает символы, их определения и ссылки на них в скомпонованных объектных файлах.

Рисунок иллюстрирует шаги, требуемые, чтобы произвести распечатку перекрестных ссылок.



Шаг 1: Сначала, вызовите ассемблер
создает таблицу перекрестных ссылок
добавляет к объектному файлу информац

Шаг 2:Скомпонуйте объектный файл
выполняемый объектный файл (.out).

Шаг 3: Вызовите листер перекрестных сс
обеспечивает синтаксис команды
перекрестных ссылок.

Чтобы использовать утилиту перекрестной ссылки, файл должен быть собран с правильными параметрами и затем скомпонован в исполняемый файл. Транслируйте файлы ассемблера с -x опцией. Эта опция создает распечатку перекрестной ссылки и прибавляет информацию перекрестной ссылки к объектному файлу. По умолчанию ассемблер делает перекрестные ссылки только для глобальных символов, но если используется -s опция при вызове ассемблера, то также добавляются локальные символы. Скомпонуйте объектные файлы, чтобы получить выполняемый объектный файл.

- Листинг перекрестных ссылок

Листинг перекрестных ссылок показывает символы и их определения.

Чтобы вызвать листер перекрестных ссылок, введите следующее:

xref2000 [опции] [имя входного файла [имя файла вывода]]

xref2000 - команда, которая вызывает листер перекрестных ссылок.

Опции идентифицируют параметры листера перекрестных ссылок, которые Вы хотите использовать. Параметры не чувствительны к регистру и могут появляться где-либо в командной строке после команды. Предшествуйте каждую опцию дефисом (-). Эти опции следующие:

- -l (нижний регистр L) определяет число строк в странице выходного файла. Формат -l опции: -l число, где число - десятичная константа. Например, -l30 устанавливает число строк в странице выходного файла = 30. Пробел между опцией и константой - необязательный. Значение по умолчанию 60 строк.
- -q подавляет заголовок и всю информацию по работе (тихий запуск).

Имя входного файла – скомпонованный объектный файл. Если Вы опускаете входное имя файла, утилита его запрашивает.

Имя файла вывода – имя файла листинга перекрестных ссылок. Если Вы опускаете имя файла вывода, заданное по умолчанию имя файла - имя входного файла с расширением .xrf.

Пример. Листинг перекрестных ссылок ассемблера. Листер для каждого объявленного символа формирует описание ссылок. Листинг содержит:

- Symbol – имя символа.
- Filename - имя файла, где встречается символ.
- RTYP – тип символа
  STAT – определен в файле, не объявлен глобальным;
  EDEF – определен в файле, объявлен глобальным;
  EREF – не определен в файле, но ссылка глобальная;
  UNDF – не определен в файле, не объявлен глобальным;
- AsmVal – шестнадцатиричное значение символа в ассемблировании.
- LnkVal - шестнадцатиричное значение символа после установления ссылки.
- DefLn – Номер оператора, где обнаружен символ.

```
================================================================================

Symbol: _SETUP

Filename          RTYP    AsmVal     LnkVal      DefLn    RefLn     RefLn     Re
________          ____  __________ __________  _______  _______  ________  ___
demo.asm          EDEF  '00000018   00000018       18       13        20

================================================================================

Symbol: _fill_tab

Filename          RTYP    AsmVal     LnkVal      DefLn    RefLn     RefLn     Re
________          ____  __________ __________  _______  _______  ________  ___
ctrl.asm          EDEF  '00000000   00000040       10        5

================================================================================

Symbol: _x42

Filename          RTYP    AsmVal     LnkVal      DefLn    RefLn     RefLn     Re
________          ____  __________ __________  _______  _______  ________  ___
demo.asm          EDEF  '00000000   00000000        7        4        18

================================================================================

Symbol: gvar

Filename          RTYP    AsmVal     LnkVal      DefLn    RefLn     RefLn     Re
________          ____  __________ __________  _______  _______  ________  ___
tables.asm        EDEF  "00000000   08000000       11       10
================================================================================
```

8.14. Утилита 16-ричного преобразования

Ассемблер и компоновщик создают объектные файлы, которые находятся в формате общего объектного файла (COFF). COFF - двоичный формат объектного файла, который улучшает модульное программирование и обеспечивает мощные и гибкие методы для управления сегментами кода и памятью целевой системы.

Большинство программаторов ПЗУ не принимает COFF объектные файлы в качестве входа. Утилита шестнадцатеричного преобразования конвертирует COFF объектный файл в один из нескольких стандартных шестнадцатеричных форматов ASCII, подходящих для загрузки в программаторы ПЗУ. Утилита также полезна в других приложениях, требующих шестнадцатеричное преобразо-

вание COFF объектного файла (например, при использовании программ отладчиков и загрузчиков).

Утилита шестнадцатеричного преобразования может создавать следующие форматы выходного файла:

- ASCII-Hex (шестнадцатеричный), поддерживающий 16-разрядные адреса.
- Расширенный Tektronix (Tektronix).
- Intel MCS-86 (Intel).
- Motorola Exorciser (Motorola-S), поддерживающий 16-разрядные адреса.
- Texas Instruments SDSMAC (TI-Tagged), поддерживающий 16-разрядные адреса.

Чтобы вызвать утилиту шестнадцатеричного преобразования, введите следующее:

hex2000 [опции] [имя файла]

hex2000 - команда, которая вызывает утилиту шестнадцатеричного преобразования.

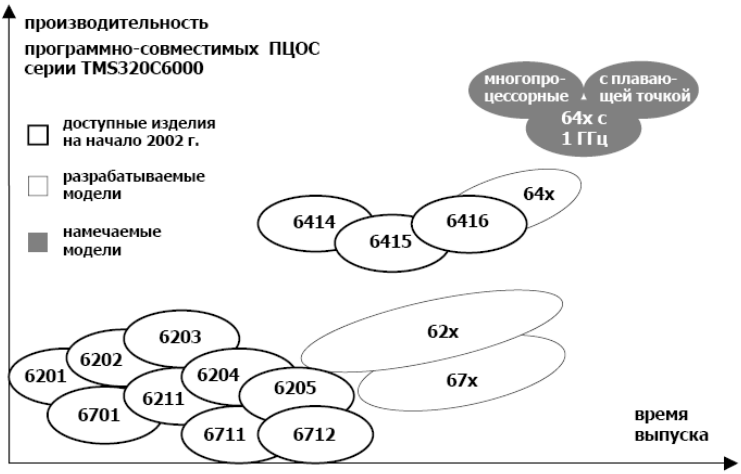
8.15. Согласование заголовочных C/C++ файлов с ассемблером

Директива .cdecls позволяет программисту согласованно использовать заголовочные C/C++ файлы и ассемблерные коды. Она обеспечивает автоматическое конвертирование методов из C кодов в ассемблерный код.

8.16. ИСР Code Composer Studio (CCS)

9. TMS320C6000

Серия TMS320C6x (сокращенно 'C6x)компании Texas Instruments Inc – это цифровые сигнальные процессоры (ЦСП) для обработки сигналов в реальном времени. Их отличает большая производительность. Высокая производительность достигается за счет внедрения параллельной архитектуры VelociTI, реализованной на основе технологии очень длинного командного слова VLIW (Very Long Instruction Word), а также за счет применения ряда других аппаратных решений и средств разработки.



По оценкам специалистов, применение данной архитектуры в будущем позволит, при сохранении совместимости по командам, достичь рубежей 8000 MIPS для ЦСП с фиксированной точкой и 3 GFLOPS для ЦСП с плавающей. Изготовляются и широко применяются следующие разновидности ЦСП серии TMS320C6000:

- TMS320C62x – устройства с фиксированной точкой и производительностью от 1200 до 2400 MIPS.
- TMS320C64x – устройства с фиксированной точкой и производительностью от 3200 до 4800 MIPS. Данные ЦСП являются наиболее скоростными.
- TMS320C67x – устройства с плавающей точкой и производительностью от 600 до 1350 MFLOPS.

Производительность ЦСП:

Семейство	Производительность			
	Тактовая частота МГц	MIPS/MFLOPS	MMAC (16 разрядные слова)	MMAC (8 разрядные слова)
TMS320C62x	150...300	1200...2400 MIPS	300...600	300...600
TMS320C64x	400...600	3200...4800 MIPS	1600...2400	3200...4800
TMS320C67x	100...225	600...1350 MFLOPS	200...550	200...550

Оценка продолжительности выполнения популярных алгоритмов.

Семейство	БПФ, комплексный спектр Выборка N=1024		Фильтр КИХ число выходов M=100	
	Тактов процессора	мкс	Тактов процессора	мкс
TMS320C62x	13228	66,0	6410	23,0
TMS320C64x	6002	12,0	1019	2,0
TMS320C67x	18055	108,3	2216	13,3

При проектировании ЦСП ‘C6x особое внимание изготовителя уделялось снижению времени, которое понадобится пользователю для разработки и выпуска конечных систем. Сокращению этих сроков способствует свойство совместимости устройства с фиксированной точкой с соответствующим устройством с плавающей точкой. ЦСП ‘C67x имеют совместимость по командам и по выводам микросхем с ЦСП ‘C62x, что позволяет разработчику быстро выполнять прототипы, используя плавающую точку, и легко переходить к ЦСП с фиксированной точкой для снижения стоимости изделия при производстве.

Вначале разработчик может взять за основу ЦСП с плавающей точкой, отработать все элементы устройства, определить оптимальные алгоритмы обработки данных. При этом большие запасы по производительности и по точности вычислений позволяют заниматься именно алгоритмами, а не экономией ресурсов. После, когда все параметры определены, наступает этап оптимизации системы с учетом наработанных решений и перевод ее на более дешевый ЦСП с фиксированной точкой.

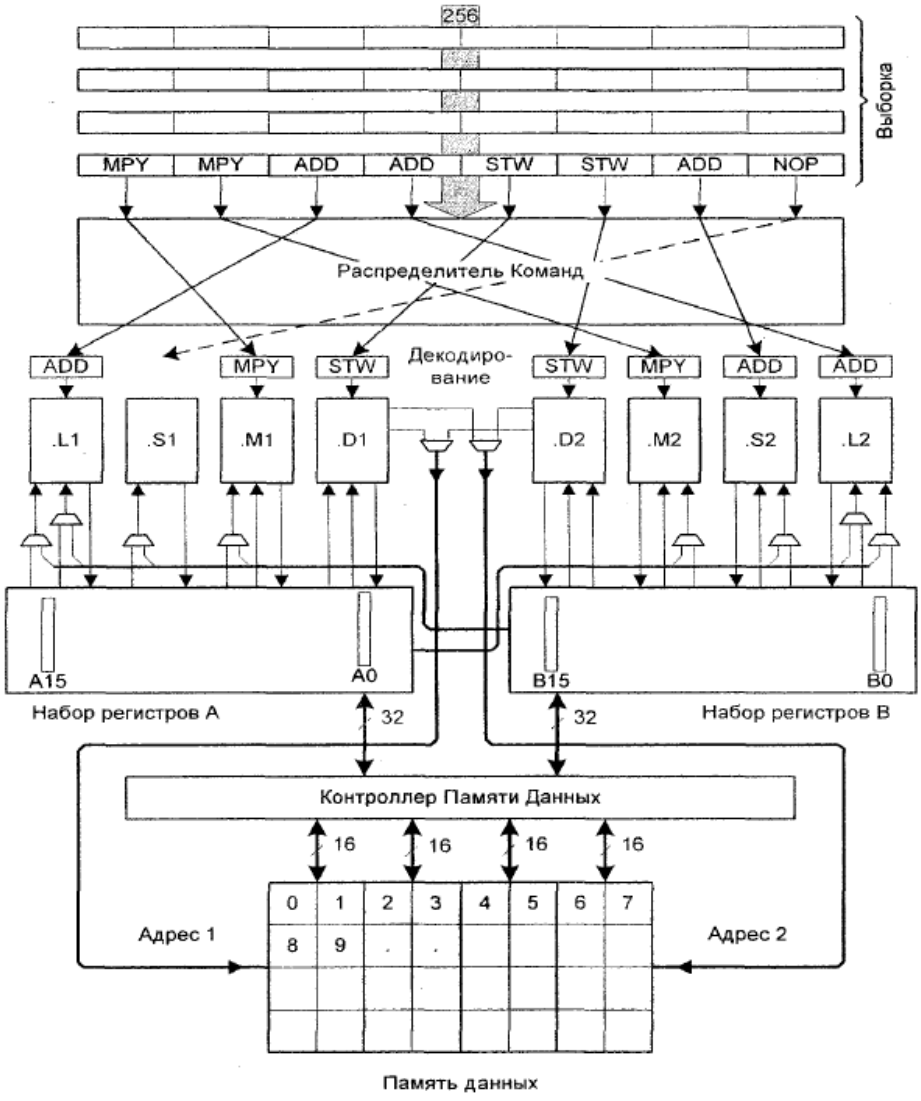
Данный подход предопределил переход от аппаратно-ориентированной среды разработки к программным моделям, что делает процесс разработки более быстрым, дешевым и простым.

9.1. Архитектура VelociTI

Все ЦСП C6x основаны на одном и том же 32-разрядном ядре центрального процессора с высоко параллельной и детерминированной архитектурой VelociTI.

Архитектура ядра ЦСП C6x включает 8 модулей - 2 умножителя и 6 АЛУ. Все модули максимально независимы, что дает компилятору и оптимизатору множество комбинаций их использования. На каждом такте ЦСП выбирается восемь 32-битных RISC-подобных инструкций. Предусмотренная в архитектуре VelociTI упаковка команд позволяет исполнять эти восемь инструкций параллельно, последовательно или параллельно/последовательно. Эта оптимизированная схема существенно снижает размер кода, количество выборок команд и потребление питания. При добавлении функции плавающей запятой к шести из восьми функциональных модулей из ЦСП с фиксированной точкой 'C62x получается ЦСП с плавающей точкой – 'C67x. При этом система команд 'C62x - расширение системы команд 'C67x и весь код написанный для 'C62x будет выполняться на 'C67x без модификаций самого кода.

Рассмотрим подробнее архитектуру ядра ЦСП серии TMS320C6000. Упрощенная схема ядра, без периферии и внешних шин, иллюстрирующая архитектуру VelociTI.


256
Выборка
MPY
MPY
ADD
ADD
STW
STW
ADD
NOP
Распределитель Команд
Декодиро-
вание
ADD
MPY
STW
STW
MPY
ADD
ADD
.L1
.S1
.M1
.D1
.D2
.M2
.S2
.L2
A15
A0
B15
B0
Набор регистров A
32
32
Набор регистров B
Контроллер Памяти Данных
16
16
16
16
0
1
2
3
4
5
6
7
8
9
Адрес 1
Адрес 2
Память данных

Как видно из рисунка, ЦСП использует очень длинные инструкции (256 бит) для выдачи до 8 команд по 32 бита для каждого из 8 функциональных модулей в каждом такте. Выбираются инструкции всегда по 256 бит, однако длина исполняемого пакета может быть разной, как показано на рисунке. Переменная длина выполняемой команды позволяет существенно сэкономить память – это отличительная черта ЦСП C6x от остальных ЦСП с очень длинным командным словом.

Ядро ЦСП C6x имеет два набора функциональных модулей. Каждый набор включает 4 модуля и регистровый файл. Каждый файл состоит из 16 32-разрядных регистров. Таким образом, всего в ядре 32 32-разрядных регистра.

Два набора функциональных модулей, связанных с двумя наборами регистров, создают разделение ядра на стороны A и B. 4 модуля с каждой стороны ЦСП C6x имеют произвольный доступ к регистровому файлу данной стороны. Кроме того, каждая сторона имеет шину, соединенную с регистровым файлом другой стороны. При доступе к регистрам своей стороны возможен доступ к регистрам всех модулей одновременно в одном такте.

Другой особенностью архитектуры ЦСП C6x является использование стратегии сохранения/загрузки, при которой все команды работают с регистрами. При этом два адресных модуля D1 и D2 выделяются только под передачу данных между регистровым файлом и памятью. Шины адреса, управляемые D-модулями, позволяют использовать адрес, сгенерированный в одном регистровом файле, для операций с данными в другом регистровом файле.

Ядро ЦСП C6x поддерживает широкий набор режимов косвенной адресации, включая линейный или кольцевой с 5- или 15-битным смещением. Все команды могут быть условными, и большинство команд могут использовать любой из 32 регистров. Некоторые регистры могут быть выделены для поддержки специфических режимов адресации или для хранения условий для условных команд.

2 M-модуля выделены под аппаратные умножители 16x16. Два S- и два L-модуля выполняют арифметические, логические операции и операции перехода, и при этом результаты их выполнения доступны в каждом такте (возможна задержка до 5 тактов конвейера, но большинство команд выполняются за 1 такт).

Процесс обработки команды в ядре начинается после выборки 256- битовой инструкции из внутренней памяти команд, которая также может быть сконфигурирована как кэш-память команд. Далее каждая из 32-битных команд распределяется на свой модуль для исполнения. При этом у команд, выполняемых на

разных модулях, проверяется младший бит. Он устанавливается в 1 для всех команд, которые должны выполняться одновременно. Команды, которые собраны для одновременного выполнения (до 8 команд), образуют пакет выполнения.

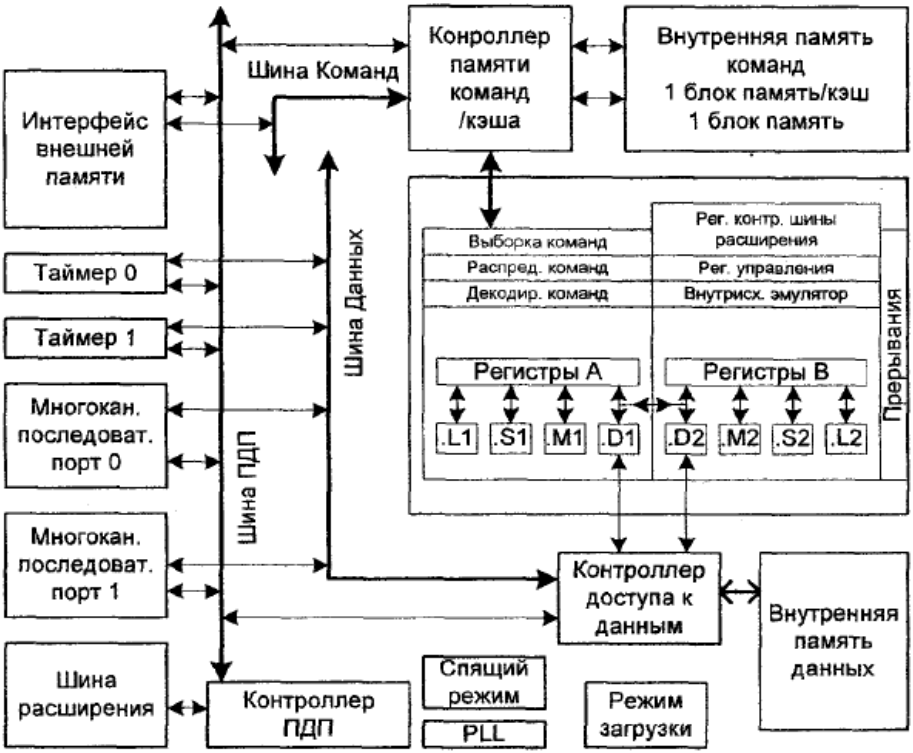
Ноль ставится в младшем бите команды, которая нарушает последовательность выполнения и откладывает команду на следующий пакет выполнения. Всего в выборке может быть до 8 пакетов выполнения. Очередной пакет размещается для выполнения в модулях в каждом такте. До окончания выполнения пакета следующий пакет выборки из памяти не выбирается. Эта «стратегия» позволяет существенно экономить память команд и менять режим работы программы от одновременного параллельного выполнения 8 команд на 8 модулях до практически последовательного выполнения команд, в зависимости от требований алгоритма.

Обратим внимание на организацию памяти данных. Как уже упоминалось, данные из функциональных модулей помещаются в регистры, а затем по адресам, генерируемым D-модулями, идет обмен с памятью данных. При этом каждый из регистровых файлов соединен 32 разрядными шинами с диспетчером памяти. Диспетчер организует одновременную выборку из памяти по четырем шинам до 64 разрядов по двум подаваемым адресам. При этом память дробится на множество мелких банков, что практически исключает конфликты доступа к памяти. Такое решение обеспечивает доступ без задержек при параллельных потоках обращения и при возможности адресовать отдельно каждый байт памяти. Фактически вся память данных ЦСП С6х организована не как двухпортовая, а как многопортовая, и количество одновременно выбираемых данных может меняться.

9.2. Структура и состав ЦСП С6х

На рисунке показана внутренняя структура ЦСП С6х. ЦСП можно условно разделить на несколько частей:

- Ядро процессора.
- Области памяти данных и памяти команд.
- Размещенная на кристалле периферия. Все эти части связаны между собой двумя контроллерами – памяти команд или кэш-памяти и памяти данных. Эти блоки связывают ядро ЦСП и банки памяти (с их специфической конфигурацией и доступом) с традиционными шинами, к которым подключаются периферийные модули и внешние устройства.



Рассмотрим подробнее периферийные устройства ЦСП TMS320C6000.

Контроллер ПДП. Устройство предназначено для передачи данных из памяти в память без участия центрального процессора. Контроллер ПДП имеет четыре основных программируемых и пять дополнительных каналов. Кроме того, контроллер ПДП используется при начальной загрузке программы в память ЦСП при старте (bootloader).

Хост «Порт-интерфейс» (ХПИ). ХПИ используется как для обмена данными с управляющим контроллером, так и для асинхронного обмена. ХПИ – это 16-разрядный параллельный порт, который обеспечивает прямой доступ к памяти

ЦСП. При этом ЦСП является управляющим устройством для данного интерфейса, что существенно упрощает процедуру доступа. ЦСП может обмениваться информацией, как через внутреннюю, так и через внешнюю память. Кроме того, ЦСП может иметь прямой доступ к большинству устройств размещенной на кристалле периферии.

Шина расширения (ШР). ШР является расширением как ХПИ, так и ИВП (см. ниже). С использованием ШР можно реализовать 32-разрядный ХПИ, который будет работать аналогично штатному 16-разрядному. ШР также может реализовать синхронный протокол обмена между хост ЦСП и ЦП, что дает возможность прямого подключения к большому набору стандартных шин хост ЦСП. Также к шине расширения могут быть подключено синхронное FIFO и асинхронные периферийные устройства.

Интерфейс внешней памяти (ИВП). ИВП это специальный блок, предназначенный для обмена данными с внешней памятью и быстродействующими внешними устройствами. ИВП может принимать запросы на обмен с внешней памятью от трех: контроллеров памяти данных, программной памяти-КЭШ и ПДП. Поскольку сам ЦСП – очень скоростное устройство, то ИВП не только выводит наружу классическую шину, но и имеет специальные сигналы для непосредственного подключения быстродействующего синхронного внешнего ОЗУ как динамического (SDRAM), так и статического (SBSRAM). Кроме того, к ИВП можно подключить и обычное статическое ОЗУ, ПЗУ, FIFO и другие устройства.

Начальный загрузчик. ЦСП TMS320C62x и TMS320C67x могут иметь множество режимов начальной загрузки, которые определяют, что именно будет делать ЦСП после сброса при подготовке к инициализации. Они могут включать загрузку программы с внешнего ПЗУ через ИВП или загрузку программы через ХПИ/ШР из внешнего устройства.

Многоканальный буферизованный последовательный порт МКБПП. Это последовательный скоростной порт, базирующийся на стандартном последовательном порте, как и в ЦСП других серий. Он имеет возможность читать и записывать данные в память без участия центрального процессора через контроллер ПДП. Кроме того, у него существуют многоканальные расширения, совместимые со стандартами EI, TI, SCSA и MVIP.

Отметим следующие функциональные возможности последовательного порта:

- полнодуплексная работа;
- двойная буферизация данных (позволяет поддерживать непрерывность потока);

- независимые тактовые частоты и схемы синхронизации для приема и передачи данных;,
- прямое подключение микросхем аналоговых интерфейсов, микросхем
- ЦАП и АЦП с последовательным интерфейсом.

МКБПП, по сравнению со стандартным последовательным портом, имеет дополнительные возможности:

- прямое подключение к шинам;
- многоканальный обмен при количестве каналов до 128;
- переменный размер данных 8, 12, 16, 20, 24 и 32 бита;
- встроенное u-Law и A-Law компандирование;
- возможность передачи первым старшего или младшего разряда данных;
- программируемая полярность сигналов синхронизации и тактовых сигналов данных;
- гибкое программирование внутренних тактовых импульсов и синхронизации.

Таймер. ЦСП серии TMS320C6000 имеют два 32-разрядных таймера, которые могут быть использованы для:

- задания временных событий;
- реализации счетчиков;
- генерации импульсов
- прерывания ЦСП;
- посылки синхроимпульсов в контроллер ПДП.

Селектор прерываний. Периферия ЦСП TMS320C6000 может иметь до 32-х источников прерываний. ЦП имеет возможность обрабатывать 12 прерываний. Селектор прерываний дает возможность выбора тех 12 прерываний, которые будут использоваться, и также дает возможность смены полярности внешних входов прерываний.

«Спящие» режимы. Логика снижения потребляемой мощности позволяет снимать тактовые сигналы с элементов ЦСП для снижения энергопотребления. Несмотря на свое предназначение для базовых станций, ЦСП TMS320C6000 также имеют режимы снижения энергопотребления. КМОП схемы в основном потребляют энергию в момент переключения, и чем выше частота работы, тем больше это потребление. При включении «спящих» режимов у ЦСП снимается тактовая частота сначала с ядра ЦСП, затем с периферии, размещенной на кристалле, и последний «третий» режим снимает тактовую частоту практически

со всего кристалла, в том числе и с блока умножения частоты. ЦСП имеет встроенный умножитель частоты с возможностью умножения внешней тактовой частоты на 2 и на 4, что делает возможным работу с низкой входной частотой и упрощает проектирование.

9.3. Средства разработки ЦСП C6x

Для разработчиков устройств на базе ЦСП серии C6x предлагается широкий набор мощных средств разработки и отладки. Новая архитектура ЦСП данного семейства предполагает и новый подход к процессу разработки, который позволяют уменьшить время и стоимость создания проекта за счет переноса большей части работы на ПО средств разработки. Разработчику остается написать алгоритм на языке высокого уровня, а его реализация и оптимизация с использованием всех преимуществ архитектуры ЦСП C6x перекладывается на компилятор. Это снимает одну из основных трудностей при работе на ЦСП с длинным командным словом – распараллеливание алгоритма. Такой подход имеет ряд преимуществ:

- Существенно сокращается срок разработки и качество получаемого продукта за счет сосредоточения именно на реализуемой задаче, а не на средствах ее реализации.
- Повышается качество и оптимальность кода за счет того, что автоматический оптимизатор всегда помнит все особенности архитектуры ЦСП и использует их по максимуму. Время разработки сокращается и за счет существенного уменьшения времени отладки из-за отсутствия ошибок в коде низкого уровня, которые часто возникают по вине разработчика (что-то забыл или не учел).

Процесс реализации алгоритма на ЦСП C6x протекает в несколько стадий. Вначале разработчик пишет алгоритм на языке Си или на ассемблере, и компилятор переводит его программу в машинный код с использованием всех возможностей ЦСП, таких как конвейерная обработка и интеллектуальное нахождение параллелизма в исходной программе для использования возможностей параллельной обработки команд в ЦСП. После наступает этап оценки производительности кода программными средствами, что позволяет оценить достигнутые результаты и провести оптимизацию кода без обращения к аппаратному обеспечению. И только следующим шагом идет проверка на макете устройства или отладочном модуле. Программные средства, предназначенные для разработки программ для ЦСП C6x:

- С-компилятор, ассемблер и компоновщик.

- Отладчик.
- Среда Code Composer Studio.

Высокоуровневый С-компилятор, ассемблер и компоновщик. Данные программные продукты представляют собой набор средств для компиляции кода языка С. Они специально ориентированы на реализацию оптимальных программ, созданных по алгоритмам ЦОС. Имеет широкий набор встроенных средств оптимизации, как общего плана, так и специализированных для ЦСП TMS320C6000. Является ANSI совместимым компилятором. В состав данного продукта входит ассемблерный оптимизатор - средство для перевода последовательного ассемблерного кода в параллельный, специфичный для ЦСП TMS320C6000.

Отдадчик. Продукт позволяет производить проверку выполнения кода на персональном компьютере без ЦСП.

Code Composer Studio. Многомодульный программный продукт представляет собой мощную интегрированную отладочную среду для ЦСП С6х и других серий. Имеет развитый оконный интерфейс, встроенные средства редактирования, возможность дизассемблирования и вызова внешнего компилятора, расширенные средства визуализации данных. По оценкам изготовителей среда Code Composer Studio станет стандартом и останется, чуть ли не единственным продуктом для программирования ЦСП компании Texas Instruments Inc. по крайней мере, до 2020 года. В настоящее время это средство платное.

9.4. Ассемблер ЦСП С6х

- Введение

Ассемблер преобразовывает (транслирует) исходные файлы ассемблера в объектные файлы в машинном коде. Эти файлы находятся в общем формате объектного файла (COFF). Исходные файлы могут содержать следующие элементы ассемблера:

- Директивы Ассемблера.
- Макро директивы.
- Команды ассемблера.

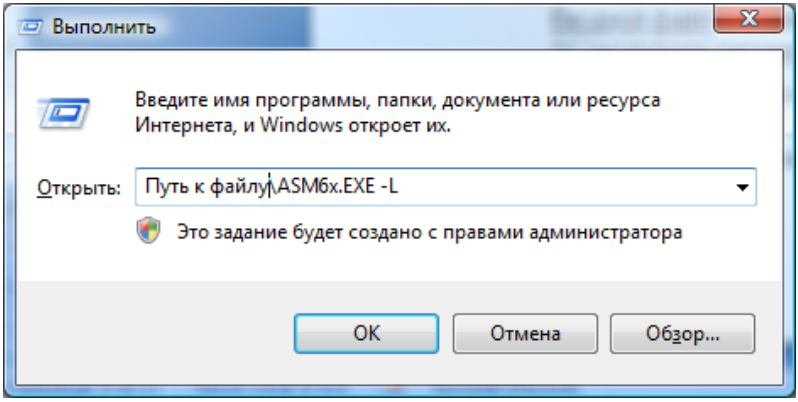
Двухпроходовый ассемблер делает следующее:

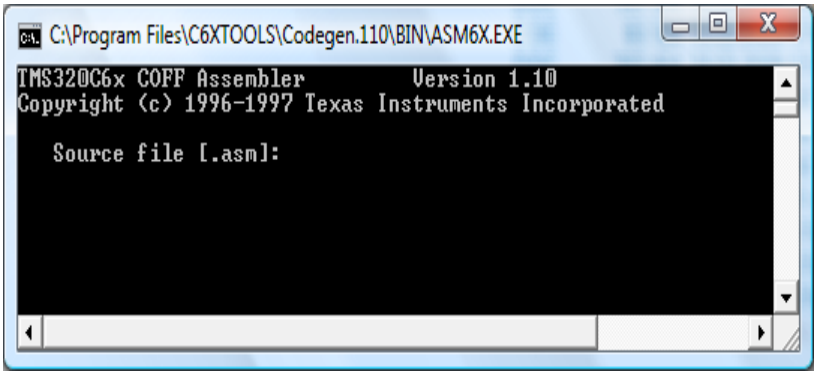
- Преобразует операторы исходника в объектный файл.

- Создает листинг исходника (если требуется) и дает Вам возможность управлять им.
- Позволяет Вам сегментировать код по разделам и устанавливает счетчик команд SPC в каждом разделе объектного кода.
- Определяет глобальные символы и ссылки на них, создает перекрестные ссылки на листинг исходника (если требуется).
- Допускает условное ассемблирование.
- Допускает макросы, позволяя создавать макросы в исходнике или в библиотеке.

Вызвать ассемблер можно двумя способами:

- Запустить ASM6X. Ассемблер запускается с опциями по умолчанию. В частности без опции –L, которая заставляет его формировать файл листинга.
- В командной строке ОС ввести путь к файлу ASM6x, имя файла и опцию –L. Файл листинга будет сформирован, и допущенные ошибки можно увидеть.





В окне ассемблера нужно задать:

- **Source file** - исходный файл ассемблера. Если Вы не даете расширение, ассемблер использует заданное по умолчанию расширение .asm.
- Формат инструкций исходника

Исходник ассемблера ЦСП C6x состоит из инструкций, которые могут содержать директивы ассемблера, команды ассемблера, макро-директивы, и комментарии. Инструкция может содержать 7 упорядоченных полей (метка, признак параллельности ||, условие, мнемоника инструкции, спецификатор модуля, список операндов, и комментарий).

Примеры инструкций:

```
two .set     2         ; Символ two = 2
Label:  MVK    two, A2 ; Запись значения two в регистр A2
.word   016h           ; Инициализация слова значением 016h
```

Ассемблер читает до 200 знаков в строке. Любые знаки свыше 200 усекаются. Операционная часть инструкций (т.е. все, кроме комментариев) должна быть короче 200 знаков для правильной трансляции. Комментарии могут простираться за пределы 200 знаков, но усеченная часть не включается в файл листинга.

Следуйте этим рекомендациям:

- Все инструкции должны начаться с метки, пробела, звездочки, или точки с запятой.
- Метки не обязательны, если они используется, они должны начаться в столбце 1.

- Один (или больше) пробелов должно отделять каждое поле. Символы табуляции интерпретируются, как пробелы. Вы должны отделить список операндов от предшествующего поля пробелом.
- Комментарии необязательны. Комментарии, которые начинаются в столбце 1, могут начинаться со звездочки или точки с запятой (* или ;), комментарии, которые начинаются в любом другом столбце должны начинаться с точки с запятой.
- Если Вы используете условную команду, имя регистра, по содержимому которого выполняется команда, должно быть окружено квадратными скобками.
- Спецификатор модуля необязательный. Если Вы не определяете функциональный блок, ассемблер назначает функциональный блок сам , основываясь на мнемоническом поле.
- Мнемоника не может начинаться с 0 или 1, иначе это будет интерпретироваться, как метка.

Метки. Они необязательны для всех команд ассемблера и для большинства (но не всех) директив ассемблера. Когда используется, метка должна начаться в столбце 1 инструкции. Метка может содержать до 128 алфавитно-цифровых знаков (A-Z, a-z, 0-9, _, и $). Метки **чувствительны к регистру**, и первый знак не может быть числом. Метка может сопровождаться двоеточием (:).

Если Вы не используете метку, знак в столбце 1 должен быть пробелом, звездочкой, или точкой с запятой.

Признак параллельности. Символы || указывают команды, которые выполняются параллельно с предыдущей командой. Вы можете иметь до восьми команд, выполняющихся параллельно. Следующий пример демонстрирует шесть команд (Inst1…Inst6), выполняющихся параллельно:

```
Inst1
|| Inst2
|| Inst3
|| Inst4
|| Inst5
|| Inst6
Inst7
```

Условие. Квадратные скобки [] указывают условные команды. Команда выполняется на основании значения регистра в пределах скобок, допустимые имена регистров – A1, A2, B0, B1, B2. Команда выполняется, если значение регистра

отлично от нуля. Если перед именем регистра стоит восклицательный знак (!), то команда выполняется, если значение регистра = 0. Например:

```
[A1]    ZERO  A2  ; Если A1 не равен 0, обнулить A2
```

Мнемоника инструкции. Например, ADD, MVK.

Поле спецификатора модуля. Это необязательное поле, которое следует за мнемоническим полем. Поле спецификатора модуля начинается с точки (.), сопровождаемой спецификатором функционального блока. Вообще, одна команда может быть назначена каждому функциональному блоку в одном командном цикле. Имеется восемь функциональных блоков, по два каждого функционального типа:

- .D1 и .D2 Данные/сложение/вычитание. Используются для формирования адресов памяти данных.
- .L1 и .L2 АЛУ/сравнение/арифметика длинных данных. АЛУ означает арифметико-логическое устройство.
- .M1 и .M2 Умножение.
- .S1 и .S2 Сдвиг/АЛУ/переходы/битовые поля.

Имеются несколько способов использовать поле спецификатора модуля:

- Вы можете определить конкретный функциональный блок (например, .D1).
- Вы можете определить только функциональный тип (например, .M), и ассемблер назначит определенный модуль (например, .M2).
- Если Вы не определяете функциональный блок, ассемблер назначает модуль, основываясь на мнемоническом поле.

Поле операнда. Оно следует за мнемоническим полем и содержит один или большее количество операндов. Поле операнда требуется не для всех команд или директив. Операнд состоит из следующих элементов: символы, константы выражения (комбинация констант и символов). Операнды друг от друга отделяются запятыми (никаких пробелов!).

Комментарий. Может начинаться в любом столбце и простирается до конца исходной строки. Комментарий может содержать любые знаки ASCII, включая пробелы. Комментарии печатаются в листинге программы ассемблера, но не влияют на процесс трансляции.

Исходная инструкция, которая содержит только комментарий, допустима. Такой комментарий – заголовок части кода. Если она начинается в столбце 1, то может начинаться с точки с запятой (;) или звездочки (*).

Комментарии, которые начинаются где-нибудь еще на строке, должны начинаться с точки с запятой. Такой комментарий описывает операцию в строке кода.

9.5. Команды ассемблера

9.5.1. Основные команды для работы с целыми числами

Инстр.	Мнемоника	Действие
ZERO	zero (.unit) dst .unit = .L1, .L2, S1, .S2, .D1, .D2	0 => dst
ADD ADDU	add (.unit) src1, src2,dst .unit = .L1, .L2, S1, S2	src1+src2 => dst
ADDK	addk (.unit) cst, dst .unit = .S1, .S2	cst+dst => dst
SUB SUBU	sub (.unit) src1, src2, dst .unit = .L1, .L2, S1, S2	src1=src2 => dst
ABS	abs (.unit) src, dst .unit = .L1, .L2	abs(src) => dst
B	b (.unit) label .unit = S1, S2	
CMPEQ	cmpeq (.unit) src1, src2, dst .unit = .L1, .L2	1 => dst при src1=src2
CMPGT CMPGTU	cmpgt (.unit) src1, src2, dst .unit = .L1, .L2	1 => dst при src1>src2
CMPLT CMPLTU	cmplt (.unit) src1, src2, dst .unit = .L1, .L2	1 => dst при src1<src2
MPY MPYU	mpy (.unit) src1, src2, dst .unit = .M1, .M2	src1*src2 => dst
MV	mv (.unit) src, dst .unit = .L1, .L2, S1, .S2, .D1, .D2	src => dst
MVK	mvk (.unit) cst, dst .unit = .S1, .S2	cst => dst
NEG	neg (.unit) src, dst .unit = .L1, .L2, .S1, .S2	=src => dst
NOP	nop	
STB STH	stb (.unit) src, *+baseR[offserR] .unit = .D1, .D2	src => baseR[offsetR]

STW		
LDB LDH LDW	ldb (.unit) *+baseR[offserR], dst .unit = .D1, .D2	baseR[offsetR] => dst
AND	and (.unit) src1, src2, dst .unit = .L1, .L2, S1, S2	src1 AND src2 => dst
OR	or (.unit) src1, src2, dst .unit = .L1, .L2, .S1, .S2	src1 OR src2 => dst
XOR	xor (.unit) src1, src2, dst .unit = .L1, .L2, S1, S2	src1 XOR src2 => dst
NOT	not (.unit) srt, dst .unit = .L1, .L2, .S1, .S2	
SHL	shl (.unit) src2, src1, dst .unit = .S1, .S2	(src2 на src1) => dst
SHR	shr (.unit) src2, src1, dst .unit = .S1, .S2	(src2 на src1) => dst

9.5.2. Основные команды для работы с вещественными числами

Инструкция	Мнемоника	Действие
ABSSP ABSDP	abssp (.unit) src, dst .unit = .S1, .S2	absdp(src) => dst
ADDSP ADDDP	addsp (.unit) src1, src2, dst .unit = .L1, .L2	src1+src2 => dst
SUBSP SUBDP	subsp (.unit) src1, src2, dst .unit = .L1, .L2	src1+src2 => dst
CMPEQSP CMPEQDP	cmpeqsp (.unit) src1, src2, dst .unit = .S1, .S2	1 => dst при src1=src2
CMPGTSP CMPGTDP	cmpgtsp(.unit) src1, src2, dst .unit = .S1, .S2	1 => dst при src1>src2
CMPLTSP CMPLTDP	cmpltsp (.unit) src1, src2, dst .unit = .S1, .S2	1 => dst при src1<src2
MPYSP MPYDP	mpysp (.unit) src1, src2, dst .unit = .M1, .M2	src1*src2 => dst
SPINT	spint (.unit) src, dst .unit = .L1, .L2	src1 => dst
INTSP	intsp (.unit) src, dst .unit = .L1, .L2	src1 => dst

DPINT	dpint (.unit) src, dst .unit = .L1, .L2	src1 => dst
INTDP	intdp (.unit) src, dst .unit = .L1, .L2	src1 => dst
DPSP	dpsp (.unit) src, dst .unit = .L1, .L2	src1 => dst
SPDP	spdp (.unit) src, dst .unit = .L1, .L2	src1 => dst

9.6. Константы

Ассемблер поддерживает каждую константу внутренне, как 32-разрядное число. Константы – не расширяются по знаку. Например, константа 00FFh равна 00FF (в 16-ричной системе) или 255 (в десятичной)..

Ассемблер поддерживает шесть типов констант:

- Двоичное целое число.
- Восьмеричное целое число.
- Десятичное целое число.
- Шестнадцатеричное целое число.
- Знак.
- Разовая ассемблерная константа.

Двоичная целочисленная константа. Это строка до 32 двоичных символов (0 и 1) с суффиксом B (или b). Если определено меньше 32 цифр, ассемблер выравнивает значение вправо и заполняет неопределенные левые биты нулями. Примеры допустимых двоичных констант:

0100000b Константа, равная 32 (10) или 20 (16)
01b Константа, равная 1 (10) или 1 (16)

Восьмеричная целочисленная константа. Это строка до 11 восьмеричных цифр (0 до 7) с суффиксом Q (или q). Примеры допустимых восьмеричных констант:

10Q Константа, равная 8 (10) или 8 (16).
226q Константа, равная 150 (10) или 96 (16).

Десятичная целочисленная константа. Это строка десятичных цифр (от 0 до 9). Им соответствуют десятичные числа в пределах от -2 147 483 648 до 4 294 967 295. Примеры допустимых десятичных констант:

1000 Константа, равная 1000 (10) или 3E8 (16).
-32768 Константа, равная -32 768 (10) или 8000 (16).
25 Константа, равная 25 (10) или 19 (16).

Шестнадцатеричная целочисленная константа. Это строка до восьми шестнадцатеричных цифр с суффиксом H (или h). Шестнадцатеричные цифры включают десятичные числа, 0-9, и символы A-F или a-f. Шестнадцатеричная константа должна начаться с десятичного числа (0-9). Если определено меньше, чем восемь шестнадцатеричных цифр, ассемблер выравнивает биты вправо. Примеры допустимых шестнадцатеричных констант:

78h Константа, равная 120 (10) или 0078 (16).
0Fh Константа, равная 15 (10) или 000F (16).

Символьная константа. Это – одиночный знак, заключенный в апострофы (одиночные кавычки). Знаки представлены внутренне, как знаки ASCII с 8 битами. Запоминается номер символа по таблице кодировки в двоичной форме. Примеры допустимых символьных констант:

'@' Определяет символьную константу, представлен внутренне как 61 (16-ричное 3d).
'C' Определяет символьную константу C и представлен внутренне как 43 (16-ричное 2b).
'' Определяет нулевой символ и представлен внутренне как 00 (16-ричное 00).

Обратите внимание на различие между символьными константами и символьными строками. Символьная константа представляет одиночное значение; строка - последовательность знаков.

Разовые константы ассемблера. Если Вы используете .set директиву, чтобы назначить значение символу, то символ становится константой. Чтобы использовать эту константу в выражениях, значение ее должно быть абсолютным (без знака). Например:

```
sym   .set   3
MVK   sym, B1
```

Вы можете также использовать .set директиву, чтобы назначить символическую константу имени регистра. В этом случае, символ становится синонимом регистра:

```
sym   .set   B1
```

```
MVK    10, sym
```

Символьная строка. Это строка кодовых знаков, заключенная в двойные кавычки. Двойные кавычки, которые являются частью символьной строки, представляются двумя последовательными двойными кавычками. Максимальная длина строки изменяется и определена для каждой директивы, которая требует символьную строку. Знаки представляются внутренне, как кодовые знаки ASCII с 8 битами. Примеры допустимых символьных строк:

“sample program” определяет строку sample program с 14 знаками.
“PLAN “”C””” определяет строку PLAN “C” с 8 знаками.

Символьные строки используются для следующих целей:

- Имена файлов, как в директиве .copy “имя файла”.
- Имена разделов, как в директиве .sect “имя раздела”.
- Директивы инициализации данных, как в .byte “символьная строка”.
- Операнды директив .string.

Символы. Используются как метки, константы и символы замены. Имя символа - строка до 200 алфавитно-цифровых знаков (A-Z, a-z, 0-9, $, и _). Первый знак в символе не может быть числом, и символы не могут содержать внутренние пробелы. Символы, которые Вы определяете, **чувствительны к регистру**. Например, ассемблер различает ABC, Abc и abc, как три уникальных символа. Символ допустим только внутри ассемблерной программы.

Метки. Символы, используемые как метки, станут символическими адресами, которые связаны с ячейками памяти в программе. Метки, используемые локально, в пределах файла должны быть уникальны. Мнемонические коды операции и имена директив ассемблера без префикса (.) - допустимые имена меток.

Метки могут также использоваться, как операнды .global, .ref, .def, или .bss директив. Например:

```
        .global  label1
label2: MVK    label2, B3
        MVKH   label2, B3
        B    label1
        NOP    5
```

Локальные метки. Это специальные метки, чьи возможности и сила - временные. Локальная метка может быть определена двумя способами:

- $n, где n - десятичная цифра в диапазоне 0-9. Например, $ 4 и $ 1 являются допустимыми локальными метками..
- имя?, где имя - любое законное имя символа, как описано выше. Ассемблер заменяет вопросительный знак точкой, сопровождаемой уникальным числом. Когда исходный текст расширен, Вы не будете видеть уникальное число в файле листинга. Ваша метка появляется с вопросительным знаком, как это сделано в исходном определении. Вы не можете объявлять эту метку как глобальную.

Нормальные метки должны быть уникальны (они могут быть объявлены только однажды), и они могут использоваться как константы в поле операнда. Локальные метки, однако, могут быть отменены и определены снова. Локальные метки не могут быть определены директивами.

Символические константы. Символам могут быть присвоены постоянные значения. Используя константы, Вы можете сопоставлять имена с постоянными значениями. Директивы .set и .struct/.tag/.endstruct дают Вам возможность присвоить константам символические имена. Символические константы не могут быть переопределены.

Предопределенные символические константы. Ассемблер имеет несколько предопределенных символов, включая следующие типы:

- $, знак доллара, представляет текущее значение счетчик команд раздела (SPC). $ - перемещаемый символ.
- Символы регистров, включая A0-A15 и B0-B15.
- Регистры управления ЦП, включая следующее:

ACR	Регистр управления анализа
ADR	Регистр данных анализа
AMR	Регистр способа адресации
ARP	Регистр возврата анализа
CSR	Регистр управления состояния
ICR	Регистр очистки прерываний
IER	Регистр разрешения прерываний
IFR	Регистр флагов прерываний
NRP	Указатель возврата из немаскируемого прерывания
IRP	Указатель возврата из маскируемого прерывания
ISR	Указатель таблицы обслуживания прерываний
ISTP	Регистр установки прерываний
PCE1	Счетчик команд

PDATA_O	Выход программных данных
STRM_HOLD	Регистр удержания потока
TCR	Регистр управления тестом
IN (’C67x только) ввода.	Универсальный регистр
OUT (’C67x только)	Универсальный регистр вывода

Регистры управления в тексте можно вводить либо всеми знаками верхнего регистра, либо всеми – нижнего; например, CSR можно ввести как csr.

Символы замены. Символы могут быть назначены строковым значениям (переменным). Это позволяет Вам заменять символьные строки, приравнивая их символическим именам. Символы, которые представляют строки знаков, называются символами замены. Когда ассемблер сталкивается с символом замены, его строковое значение заменяется именем символа. В отличие от символических констант, символы замены могут быть переопределены. Строка может быть назначена символу замены где-нибудь в пределах программы. Например:

```
.global     _table
.asg        ”B14”, PAGEPTR
.asg        ”*+B15(4)”, LOCAL1
.asg        ”*+B15(8)”, LOCAL2
    LDW   *+PAGEPTR(_table),A0
    NOP   4
    STW   A0,LOCAL1
```

Когда Вы используете макрокоманды, символы замены важны, потому что макропараметры - фактически символы замены, которые назначены аргументу макрокоманды. Следующий код показывает, как символы замены используются в макрокоманде:

```
MAC   .macro      src1, src2, dst      ; макрокоманда умножения/сложения
      MPY   src1, src2, src2
      NOP
      ADD   src2, dst, dst
.     endm
* Вызов макрокоманды MAC
      MAC       A0,A1,A2
```

9.7. Выражения

9.7.1. Простые выражения

Выражение - константа, символ, или ряд констант и символов, разделенные арифметическими операторами. 32-разрядные диапазоны допустимых значений выражения: от -2147 483 648 до 2147 483 647 для знаковых значений, от 0 до 4 294 967 295 для значений без знака. Три основных фактора влияют на порядок выполнения выражения:

- **Круглые скобки.** Выражения, включенные в круглые скобки, всегда рассчитываются сначала. 8 / (4 / 2) = 4, но 8 / 4 / 2 = 1. Вы не можете заменять круглые скобки на фигурные скобки ({}) или квадратные скобки ([]).
- **Группы по старшинству.** Операторы, перечисленные ниже, разделены на девять групп по старшинству. Когда круглые скобки не определяют порядок оценки выражения, первой выполняется самая высокая по старшинству операция. 8+4/2=10 (сначала вычислено 4/2).
- **Выполнение слева направо.** Когда круглые скобки и группы по старшинству не определяют порядок оценки выражения, выражения вычисляются слева направо, кроме группы 1, в которой они вычисляются справа налево. 8/4*2=4, но 8/(4*2)=1.

Список операторов, которые могут использоваться в выражениях, в соответствие с группами старшинства.

Группа	Оператор	Описание
1	+ - ~ !	Унарный плюс Унарный минус Дополнение до 1 Логическое НЕ
2	* / %	Умножение Деление Модуль
3	+ -	Сложение Вычитание
4	<< >>	Сдвиг влево Сдвиг вправо
5	< < = >	Меньше чем Меньше или равно Больше чем

	> =	Больше или равно
6	= ! =	Равно Не равно
7	&	Поразрядное И
8	^	Поразрядное исключающее ИЛИ (XOR)
9	|	Поразрядное ИЛИ

Внимание: операторы группы 1 вычисляются справа налево. Все другие операторы вычисляются слева направо.

Ассемблер проверяет условия переполнения и антипереполнения, когда арифметические операции выполняются во время трансляции. Он дает предупреждение (Value truncated – значение усечено) всякий раз, когда происходит переполнение или антипереполнение. Ассемблер не проверяет переполнение или антипереполнение при умножении.

Четкие выражения. Некоторые директивы ассемблера требуют четких выражений в качестве операндов. Четкие выражения содержат только символы или разовые константы ассемблера, которые определены прежде, чем с ними сталкиваются в выражении. Значение четкого выражения должно быть абсолютным (без знака). Это - пример четкого выражения:

1000h+X

где X был предварительно определен как абсолютный символ.

9.7.2. Условные выражения

Ассемблер поддерживает условные операторы, которые могут использоваться в любом выражении. Они особенно полезны для условной трансляции. Условные операторы включают следующие:

Оператор	Описание
=	Равно
!=	Не равно
<	Меньше
<=	Меньше или равно
>	Больше
>=	Больше или равно

Условные выражения равны 1, если они истинны, и 0, если ложны и могут использоваться только на операндах эквивалентных типов. Например, абсолют-

ная величина сравнивается с абсолютной величиной, но не с перемещаемым значением.

9.7.3. Законные выражения

За исключением перечисленных ниже случаев в выражениях нет ограничений на использование операторов, констант, внутренне или внешне определенных символов.

Когда выражение содержит более чем один перемещаемый символ или не может быть вычислено во время ассемблирования, ассемблер кодирует и помещает его в объектный файл и оно вычисляется линкером. Если финальное значение выражения требует больше места, чем отведено для него, то Вы получите сообщение линкера об ошибке.

Исключения для законных выражений. При использовании в регистрах режима относительной адресации выражение в квадратных скобках должно быть четким. Например, *+A4[15]

9.8. Листинги

9.9. Листинги программ

Листинг программы показывает исходные инструкции и объектный код, который они производят. Чтобы получить файл листинга, вызовите ассемблер с опцией -L. Листинг печатается постранично. Незаполненную строку и строку заголовка, имеют наверху каждая страница распечатки. Любой заголовок, определенный .title директивой, печатается в строке заголовка. Номер страницы печатается справа от заголовка. Если Вы не используете .title директиву, печатается имя исходного файла. Ассемблер вставляет незаполненную строку ниже строки заголовка.

Каждая строка в исходном файле создает, по крайней мере, одну строку в файле листинга. Она содержит номер исходной инструкции, значение SPC, объектный код, и исходную инструкцию..

Пример показывает листинг ассемблера.

9.9.1. Листинг перекрестных ссылок

Листинг перекрестных ссылок показывает символы и их определения. Чтобы получить этот листинг, вызовите ассемблер с -x опцией или используйте .option

директиву с операндом X. Ассемблер добавляет перекрестную ссылку в конец листинга программы.

Пример. Листинг перекрестных ссылок ассемблера

```
LABEL              VALUE          DEFN      REF

.BIG_ENDIAN        00000000  0
.LITTLE_ENDIAN     00000001  0
.TMS320C6200       00000001  0
.TMS320C6700       00000000  0
.TMS320C6X         00000001  0
_func              00000000' 18
var1               00000000–      4    17
var2               00000004–      5    18
```

Заголовки столбцов:

- LABEL (Метка) содержит каждый символ, который был определен или упомянут во время трансляции.
- VALUE (Значение) содержит шестнадцатеричное число с 8 цифрами (которое является значением, назначенным символу) или имя, которое описывает атрибуты символа. Значению может также предваряться знаком, который описывает атрибуты символа.
- DEFN (Определение) содержит номер инструкции, которая определяет этот символ. Этот столбец пустой для неопределенных символов.
- REF (Ссылка) перечисляет номера строк инструкций, которые обращаются к этому символу. Пробел в этом столбце указывает, что символ никогда не использовался.

9.10. Директивы ассемблера

Директивы ассемблера поставляют данные программе и управляют процессом трансляции. Директивы ассемблера дают возможность Вам делать следующее:

- Транслировать код и данные в указанные разделы.
- Резервировать пространство в памяти для неинициализированных переменных.
- Управлять видом листинга.
- Инициализировать память.
- Транслировать условные блоки.

- Определять глобальные переменные.
- Определять библиотеки, из которых ассемблер может получить макрокоманды.
- Исследовать информацию о символьной отладке.

Таблица дает сводку директив ассемблера. Помимо директив ассемблера, указанных здесь, программные средства 'C6x поддерживают следующие директивы:

- Ассемблер использует несколько директив для макрокоманд. Макродирективы обсуждаются в главе 5, Макроязык; они не обсуждаются в этой главе.
- Оптимизатор ассемблера использует несколько директив, которые поставляют данные и управляют процессом оптимизации. Директивы оптимизатора Ассемблера обсуждены в Руководстве «Оптимизирующий компилятор C TMS320C6x»; они не обсуждаются в этой книге.
- Компилятор C использует директивы для символьной отладки. В отличие от других директив, директивы символьной отладки не используются в большинстве программ на языке ассемблера. Приложение B, Директивы символьной отладки, обсуждает эти директивы; они не обсуждаются в этой главе.

Внимание: Метки и комментарии не показаны в синтаксисе.

Любая исходная инструкция, которая содержит директиву, может также содержать метку и комментарий. Метки начинаются в первом столбце (они - единственные элементы, кроме комментариев, которые могут появляться в первом столбце), а комментарии должны начинаться с точки с запятой или звездочки, если комментарий - единственный элемент в строке. Чтобы улучшать разборчивость, метки и комментарии не показываются, как часть синтаксиса директив. В описании части, заключенные в квадратные скобки, могут пропускаться, ассемблер будет их задавать по умолчанию.

9.10.1. Директивы, которые определяют разделы

Мнемоника и синтаксис	Описание
bss символ, размер в байтах [, выравнивание [, сдвиг банка]]	Резервирует пространство в разделе .bss (неинициализированные данные)
.data	Транслирует в раздел .data

	(инициализированные данные)
.sect ”имя раздела”	Транслирует в названный (инициализированный) раздел
.text	Транслирует в раздел .text (выполняемый код)
символ .usect ”имя раздела”, размер в байтах [,выравнивание]	Резервирует пространство в названном разделе (неинициализированном)

9.10.2. Директивы, которые инициализируют константы (данные и память)

Мнемоника и синтаксис	Описание
.bss размер в байтах	Резервирует пространство в текущем разделе; метка указывает на конец зарезервированного пространства
.byte значение1 [,..., значениеN]	Инициализирует один или более байт в текущем разделе
.char значение1 [,..., значениеN]	Инициализирует один или более байт в текущем разделе
.double значение1 [,...,значениеN]	Инициализирует 64-битнуые константы с плавающей точкой, IEEE с двойной точностью
.field значение [, размер]	Инициализирует поле размером в битах (1-32) со значением
.float значение1 [,...,значениеN]	Инициализирует 32-битные константы с плавающей точкой, IEEE с однократной точностью
.half значение1 [,...,значениеN]	Инициализирует 16-разрядные целые числа
.int значение1 [,...,значениеN]	Инициализирует 32-разрядные целые числа
.long значение1 [,...,значениеN]	Инициализирует 32-разрядные дробные числа
.short значение1 [,...,значениеN]	Инициализирует 16-разрядные дробные числа
.space размер	Резервирует пространство в текущем разделе; метка указывает на начало зарезервированного пространства

.string {выраж.1 ”строка1”}	Инициализирует одну или более текстовых строк
.word значение1 [,...,значениеN]	Инициализирует 32-разрядные целые числа

9.10.3. Директивы, которые выравнивают счетчик команд раздела (SPC)

Мнемоника и синтаксис	Описание
.align [размер в байтах]	Выравнивает SPC на границе, указанной размером в байтах, который должен быть степенью 2; по умолчанию - 1 байт

9.10.4. Директивы, которые форматируют выходной листинг

Мнемоника и синтаксис	Описание
.drlist	Допускает распечатку всех строк директив (по умолчанию)
.drnolist	Подавляет распечатку определенных строк директив
.fclist	Позволяет распечатку ложного условного блока (по умолчанию)
.fcnolist	Подавляет распечатку ложного условного блока кода
.length [длина страницы]	Устанавливает длину страницы листинга программы
.list	Повторный запуск распечатки программы
.mlist	Позволяет распечатку макрокоманд и блоков циклов(по умолчанию)
.mnolist	Подавляет распечатку макрокоманд и блоков циклов
.nolist	Останавливает распечатку программы
.option опция1 [, опция2,...]	Выбирает опции листинга; доступны опции - A,B,D,H,L,M,N,O,R,T,W и X
.page	Пропускает страницу в распечатке программы
.sslist	Позволяет расширенный листинг символов замены
.ssnolist (по умолчанию)	Подавляет расширенный листинг символов замены
.tab размер	Устанавливает размер знаков табуляции (в символах)

.title “строка”	Печатает заголовок в начале страницы листинга
.width [ширина страницы]	Устанавливает ширину страницы распечатки программы

9.10.5. Директивы, которые ссылаются на другие файлы

Мнемоника и синтаксис	Описание
.copy [“]имя файла[“]	Включает исходные инструкции из другого файла
.def символ1 [,...,символN]	Идентифицирует один или более символов, которые определены в текущем модуле и могут использоваться в других модулях
.global символ1 [,...,символN]	Идентифицирует один или более глобальных символов
.include [“]имя файла[“]	Включает исходные инструкции из другого файла
.mlib [“]имя файла[“]	Определяет библиотеку макрокоманд
.ref символ1 [,...,символN]	Идентифицирует один или более символов, используемых в текущем модуле, которые определены в другом модуле

9.10.6. Директивы, которые допускают условную трансляцию

Мнемоника и синтаксис	Описание
.break [четкое выражение]	Заканчивает трансляцию .loop, если четкое выражение - истина. При использовании конструкции .loop, конструкция .break - необязательна
.else	Транслирует блок кода, если (.if четкое выражение) является ложным. При использовании конструкции .if, конструкция .else необязательна
.elseif четкое выражение	Транслирует блок, если .if четкое выражение является ложным, а условие .elseif - истинно. При использовании конструкции .if , конструкция .elseif - необязательна
.endif	Заканчивает блок кода .if
.endloop	Заканчивает блок кода .loop

.if четкое выражения	Транслирует блок, если четкое выражение является истинным
.loop [четкое выражение]	Начинает повторяемую трансляцию кодового блока; счетчик цикла определен четким выражением

9.10.7. Директивы, которые определяют символы во время трансляции

Мнемоника и синтаксис	Описание
.asg [“]строка знаков[“], символ замены	Назначает строку знаков символу замены
.endstruct	Заканчивает определение структуры
символ .equ значение	Приравнивает значение символу
.eval четкое выражение, символ замены	Исполняет арифметику на числовом символе замены
.label символ	Определяет переместимую во время загрузки метку в разделе
символ .set значение	Приравнивает значение символу
.struct	Начинает определение структуры
.tag структура	Приписывает атрибуты структуры метке
(Н) Разные директивы	
.clink [”имя раздела”]	Допускает условную компоновку для текущего или указанного раздела
.emsg строка	Посылает определяемые пользователем сообщения об ошибке устройству вывода; не производит объектный файл
.end	Заканчивает программу
.mmsg строка	Посылает определяемые пользователем сообщения устройству вывода
.newblock	Снимает определение локальных меток
.wmsg строка	Посылает определяемые пользователем предупреждающие сообщения устройству вывода

9.11. Макроязык и макрокоманды

Ассемблер поддерживает макроязык, который дает Вам возможность создать ваши собственные команды. Это особенно полезно, когда программа выполняет частные задачи несколько раз. Макроязык позволяет Вам:

- Определить ваши собственные макрокоманды, и переопределить существующие макрокоманды.
- Упростить длинный или сложный ассемблерный код.
- Обратиться к макробиблиотекам, созданным архиватором.
- Определить условные и повторяемые блоки в пределах макрокоманд.
- Управлять строками в пределах макрокоманд.
- Управлять листингом расширения макрокоманд.

Программы часто содержат подпрограммы, которые выполняются несколько раз. Вместо повтора исходных инструкций подпрограммы, Вы можете определять подпрограмму как макрокоманду, а затем вызывать макрокоманду там, где Вы повторяли бы данную подпрограмму. Это упрощает и сокращает вашу исходную программу.

Если Вы хотите вызвать макрокоманду несколько раз, но каждый раз с различными данными, Вы можете назначить параметры в пределах макрокоманды. Это дает Вам возможность обеспечивать различную информацию макрокоманде каждый раз, когда Вы ее вызываете. Макроязык поддерживает специальный символ, называемый символом замены, который используется для параметров макрокоманды.

Использование макрокоманды - процесс из 3 шагов.

Шаг 1: Определение макрокоманды

Вы должны определить макрокоманды прежде, чем Вы можете использовать их в вашей программе. Имеются два метода для определения макрокоманд:

- Макрокоманды могут быть определены в начале исходного файла или в копируемом / включаемом файле.
- Макрокоманды могут также быть определены в макробиблиотеке. Макробиблиотека является собранием файлов в формате архива, созданном архиватором. Каждый элемент архивного файла (макробиблиотеки) может содержать одно макроопределение, соответствующее имени этого элемента. Вы можете обратиться к макробиблиотеке, используя .mlib директиву.

Шаг 2: Вызов макрокоманды

После того, как Вы определили макрокоманду, вызовите ее, используя имя макрокоманды как мнемонику в тексте исходной программы. Это называется вызовом макрокоманды.

Шаг 3: Расширение макрокоманды

Ассемблер разворачивает ваши макрокоманды, когда исходная программа вызывает их. Во время расширения ассемблер передает переменные аргументы параметрам макрокоманды, заменяет инструкция макровызова определением макрокоманды, затем транслирует исходный код. По умолчанию, макрорасширения печатаются в файле листинга. Вы можете выключить распечатку расширения, используя директиву .mnolist. Для получения дополнительной информации, см. раздел 5.8, Использование директив для форматирования листинга.

Когда ассемблер сталкивается с макроопределением, он размещает имя макрокоманды в таблице кодов операций (opcode table). Это переопределяет, любую предварительно определенную макрокоманду, библиотечный вход, директиву, или мнемонику команды, которые имеют то же самое имя, что и данная макрокоманда. Это позволяет Вам расширить функции директив и команд, а также добавить новые команды.

Определение макрокоманд. Вы можете определять макрокоманду где-либо в вашей программе, но Вы должны определить ее прежде, чем Вы сможете ее использовать. Макрокоманды могут быть определены в начале исходного файла, или в копируемом/включаемом файле, или в макробиблиотеке. Макроопределения могут быть вложенными, и они могут вызывать другие макрокоманды, но все элементы макрокоманды должны быть определены в том же самом файле. Макроопределение - ряд исходных инструкций в следующем формате:

имя макрокоманды **.**macro [параметр 1] [**,** ...**,** параметр n]

- **Имя макрокоманды** называет макрокоманду. Вы должны поместить имя в поле метки исходной инструкции. Только первые 128 знаков имени существенны. Ассемблер размещает макро-имя во внутренней таблице кодов операций, заменяя любую команду или прежнее макроопределение, имеющее то же самое имя.
- **Директива .macro** идентифицирует исходную инструкцию, как первую строку макроопределения. Вы должны разместить .macro в поле кода операции.

- **Параметр 1,…, параметр n** - являются необязательными символами замены, которые появляются как операнды директивы .macro.

Пример. Определение, вызов и расширение макрокоманды. Код определяет макрокоманду sadd4 с четырьмя параметрами r1, r2, r3, r4.

```
1       sadd4   .macro  r1,r2,r3,r4
2       sadd4   r1, r2 ,r3, r4
4       r1 = r1 + r2 + r3 + r4
5       .endm
```

Макровызов: следующий код вызывает макрокоманду sadd4 с четырьмя параметрами:

```
10
11  00000000        sadd4   A0,A1,A2,A3
```

Макробиблиотеки. Один из способов определения макрокоманд - создание макробиблиотеки. Макробиблиотека - собрание файлов, которые содержат макроопределения. Вы должны использовать архиватор, чтобы собрать эти файлы, или элементы, в одном файле (называемом архивом). Каждый элемент макробиблиотеки содержит одно макроопределение. Файлы в макробиблиотеке должны быть не оттранслированными исходными файлами. Имя макрокоманды и имя элемента должны быть одинаковыми, а расширение имени файла с макрокомандой должно быть .asm.

Например:

Макрокоманда	Имя файла в макробиблиотеке
simple	simple.asm
add3	add3.asm

Вы можете обращаться к макробиблиотеке, используя .mlib директиву ассемблера. **Синтаксис**: **.**mlib имя файла

Когда ассемблер сталкивается с .mlib директивой, он открывает названную (в имени файла) библиотеку и создает таблицу содержания библиотеки. Ассемблер вводит имена индивидуальных элементов библиотеки в таблицы кодов операций в качестве библиотечных входов; это переопределяет любые существующие коды операции или макрокоманды, которые имеют то же самое имя. Если одна из этих макрокоманд вызывается, ассемблер извлекает вход библиотеки и загружает его в таблицу макрокоманд.

Ассемблер разворачивает этот библиотечный вход таким же образом, как он разворачивает другие макрокоманды. Извлекаются только те макрокоманды, которые фактически вызываются из библиотеки, и они извлекаются только один раз.

Рекурсивные и вложенные макрокоманды. Макроязык поддерживает рекурсивные и вложенные макровызовы. Это означает, что Вы можете вызывать другие макрокоманды внутри макроопределения. Вы можете вкладывать макрокоманды глубиной до 32 уровней. Когда Вы используете рекурсивные макрокоманды, Вы вызываете макрокоманду из ее собственного определения (макрокоманда вызывает саму себя).

Когда Вы создаете рекурсивные или вложенные макрокоманды, Вы должны обратить особое внимание на аргументы, которые Вы передаете макропараметрам, потому что ассемблер использует динамический обзор параметров. Это означает, что вызываемая макрокоманда использует окружающую среду макрокоманды, из которой она вызвана.

Следующие директивы могут использоваться с макрокомандами. Директивы .macro, .mexit, .endm и .var допустимы только с макрокомандами; оставшиеся директивы - общие директивы языка ассемблера.

Создание макрокоманд:

Мнемоника и синтаксис	Описание
.endm	Завершает макроопределение
имя .macro [параметр 1] [, ..., параметр n]	Определяет макрокоманду с указанным именем
.mexit	Выполняет переход к .endm
.mlib имя файла	Указывает библиотеку, содержащую макроопределения

Управление символами замены:

Мнемоника и синтаксис	Описание
.asg [“]строка знаков[“]	Назначает знаковую строку символу замены
.eval четкое выражение, символ замены	Исполняет арифметику на числовом символе замены
.var символ 1 [, символ 2, ..., символ n]	Определяет локальные символы макрокоманды

Условная трансляция:

Мнемоника и синтаксис	Описание
.break [четкое выражение]	Прерывает трансляцию повторяемого блока (необязательная)
.endif	Заканчивает условную трансляцию
.endloop	Заканчивает трансляцию повторяемого блока
.else	Необязательный условный блок
.elseif четкое выражение	Необязательный условный блок
.if четкое выражение	Начинает условную трансляцию
.loop [четкое выражение]	Начинает трансляцию повторяемого блока

Создание сообщений во время трансляции:

Мнемоника и синтаксис	Описание
.emsg	Посылает сообщение об ошибке стандартному устройству вывода
.mmsg	Посылает сообщение стандартному устройству вывода
.wmsg	Посылает предупреждение стандартному устройству вывода

Форматирование листинга:

Мнемоника и синтаксис	Описание
.fclist	Разрешает распечатку ложных условных блоков (по умолчанию)
.fcnolist	Подавляет распечатку ложных условных блоков
.mlist	Разрешает распечатку макрокоманды (по умолчанию)
.mnolist	Подавляет распечатку макрокоманды
.sslist	Разрешает распечатку расширений символов замены
.ssnolist	Подавляет листинг расширений символов замены (по умолчанию)

9.12. Компоновщик

Компоновщик (Linker) создает исполняемые модули, объединяя объектные файлы COFF.

Вызов компоновщика. Общий синтаксис для вызова компоновщика: LNK6X [опции] имя файла 1 ... имя файла n

Опции - могут появляться где-нибудь в командной строке или в командном файле компоновщика.

Имя файла 1…имя файла n - могут быть объектные файлы, командные файлы компоновщика, или архивные библиотеки. Заданное по умолчанию расширение для всех входных файлов - .obj; любое другое расширение должно быть явно определено. Компоновщик может определять, является ли входной файл объектным или файлом ASCII, который содержит команды компоновщика. Заданное по умолчанию имя файла вывода a.out, если только Вы не используете -o опцию, чтобы назвать выходной файл.

Имеются два метода для вызова компоновщика:

- Определить параметры и имена файлов в командной строке. Этот пример связывает два файла, file1.obj и file2.obj, и создают названный модуль вывода link.out.
  lnk6x file1.obj file2.obj -o link.out
- Ввести команду LNK6X без имен файла или параметров; компоновщик запрашивает их:

Command files: Командные файлы.
Object files [.obj]: Объектные файлы.
Output file []: Выходной файл.
Options: Опции.

Для командных файлов, введите один или большее количество имен командных файлов компоновщика.

Для объектных файлов, введите один или большее количество имен объектных файлов. Заданное по умолчанию расширение - .obj. Отделите имена файла пробелами или запятыми; если последний знак - запятая, компоновщик предлагает дополнительную строку для имен объектных файлов.

Выходной файл - имя выходного модуля компоновщика. Это имя отменяет любые опции –o, которые Вы вводите. Если нет параметра -o, и Вы не отвечаете

на этот запрос, компоновщик создает объектный файл со значением по умолчанию - a.out.

Опции (параметры) - для дополнительных параметров, хотя Вы можете также ввести их в командный файл. Введите их с дефисами, так, как если бы Вы были в командной строке.

9.13. Утилиты

9.13.1. Архиватор

Архиватор позволяет Вам объединять несколько индивидуальных файлов в один файл архива. Например, Вы можете собрать несколько макрокоманд в макробиблиотеку. Ассемблер ищет библиотеку и использует ее элементы, которые вызываются как макрокоманды исходным файлом. Вы можете также использовать архиватор, чтобы собрать группу объектных файлов в библиотеку объектных модулей. Компоновщик включает в библиотеку элементы, которые решают внешние ссылки в течение компоновки. Архиватор позволяет Вам изменить библиотеку, удаляя, заменяя, извлекая, или добавляя элементы.

Вы можете формировать библиотеки из любого типа файлов. И ассемблер и компоновщик принимают архивные библиотеки в качестве входа; ассемблер может использовать библиотеки, которые содержат индивидуальные исходные файлы, а компоновщик может использовать библиотеки, которые содержат отдельные объектные файлы.

Одно из наиболее полезных приложений архиватора - построение библиотек из объектных модулей. Например, Вы можете записать несколько арифметических подпрограмм, оттранслировать их и использовать архиватор, чтобы собрать объектные файлы в одну логическую группу. Вы можете тогда определить библиотеку объектных модулей как вход компоновщика. Компоновщик ищет эту библиотеку и включает те ее элементы, которые решают внешние ссылки.

Вы можете также использовать архиватор, чтобы формировать макробиблиотеки. Вы можете создавать несколько исходные файлы, каждый из которых содержит одиночную макрокоманду, и использовать архиватор, чтобы собрать эти макрокоманды в одну функциональную группу. Вы можете использовать директиву .mlib в течение трансляции, чтобы определить, какую макробиблиотеку нужно искать для макрокоманд, которые Вы вызываете.

Чтобы вызывать архиватор, введите:

AR6X [-]команда [параметры] имя библиотеки [имя файла 1 ... имя файла n]

- [-]Команда - сообщает архиватору, как управлять существующими элементами библиотеки и любыми файлами с указанными именами. Перед командой может быть необязательный дефис. Вы должны использовать одну из следующих команд, когда Вы вызываете архиватор, но Вы можете использовать только одну команду в обращении к архиватору. Команды архиватора следующие:
- @ Использует содержание указанного файла вместо ввода в командной строке. Вы можете использовать эту команду, чтобы избежать ограничения на длину командной строки, наложенную операционной системой компьютера. Используйте (;) в начале строки в командном файле, чтобы включить комментарии.
- a Прибавляет указанные файлы к содержимому библиотеки. Эта команда не заменяет существующий элемент, который имеет то же самое имя, как добавленный файл; она просто добавляет новые элементы в конец архива.
- d Удаляет указанные элементы из библиотеки.
- r Заменяет указанные элементы библиотеки. Если Вы не указываете имена файлов, архиватор заменяет элементы библиотеки файлами с тем же самым именем из текущего каталога. Если указанный файл не найден в библиотеке,
- архиватор добавляет его, вместо замены.
- t Печатает оглавление библиотеки. Если Вы определяете имена файлов, только эти файлы печатаются. Если Вы не определяете никаких имен, архиватор перечисляет все элементы указанной библиотеки.
- x Извлекает указанные файлы. Если Вы не определяете имена элементов, архиватор извлекает все элементы библиотеки. Когда архиватор извлекает элемент, он просто копирует этот элемент в текущий каталог; он не удаляет его из библиотеки.

Параметры. В дополнение к одной из команд, Вы можете определять параметры. Чтобы использовать параметры, объедините их с командой; например, чтобы использовать команду a и опцию s, введите –as или as. Дефис - необязательный только для параметров архиватора. Это - параметры архиватора:

- -q (тишина). Подавляет сообщения о состоянии и заголовок.
- -s Печатает список глобальных символов, которые определены в библиотеке. Этот параметр допустим только с командами a, r и d.

- -u Заменяет библиотечные элементы только, если замена имеет более позднюю дату модификации. Вы должны использовать команду r с -u опцией, чтобы определить, какие элементы заменять.
- -v (подробный). Обеспечивает описание “файла за файлом” при создании новой библиотеки из старой библиотеки и ее элементы.

Имя библиотеки. Называет библиотеку архивов, которая будет сформирована или изменена. Если Вы не определите расширение для файла библиотеки, архиватор использует заданное по умолчанию расширение .lib.

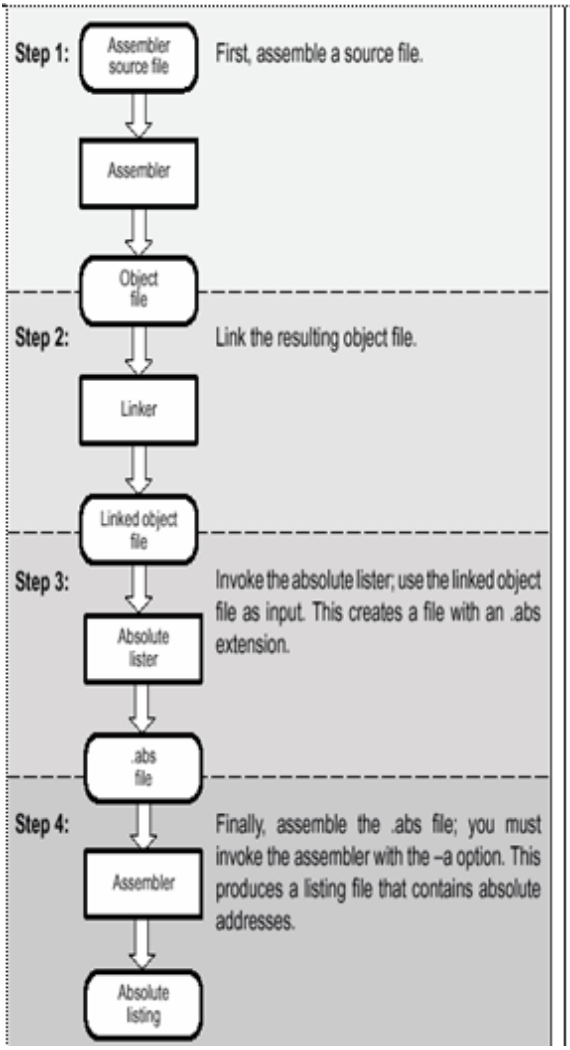
Имена файлов. Называют индивидуальные файлы, которыми управляет команда. Эти файлы могут быть существующими элементами библиотеки или новыми файлами, которые будут добавлены в библиотеку. Когда Вы вводите имя файла, Вы должны ввести полное имя файла, включая расширение, если оно есть. Имя файла может иметь длину до 15 знаков; архиватор усекает имена файлов, которые длиннее, чем 15 знаков.

Внимание: для библиотеки возможно (но не желательно) содержать несколько членов с тем же самым именем. Если Вы пытаетесь удалять, заменять, или извлекать элемент, чье имя совпадает с именем другого библиотечного элемента, архиватор удаляет, заменяет, или извлекает первый библиотечный элемент с этим именем.

- Абсолютный листер

Абсолютный листер - средство отладки, которое принимает скомпонованные объектные файлы в качестве входа, и создает файлы *.abs. Эти файлы *.abs можно оттранслировать, чтобы получить листинг, который показывает абсолютные адреса объектного кода. Вручную этот процесс требует длительного времени, однако абсолютный листер делает все автоматически.

Рисунок иллюстрирует шаги, требуемые, чтобы произвести абсолютный листинг.



ШАГ 1: Сначала оттранслируйте исхо

ШАГ 2: Скомпонуйте результирующи

ШАГ 3: Вызовите абсолютн скомпонованный объектный файл в ка файл с расширением .abs.

ШАГ 4: Оттранслируйте файл .а ассемблер с опцией –a. Он создаст ф содержит абсолютные адреса.

Синтаксис вызова абсолютного листера следующий:

ABS6X [-опции] входной файл

ABS6X - команда вызова абсолютного листера

Опции - указывают опции, которые Вы хотите использовать. Они не чувствительны к регистру и могут появляться где-нибудь в командной строке. Предшествуйте каждую опцию дефисом (-). Возможны следующие опции:

- -e позволяет изменить заданное по умолчанию расширение для файлов:
- -ea [.] расширение для ассемблерных файлов (по умолчанию .asm).
- -ec [.] расширение для файлов на C (по умолчанию .c).
- -eh [.] расширение для файлов заголовка C (по умолчанию .h). Точка в расширении и пробел между опцией и расширением - необязательны.
- -q подавляет заголовок и всю информацию процесса работы.

Входной файл - именует скомпонованный объектный файл. Если Вы опускаете расширение имени входного файла, абсолютный листер считает, что входной файл имеет расширение .out. Если Вы не даете имя входного файла, абсолютный листер запрашивает его у Вас.

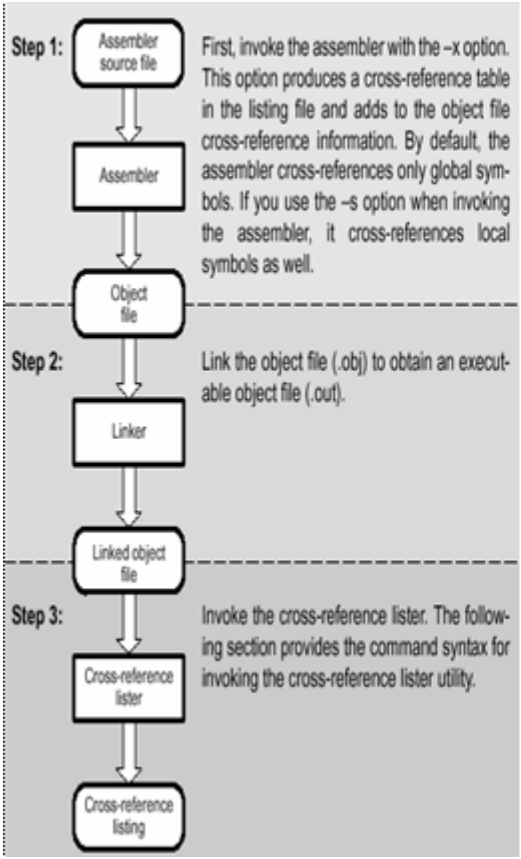
Абсолютный листер создает выходной файл для каждого входного файла, который был скомпонован. Эти файлы имеют то же имя, что и входные файлы, и расширение .abs. Файлы заголовка, однако, не создают соответствующий .abs файл. Транслируйте эти файлы с опцией ассемблера –a, как показано ниже, чтобы получить абсолютный листинг:

```
asm6x -a имя.abs
```

- Листер перекрестных ссылок

Листер перекрестных ссылок TMS320C6X - средство отладки. Эта утилита принимает в качестве входа скомпонованные объектные файлы и производит листинг перекрестных ссылок на выходе. Эта распечатка показывает символы, их определения и ссылки на них в скомпонованных объектных файлах.

Рисунок иллюстрирует шаги, требуемые, чтобы произвести распечатку перекрестных ссылок.



Шаг 1: Сначала, вызовите ассемблер с -x создает таблицу перекрестных ссылок в о добавляет к объектному файлу информац

Шаг 2:Скомпонуйте объектный файл (.obj выполняемый объектный файл (.out).

Шаг 3: Вызовите листер перекрестных сс обеспечивает синтаксис команды для ути перекрестных ссылок.

Чтобы использовать утилиту перекрестной ссылки, файл должен быть собран с правильными параметрами и затем скомпонован в исполняемый файл. Транслируйте файлы ассемблера с -x опцией. Эта опция создает распечатку перекрестной ссылки и прибавляет информацию перекрестной ссылки к объектному файлу. По умолчанию ассемблер делает перекрестные ссылки только для глобальных символов, но если используется -s опция при вызове ассемблера, то также добавляются локальные символы. Скомпонуйте объектные файлы, чтобы получить выполняемый объектный файл.

Чтобы вызвать листер перекрестных ссылок, введите следующее:

XREF6X [опции] [имя входного файла [имя файла вывода]]

XREF6X - команда, которая вызывает утилиту перекрестной ссылки.

Опции идентифицируют параметры листера перекрестных ссылок, которые Вы хотите использовать. Параметры не чувствительны к регистру и могут появляться где-либо в командной строке после команды. Предшествуйте каждую опцию дефисом (-). Эти опции следующие:

- -l (нижний регистр L) определяет число строк в странице выходного файла. Формат -l опции: -l число, где число - десятичная константа. Например, -l30 устанавливает число строк в странице выходного файла = 30. Пробел между опцией и константой - необязательный. Значение по умолчанию 60 строк.
- -q подавляет заголовок и всю информацию по работе (тихий запуск).

Имя входного файла – скомпонованный объектный файл. Если Вы опускаете входное имя файла, утилита его запрашивает.

Имя файла вывода – имя файла листинга перекрестных ссылок. Если Вы опускаете имя файла вывода, заданное по умолчанию имя файла - имя входного файла с расширением .xrf.

9.13.2. Утилита 16-ричного преобразования

Ассемблер и компоновщик ЦСП 'C6x создают объектные файлы, которые находятся в формате общего объектного файла (COFF). COFF - двоичный формат объектного файла, который улучшает модульное программирование и обеспечивает мощные и гибкие методы для управления сегментами кода и памятью целевой системы.

Большинство программаторов ПЗУ не принимает COFF объектные файлы в качестве входа. Утилита шестнадцатеричного преобразования конвертирует COFF объектный файл в один из нескольких стандартных шестнадцатеричных форматов ASCII, подходящих для загрузки в программаторы ПЗУ. Утилита также полезна в других приложениях, требующих шестнадцатеричное преобразование COFF объектного файла (например, при использовании программ отладчиков и загрузчиков).

Утилита 16-ричного преобразования может создавать следующие форматы выходного файла:

- ASCII-Hex (шестнадцатеричный), поддерживающий 16-разрядные адреса.

- Расширенный Tektronix (Tektronix).
- Intel MCS-86 (Intel).
- Motorola Exorciser (Motorola-S), поддерживающий 16-разрядные адреса.
- Texas Instruments SDSMAC (TI-Tagged), поддерживающий 16-разрядные адреса.

10. Поддержка в MATLAB

10.1. Введение

В СКМ MATLAB обеспечена поддержка моделирования устройств, использующих ЦСП. С их помощью устанавливается связь Simulink и MATLAB с инструментами eXpressDSP, разработанными компанией TI для работы с встроенными в хост-компьютер платами. Имеются пакеты расширения:

- Embedded Target for TI C2000 (tm) DSP – для работы с встроенными платами с ЦПОС серии C2000.
- Embedded Target for TI C6000 (tm) DSP - для работы с встроенными платами с ЦПОС серии C6000.
- Link for Code Composer Studio (tm) – для связи Simulink и MATLAB со средством разработки программ «Code Composer Studio» (CCS).

Каждый пакет для работы с встроенными платами включает:

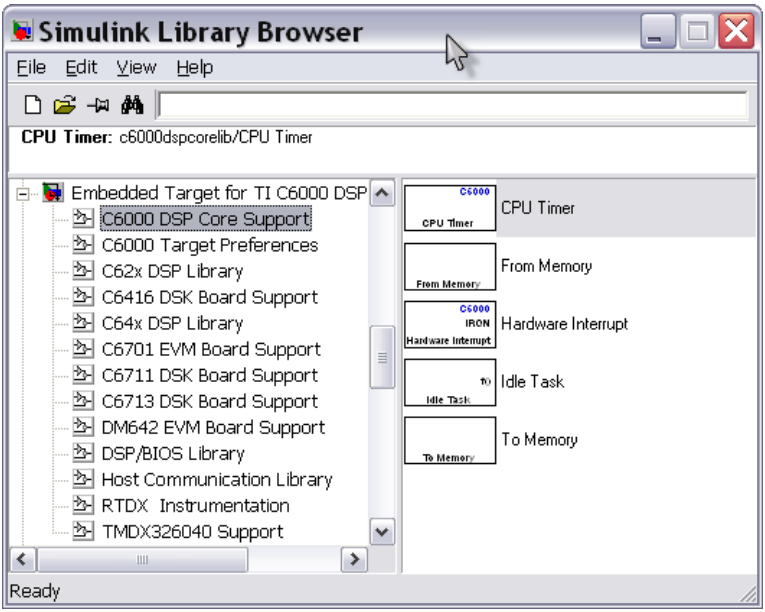
- Обзор встроенных плат с ЦПОС данного типа.
- Обзор библиотек подпрограмм для ЦПОС данного типа.
- Библиотеку блоков для Simulink.

В качестве примера рассмотрим ЦПОС серии C6000. Для этой серии поддерживаются следующие платы:

- C6416 DSP Starter Kit from TI – стартовый набор (DSK) с ЦПОС C6416.
- C6711 DSP Starter Kit from TI – DSK с ЦПОС C6711.
- C6713 DSP Starter Kit from TI – DSK с ЦПОС C6713.
- C6701 Evaluation Module from TI – отладочный модуль с ЦПОС C6701. От DSK отличается большими возможностями.
- TMDX326040A Daughter Card for the C6711 DSK – дочерняя звуковая карта для DSK C6711.

10.2. Встроенные платы для ЦСП ‘C6x

В Simulink поддерживаются блоки, сгруппированные по категориям в библиотеки. Для каждой библиотеки в правом поле отображаются входящие в нее блоки.



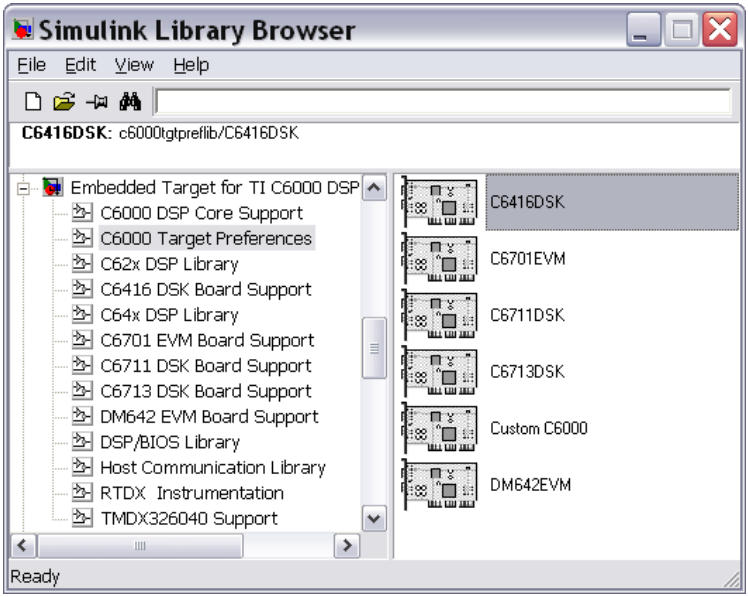
Имя	Содержание
C6000 DSP Core Support	Поддержка ядра ЦПОС C6000.
C6000 Target Preferences	Опциональные параметры платформы C6000.
C62x DSP Library	Библиотека ЦПОС C62x
C6416 DSK Library	Библиотека DSK с ЦПОС C6416.
C64x DSP Library	Библиотека ЦПОС C64x
C6701 EVM Board Support	Поддержка отладочного модуля с ЦПОС C6701.
C6711 EVM Board Support	Поддержка отладочного модуля с ЦПОС C6711.
C6713 EVM Board Support	Поддержка отладочного модуля с ЦПОС C6713.
DM642 EVM Board Support	Поддержка отладочного модуля DM642.
Host Communication Library	Библиотека связи с хостом.
RTDX Instrumentation	Инструментарий RTDX.
TMDX326040 Support	Поддержка TMDX326040.

Библиотека C6000 DSP Core Support. Поддержка ядра ЦПОС C6000. Включает блоки:

Имя	Содержание
CPU Timer	Таймер ЦПОС.

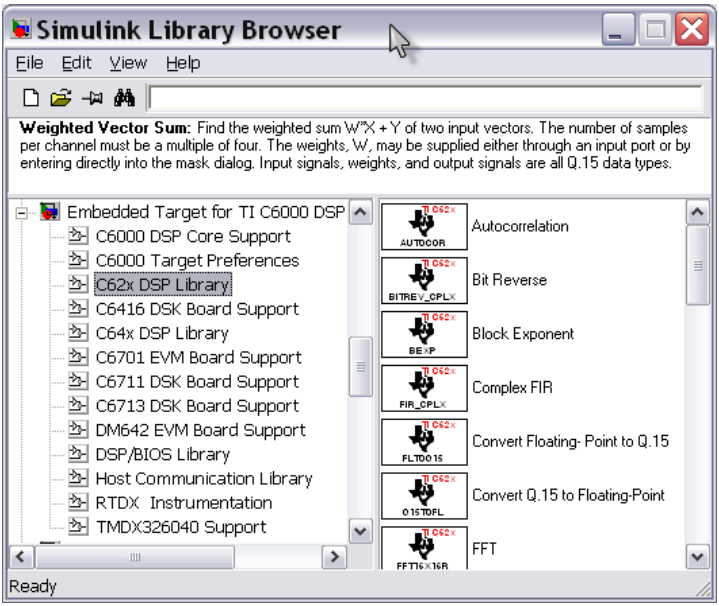
From Memory	Из памяти.
Hardware Interrupt	Аппаратное прерывание.
Idle Task	Холостая задача.
To Memory	В память.

Библиотека C6000 Target Preferences. Эти блоки можно использовать только при установленной на хост-компьютер платформе.



Имя	Содержание
C6416DSK	Для DSK с ЦПОС C6416.
C6701EVM	Для отладочного модуля с ЦПОС C6701
C6711DSK	Для DSK с ЦПОС C6711
C6713DSK	Для DSK с ЦПОС C6713
Custom C6000	Для платы конкретного пользователя
DM642EVM	Поддержка отладочного модуля DM642

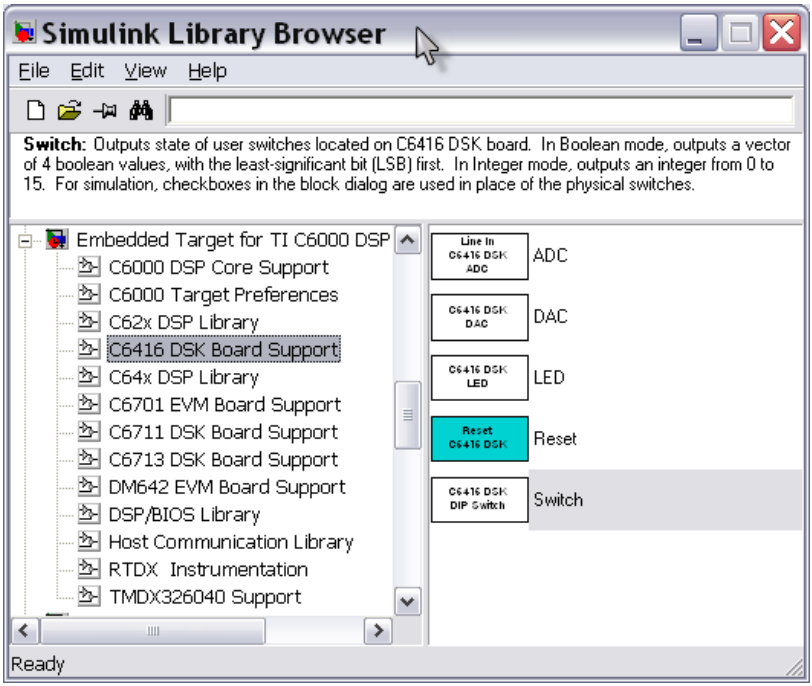
Библиотека C62x DSP Library.



Имя	Содержание
Autocorrelation	Автокорреляция векторов или матриц.
Bit Reverse	Реверс битов для каждого элемента входного комплексного вектора.
Block Exponent	Экспонента блока. Вычисление экспонент (число битов за пределами бита знака).
Complex FIR	КИХ фильтр комплексного сигнала.
Convert Floating-Point to Q.15	Преобразует вход (вещественный или комплексный) в формате с плавающей точкой с ординарной точностью в формат Q.15.
Convert Q.15 to Floating-Point	Преобразует вход в формате Q.15 (вещественный или комплексный) в сигнал в формате с плавающей точкой с ординарной точностью.
FFT	Прямое преобразование Фурье. Используется разделение по частоте.
General Real FIR	КИХ фильтр вещественного сигнала.
LMS Adaptive FIR	Адаптивный КИХ фильтр, синтезированный по алгоритму наименьших квадратов.

Matrix Multiply	Умножение матриц Y=A*B.
Matrix Transpose	Вычисляется транспонированная матрица.
Radix-2 FFT	Вычисляет прямое FFT комплексного вектора в режиме разделения по частоте с делением на 2.
Radix-2 IFFT	Вычисляет обратное FFT комплексного вектора в режиме разделения по частоте с делением на 2.
Radix-4 Real FFT	КИХ фильтр вещественного сигнала X. Коэффициенты фильтра образуют вещественный вектор H, их количество должно делиться на 4.
Radix-8 Real FFT	КИХ фильтр вещественного сигнала X. Коэффициенты фильтра образуют вещественный вектор H, их количество должно делиться на 8.
Real Forward Lattice All-Pole IIR	БИХ фильтр авто-регрессионного лестничного типа.
Real IIR	БИХ фильтр авто-регрессионного типа с использованием бегущего среднего.
Reciprocal	Преобразует входной сигнал из формата Q.15 в формат F*(2^E). F и E - вещественные знаковые 16-разрядные целые числа.
Symmetric Real FIR	КИХ симметричный фильтр. Коэффициенты фильтра образуют вещественный вектор H, симметричный относительно центрального элемента.
Vector Dot Product	Скалярное произведение векторов.
Vector Maximum Index	Индекс максимального значения в векторе.
Vector Maximum Value	Максимальное значение в векторе.
Vector Minimum Value	Максимальное значение в векторе.
Vector Multiply	Произведение векторов.
Vector Negate	Смена знака элементов вектора.
Vector Sum of Square	Сумма квадратов элементов вектора.
Weighted Vector Sum	Взвешенная сумма векторов W*X+Y. W – весовой коэффициент.

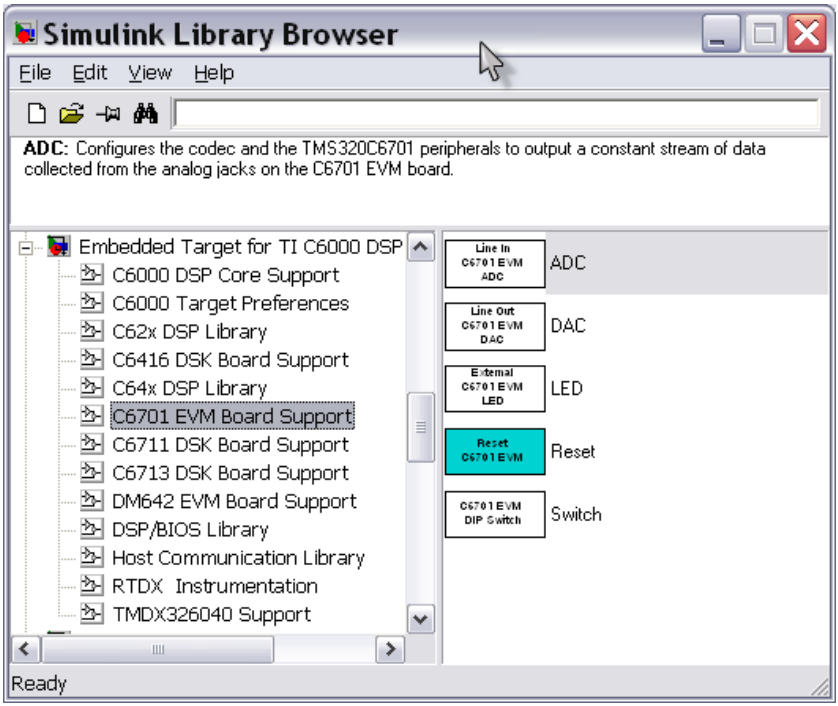
Библиотека C6416 DSK Library.



Имя	Содержание
ADC	Конфигурирует АЦП встроенного кодека.
DAC	Конфигурирует ЦАП встроенного кодека.
LED	Управляет светодиодами платы.
Reset	Сброс
Switch	Возвращает состояния встроенных переключателей платы.

Библиотека C64x DSP Library. Библиотека ЦПОС C64x. Включает блоки, аналогичные блокам, используемым в ЦПОС C62x.

Библиотека C6701 EVM Board Support. Поддержка отладочного модуля с ЦПОС C6701.

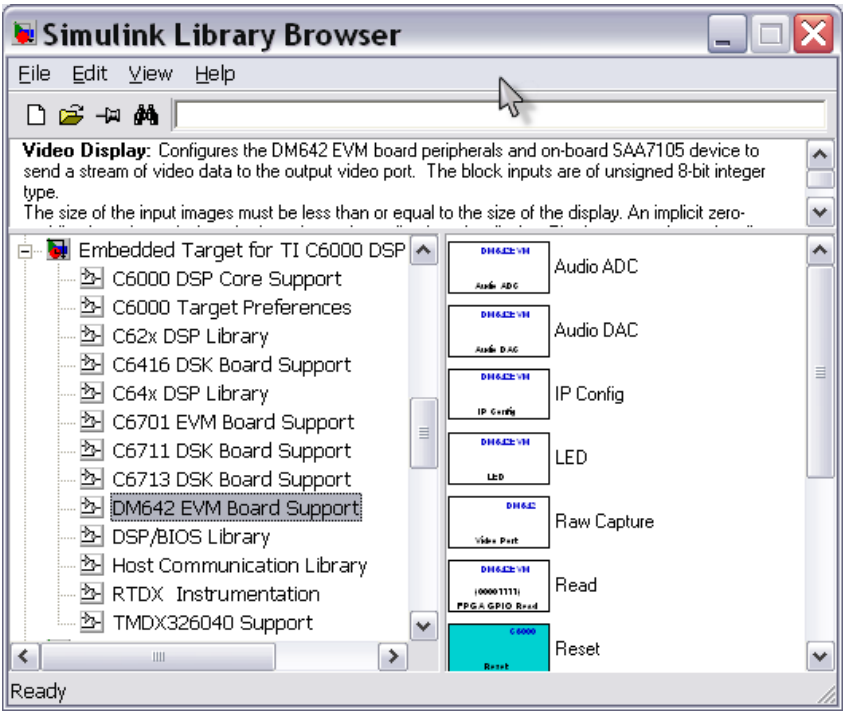


Имя	Содержание
ADC	Конфигурирует АЦП встроенного кодека.
DAC	Конфигурирует ЦАП встроенного кодека.
LED	Управляет светодиодами платы.
Reset	Сброс.
Switch	Возвращает состояния встроенных переключателей платы.

Библиотека C6711 EVM Board Support. Поддержка отладочного модуля с ЦПОС C6711. Включает блоки, аналогичные блокам, используемым в модуле C6701 EVM.

Библиотека C6713 EVM Board Support. Поддержка отладочного модуля с ЦПОС C6713. Включает блоки, аналогичные блокам, используемым в модуле C6701 EVM.

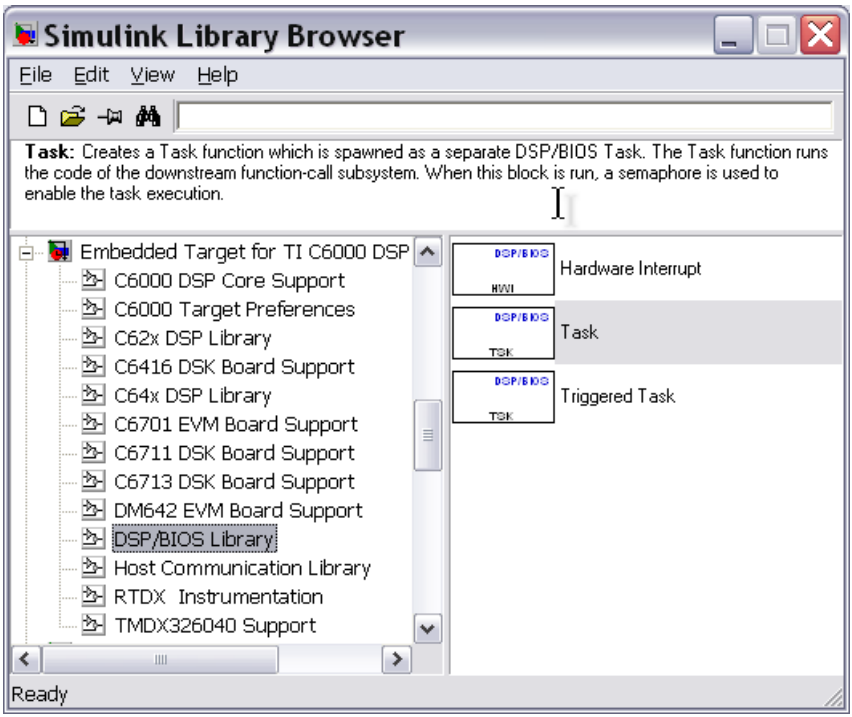
Библиотека DM642 EVM Board Support. Поддержка отладочного модуля DM642.



Имя	Содержание
Audio ADC	Конфигурирует звуковой АЦП встроенного кодека.
Audio DAC	Конфигурирует звуковой ЦАП встроенного кодека.
IP Config	Конфигурирует прерывания модуля.
LED	Управляет светодиодами платы.
Raw Capture	Конфигурирует видео порт.
Read	Чтение с внешних выводов
Reset	Сброс.
UDP Receive	Конфигурирует Ethernet драйвер для приема сообщений.
UDP Send	Конфигурирует Ethernet драйвер для передачи сообщений.
Video	Конфигурирует периферию модуля для приема от видео порта.

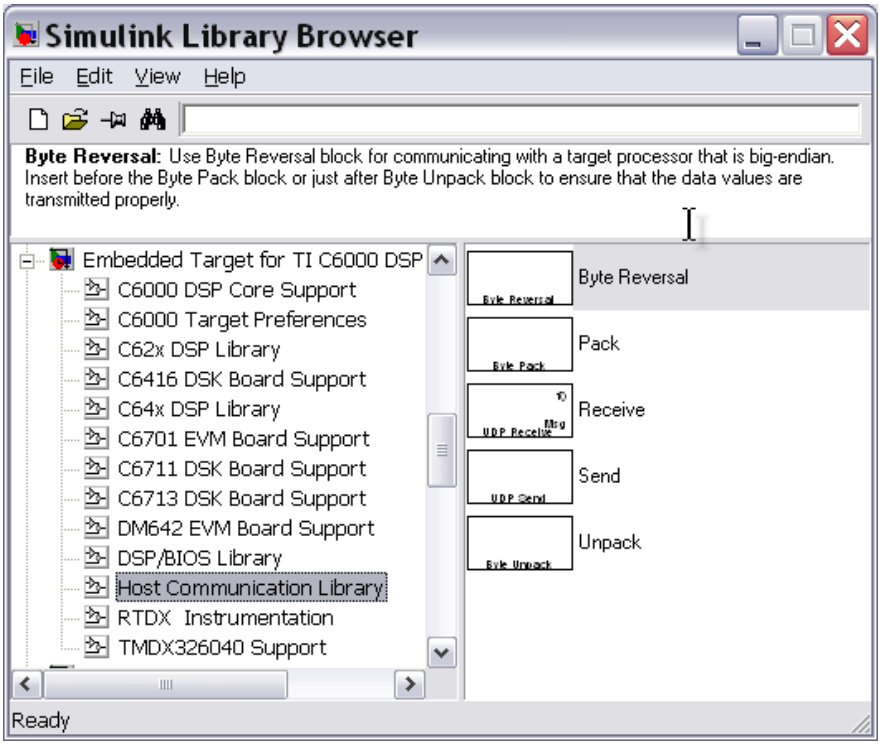
Capture	
Video Display	Конфигурирует периферию модуля для передачи на видео порт.
Write	Запись на внешние выводы.

Библиотека DSP/BIOS Library.



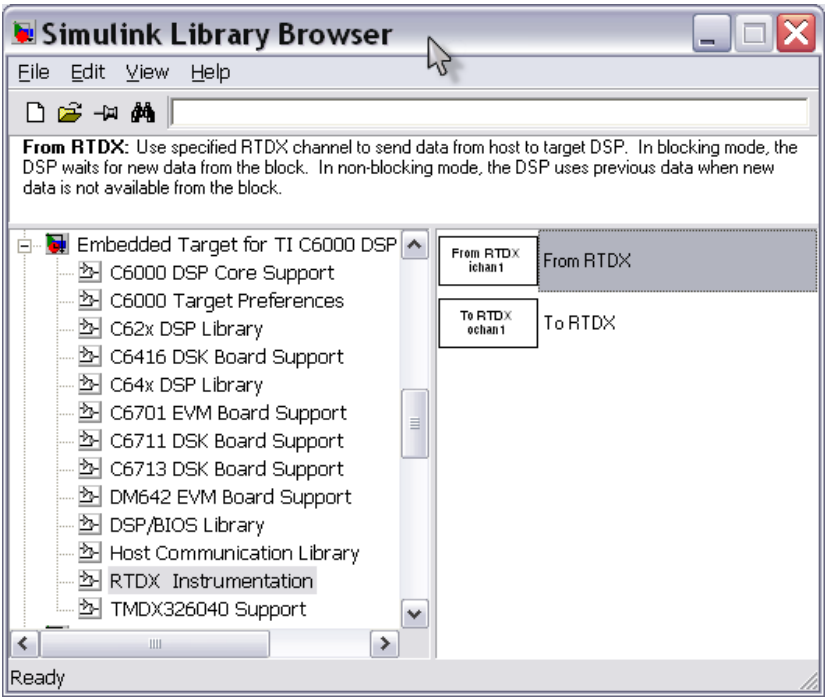
Имя	Содержание
Hardware Interrupt	Конфигурирует аппаратные прерывания.
Task	Задача
Triggered Task	Переключаемая задача

Библиотека Host Communication Library.



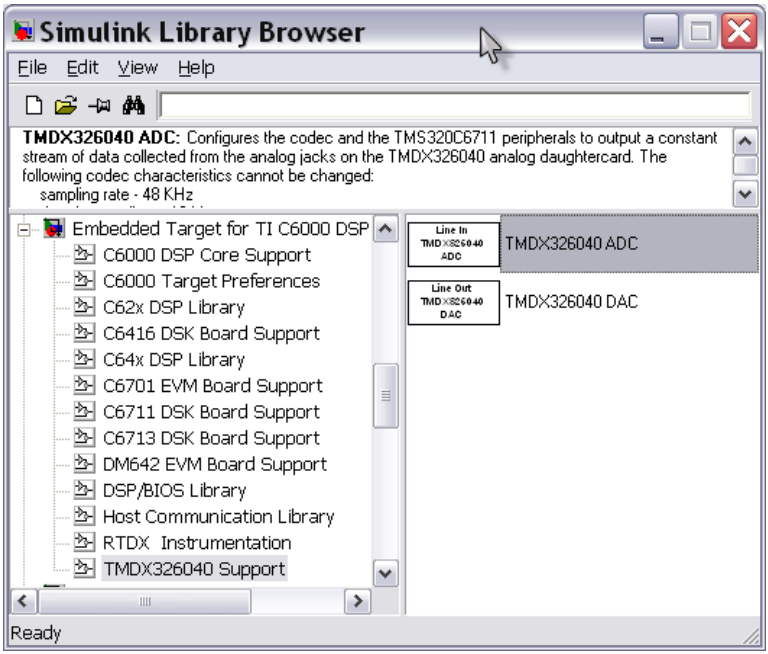
Имя	Содержание
Byte Reversal	Реверс битов
Pack	Упаковать
Receive	Принять
Send	Передать
Unpack	Распаковать

Библиотека RTDX Instrumentation.



Имя	Содержание
From RTDX	Конфигурирует канал RTDX для передачи данных от хоста к ЦПОС.
To RDDX	Конфигурирует канал RTDX для передачи данных от ЦПОС к хосту.

Библиотека TMDX326040 Support. Поддержка дочерней звуковой карты TMDX326040.



Имя	Содержание
TMDX326040 ADC	Конфигурирует АЦП встроенного кодека.
TMDX326040 DAC	Конфигурирует ЦАП встроенного кодека.