Московский государственный технический университет им. Н.Э. Баумана

Факультет «Информатика и системы управления»

Кафедра «Системы обработки информации и управления»



**Домашнее задание**

**по дисциплине “ЭУ в АСОИУ”**

**«Обзор и анализ систем команд микропроцессоров обработки сигналов»**

**ИСПОЛНИТЕЛЬ:**

Группа ИУ5-72

Мелконьянц А.Р.

"04" декабря 2021 г.

**ПРЕПОДАВАТЕЛЬ:**

Нестеров Ю. Г.

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

"\_\_"\_\_\_\_\_\_\_\_\_\_\_2021 г.

Москва 2021

Оглавление

[Введение 2](#_Toc89533123)

[1. Цифровая обработка сигналов 3](#_Toc89533124)

[2. Система команд сигнальных процессоров 9](#_Toc89533125)

[2.1. Программирование ЦСП 9](#_Toc89533126)

[2.2. Особенности ассемблеров ЦСП 10](#_Toc89533127)

[2.3. Совместимость внутри семейств ЦСП 11](#_Toc89533128)

[2.4. Отладка программ 11](#_Toc89533129)

[3. Особенности системы команд и программирования ЦСП 11](#_Toc89533130)

[4. Принцип работы 15](#_Toc89533131)

[4.1. Архитектура программного обеспечения 15](#_Toc89533132)

[4.2. Наборы команд 15](#_Toc89533133)

[4.3. Команды по данным 16](#_Toc89533134)

[4.4. Программы 16](#_Toc89533135)

[4.5. Аппаратная архитектура 17](#_Toc89533136)

[4.6. Архитектура памяти 17](#_Toc89533137)

[4.7. Адресация и виртуальная память 18](#_Toc89533138)

[5. Ассемблеры 18](#_Toc89533139)

[5.1. Программа Ассемблер 18](#_Toc89533140)

[5.2. Ассемблеры для DOS. 19](#_Toc89533141)

[5.3. Ассемблеры для Windows. 19](#_Toc89533142)

[5.4. Ассемблеры для GNU и GNU/Linux. 19](#_Toc89533143)

[5.5. Переносимые ассемблеры. 19](#_Toc89533144)

[5.6. Язык Ассемблер 20](#_Toc89533145)

[5.7. Достоинства: 20](#_Toc89533146)

[5.8. Недостатки: 21](#_Toc89533147)

[5.9. Основы 32-битного программирования в Windows 21](#_Toc89533148)

[5.10. Api функции 24](#_Toc89533149)

[5.11. Сообщения Windows 30](#_Toc89533150)

[5.12. Версии ассемблеров 32](#_Toc89533151)

[6. Вывод 36](#_Toc89533152)

[Список литературы 37](#_Toc89533153)

# Введение

Процессоры цифровой обработки сигналов представляют собой класс специализированных микропроцессоров, предназначенных для решения задач цифровой обработки сигналов (ЦОС), к которым относятся обработка звуковых сигналов, обработка изображений, распознавание речи, распознавание образов, цифровая фильтрация, спектральный анализ и др.

Часто в литературе такие микропроцессоры называются цифровыми сигнальными процессорами (ЦСП), илиDSP (Digital Signal Processors).

Первые процессоры этого класса появились в конце 1970-х годов. Требования практики, связанные с широким развитием мобильной беспроводной связи, стационарных систем широкополосной связи, использованием цифровой обработки сигналов в бытовой аудио- и видеотехнике и устройствах компьютерной периферии, с одной стороны и колоссальный прогресс электронной промышленности с другой привели к тому, что к настоящему времени производительность ЦСП возросла до сотен миллионов операций в секунду, а цена упала более чем на 90 % и даже для самых мощных процессоров составляет в настоящее время менее $20. Низкая потребляемая мощность (менее 1 Вт на максимальной частоте работы процессора) обеспечивает их широкое использование в различных встраиваемых устройствах от бытовой электроники до бортовых систем специального назначения.

# Цифровая обработка сигналов

Цифровая обработка сигналов — это арифметическая обработка последовательности значений амплитуд сигнала, получаемых через равные промежутки времени. Главное, что отличает эту информацию, - она необязательно заносится в память и поэтому может оказаться недоступной в будущем. Следовательно, обрабатывать ее нужно в реальном масштабе времени (РМВ). [1]

Особенности цифровой обработки сигналов рассмотрим на примере алгоритма нерекурсивной фильтрации. Структура устройства, реализующего данный алгоритм, показана на рисунке 1. [2]

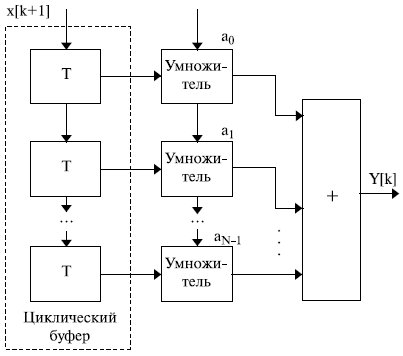


Рис. 1. Схема работы нерекурсивного фильтра

Обработка заключается в формировании выходного сигнала Y[k] по значениям N последних входных отсчетов x[k], которые поступают на вход устройства через определенный интервал времени Т. Принятые отсчеты сохраняются в ячейках циклического буфера. При приеме очередного отсчета содержимое всех ячеек буфера переписывается в соседнюю позицию, самый старый отсчет покидает буфер, а новый записывается в его младшую ячейку.

Аналитически алгоритм работы нерекурсивного фильтра записывается как:

где ai - коэффициенты, определяемые типом фильтра.

Отсчеты с выходов элементов буфера поступают на умножители, на вторые входы которых поступают коэффициенты ai. Результаты произведений складываются и формируют отсчет выходного сигнала Y[k], после чего содержимое буфера сдвигается на 1 позицию и цикл работы фильтра повторяется. Выходной сигнал Y[k] должен быть вычислен до поступления следующего входного сигнала, то есть за интервал T. В этом заключается суть работы устройства в реальном масштабе времени. Интервал времени T задается частотой дискретизации, которая определяется областью применения фильтра. По следствию из теоремы Котельникова в дискретном сигнале период, соответствующий наивысшей представимой частоте, соответствует двум периодам дискретизации. При обработке звукового сигнала частоту дискретизации можно принять в 40 кГц. В этом случае если необходимо реализовать цифровой нерекурсивный фильтр 50-го порядка, то за время в 1/40 кГц = 25 мкс должно быть выполнено 50 умножений и 50 накоплений результатов умножения. Для обработки видеосигнала интервал времени, за который должны быть выполнены эти действия, будет на несколько порядков меньше.

Если выполнять ДПФ входной последовательности напрямую, строго по исходной формуле, то потребуется много времени. Посчитав по определению (N раз суммировать N слагаемых), получаем величину порядка N2.

Тем не менее, можно обойтись существенно меньшим числом операций.

Наиболее популярным из алгоритмов ускоренного вычисления ДПФ является метод Кули-Тьюки (Cooley-Tukey), позволяющий вычислить ДПФ для числа отсчетов N = 2k за время порядка N\*log2 N (отсюда и название - **быстрое преобразование Фурье**, БПФ, или в английском варианте FFT - Fast Fourier TRansformation ). Основная идея этого метода заключается в рекурсивном разбиении массива чисел на два подмассива и сведении вычисления ДПФ от целого массива к вычислению ДПФ от подмассивов в отдельности. При этом процесс разбиения исходного массива на подмассивы проводится по методу побитовой обратной сортировки ( bit-reversal sortINg ).

Сначала входной массив делится на две подмассива - с четными и нечетными номерами. Каждый из подмассивов перенумеровывается и снова делится на два подмассива - с четными и нечетными номерами. Эта сортировка продолжается до тех пор, пока размер каждого подмассива не достигнет 2 элементов. В результате (что можно показать математически) номер каждого исходного элемента в двоичной системе переворачивается. То есть, например, для однобайтных номеров двоичный номер 00000011 станет номером 110000000, номер 01010101 - номером 10101010.

Существуют алгоритмы БПФ для случаев, когда N является степенью произвольного простого числа (а не только двойки), а также в случае, когда число N является произведением степеней простых чисел любого числа отсчетов. Однако БПФ, реализованное по методу Кули-Тьюки для случая N = 2k, получило наиболее широкое распространение. Причина этого в том, что алгоритм, построенный по этому методу, обладает рядом очень хороших технологических свойств:

* структура алгоритма и его базовые операции не зависят от числа отсчетов (меняется только число прогонов базовой операции);
* алгоритм легко распараллеливается с использованием базовой операции и конвейеризуется, а также легко каскадируется (коэффициенты БПФ для 2N отсчетов могут быть получены преобразованием коэффициентов двух БПФ по N отсчетов, полученных "прореживанием" исходных 2N отсчетов через один);
* алгоритм прост и компактен, допускает обработку данных "на месте" и не требует дополнительной оперативной памяти.

Однокристальные микроконтроллеры и даже универсальные микропроцессоры оказываются относительно медленными при выполнении операций, характерных для ЦОС. К тому же требования к качеству преобразования аналоговых сигналов постоянно возрастают. В **сигнальных микропроцессорах** такие операции поддерживаются на аппаратном уровне и выполняются, соответственно, достаточно быстро. Работа в реальном масштабе времени требует от процессора также поддержки на аппаратном уровне таких действий, как обработка прерываний, программных циклов.

Все это приводит к тому, что DS P-процессоры, архитектурно включая в себя многие черты как универсальных микропроцессоров, особенно с **RISC-архитектурой**, так и **однокристальных микроконтроллеров**, в то же время значительно отличаются от них. Универсальный микропроцессор помимо чисто вычислительных операций выполняет функцию объединяющего звена всей микропроцессорной системы, в частности компьютера.

Он должен управлять работой различных компонентов аппаратного обеспечения, таких как дисководы, графические дисплеи, сетевой интерфейс, с тем чтобы обеспечить их согласованную работу. Это приводит к достаточно сложной архитектуре, поскольку она должна поддерживать наряду с целочисленной арифметикой и операциями с плавающей запятой такие базовые функции, как защита памяти, мультипрограммирование, обработка векторной графики и т. п. В итоге типичный универсальный микропроцессор с CISC-, а зачастую и RISC-архитектурой имеет систему из несколько сотен команд, которые обеспечивают выполнение всех этих функций, и соответствующую аппаратную поддержку. Это ведет к необходимости иметь в составе такого МП десятки миллионов транзисторов.

В то же время DSP-процессор является узкоспециализированным устройством. Его единственная задача - быстро обрабатывать поток цифровых сигналов. Он состоит главным образом из высокоскоростных аппаратных схем, выполняющих арифметические функции и манипулирующих битами, оптимизированных таким образом, чтобы быстро обрабатывать большие объемы данных. В силу этого набор команд у DSP куда меньше, чем у универсального микропроцессора: их число обычно не превышает 80. Это значит, что для DSP требуется облегченный декодер команд и гораздо меньшее число исполнительных устройств. Кроме того, все исполнительные устройства в итоге должны поддерживать высокопроизводительные арифметические операции. Таким образом, типичный DSP-процессор состоит не более чем из нескольких сот тысяч (а не десятков миллионов, как в современных CISC-МП) транзисторов. В силу этого такие МП потребляют меньше энергии, что позволяет использовать их в продуктах, работающих от батарей. Крайне упрощается их производство, поэтому они находят себе применение в недорогих устройствах. Сочетание низкого энергопотребления и невысокой стоимости позволяет использовать их не только в высокой сфере телекоммуникаций, но и в сотовых телефонах и роботах-игрушках.

Отметим основные особенности архитектуры процессоров цифровой обработки сигналов:

1. Гарвардская архитектура, основу которой составляет физическое и логическое разделение памяти команд и памяти данных. Ключевые команды DSP-процессора являются многооперандными, и ускорение их работы требует одновременного чтения нескольких ячеек памяти. Соответственно на кристалле имеются раздельные шины адреса и данных (в некоторых типах процессоров - несколько шин адреса и данных). Это позволяет совмещать во времени выборку операндов и исполнение команд. Использование модифицированной гарвардской архитектуры предполагает, что операнды могут храниться не только в памяти данных, но и в памяти команд вместе с программами. Например, в случае реализации цифровых фильтров коэффициенты могут храниться в памяти программ, а значения данных - в памяти данных. Поэтому коэффициент и данные могут выбираться в одном машинном цикле. Для обеспечения выборки команды в том же машинном цикле используется либо кэш-память программ, либо двукратное обращение к памяти программ за время машинного цикла.
2. Для сокращения времени выполнения одной из основных операций цифровой обработки сигнала - умножения - применяется аппаратный умножитель. В процессорах общего назначения эта операция реализуется за несколько тактов сдвига и сложения и занимает много времени, а в **DSP-процессорах** благодаря специализированному умножителю нужен всего один цикл. Встроенная схема аппаратного умножения позволяет выполнить за 1 такт основную операцию ЦОС - **умножение с накоплением** ( MultIPly-Accumulate - MAC ) для 16- и/или 32-разрядных операндов.
3. Аппаратная поддержка циклических буферов. Например, для фильтра, представленного на [рис. 16.3](https://intuit.ru/studies/courses/604/460/lecture/10351?page=2#image.16.3), при каждом вычислении отсчета выходного сигнала используется новый отсчет входного сигнала, который сохраняется в памяти на месте самого старого. Для такого циркулирующего буфера может использоваться фиксированная область ОЗУ. При этом во время вычислений генерируются лишь последовательные значения адресов ОЗУ вне зависимости от того, какая операция - запись или чтение - выполняется в настоящий момент. Аппаратная реализация циклических буферов позволяет установить параметры буфера (адрес начала, длина) в программе вне тела цикла фильтрации, что позволяет сократить время выполнения циклического участка программы.
4. Сокращение длительности командного такта. Это во многом обеспечивается приемами, характерными для RISC-процессоров. Главными из них являются размещение операндов большинства команд в регистрах, а также конвейеризация на уровне команд и микрокоманд. Конвейер имеет от 2 до 10 ступеней, что позволяет на различных стадиях выполнения одновременно обрабатывать до 10 команд. При этом используется генерация адресов регистров параллельно с выполнением арифметических операций, а также многопортовый доступ к памяти. Сюда же можно отнести и такой прием, характерный для универсальных микропроцессоров с **EPIC-архитектурой**, как применение команд со сверхбольшой длиной слова (VLIW), генерируемых на стадии компиляции программы. Этому же служит и рассмотренная выше **гарвардская архитектура** процессора, характерная для однокристальных микроконтроллеров.
5. Наличие на кристалле процессора внутренней памяти, что роднит ЦСП с однокристальными МК. Встроенная в процессор память обычно имеет значительно большее быстродействие, чем внешняя. Наличие встроенной памяти позволяет значительно упростить систему в целом, уменьшить ее размеры, энергопотребление и стоимость. Емкость внутренней памяти является результатом определенного компромисса. Ее увеличение ведет к удорожанию процессора и увеличивает энергопотребление, а ограниченная емкость памяти программ не позволяет хранить сложные алгоритмы. Большинство DS P-процессоров с фиксированной точкой имеют малую емкость внутренней памяти, обычно от 4 до 256 Кбайт, и невысокую разрядность внешних шин данных, связывающих процессор с внешней памятью. В то же время ЦСП с плавающей точкой обычно предполагают работу с большими массивами данных и сложными алгоритмами и имеют либо встроенную память большой емкости, либо большую разрядность адресных шин для подключения внешней памяти (а иногда и то, и другое).
6. Широкие возможности по аппаратному взаимодействию с внешними устройствами, включающие:

* большое разнообразие интерфейсов, в том числе контроллеры локальной промышленной сети CAN, встроенные коммуникационные (SCI) и периферийные (SPI) интерфейсы, I2C, UART;
* несколько входов для аналоговых сигналов и, соответственно, встроенный АЦП;
* выходные каналы широтно-импульсной модуляции (ШИМ);
* развитую систему внешних прерываний;
* контроллеры прямого доступа в память.

1. В некоторых DSP-семействах предусмотрены специальные аппаратные средства, облегчающие создание мультипроцессорных систем с параллельной обработкой данных для наращивания производительности.
2. DSP-процессоры широко используются в мобильных устройствах, где потребляемая мощность является основной характеристикой. Для снижения энергопотребления в сигнальных процессорах применяется множество методов, в том числе уменьшение напряжения питания и введение функций управления потреблением, например, динамическое изменение тактовой частоты, переключение в спящий или дежурный режим либо отключение не используемой в данный момент периферии. Следует отметить, что эти меры оказывают значительное воздействие на скорость работы процессора и при некорректном использовании могут привести к неработоспособности проектируемого устройства (в качестве примера можно упомянуть некоторые сотовые телефоны, которые в результате ошибок в программах управления энергопотреблением иногда переставали включаться) или к ухудшению его эксплуатационных характеристик (например, значительному времени восстановления работоспособности при выходе из спящего режима).

# Система команд сигнальных процессоров

## Программирование ЦСП

Для программирования ЦСП обычно используют один из двух языков — ассемблер и C. Основные особенности ассемблеров ЦСП совпадают с аналогичными языками обычных микропроцессоров и, в целом, могут быть описаны как:

* Язык ассемблера является машинно-ориентированным, то есть каждое семейство процессоров имеет язык, отличающийся от языка других семейств;
* Одна команда на ассемблере обычно эквивалентна одной команде машинного языка;
* При программировании на ассемблере программисту доступны все ресурсы процессора и системы, что позволяет использовать их максимально эффективно;
* От программиста требуется хорошее знание архитектуры каждого конкретного процессора, с которым он работает, то есть требуемая квалификация персонала должна быть достаточно высокой;
* Создание и отладка программ на ассемблере — длительный трудоёмкий процесс, также требующий высокой квалификации.

С другой стороны, при использовании языков среднего и высокого уровня, в частности, C, можно заметно упростить и ускорить создание программ, но при этом ресурсы системы будут использоваться менее эффективно, по сравнению с программой, целиком написанной на ассемблере.

В реальности обычно используется подход, совмещающий достоинства как языков высокого уровня, так и эффективности программ на ассемблере. Выражается это в том, что стандартные библиотеки обычно создаются на ассемблере, как и критичные ко времени исполнения и объёму памяти части кода. В то же время вспомогательные модули могут создаваться на языке высокого уровня, ускоряя и упрощая разработку программной системы в целом.

## Особенности ассемблеров ЦСП

К интересным особенностям ассемблеров ЦСП можно отнести следующее:

* Наличие двух форм записи многих команд — мнемонической и алгебраической. Мнемоническая форма аналогична записи команд для обычных микропроцессоров, например, ADD dst, src. Другая, алгебраическая, в ассемблерах стандартных микропроцессоров используется реже, в то время как на языке ЦСП упомянутая команда может быть записана в виде dst = dst + src. Обычно ассемблеры ЦСП понимают обе формы записи, но, например, ассемблеры Analog Devices и Lucent Technologies используют только алгебраическую запись.
* Средства организации стандартных структур, например, специальных аппаратных команд повторения одной команды или блока кода. При этом, в отличие от команд повторения обычных процессоров, ЦСП может пропускать цикл выборки кода повторяемой команды, что уменьшает время выполнения каждого повторения как минимум на 1 цикл шины, что при двухцикловой команде даёт двойной выигрыш по времени.

## Совместимость внутри семейств ЦСП

Обычно ЦСП выпускаются семействами, и изделия внутри семейств имеют аналогичные языки ассемблера, или даже совместимы на уровне машинных кодов. Также внутри семейства обычно используются одинаковые наборы библиотек подпрограмм. Как и в обычных микропроцессорах, зачастую старшие модели ЦСП могут исполнять машинный код младших моделей, либо их ассемблер включает все команды младших моделей как подмножество собственного набора команд.

## Отладка программ

Обычно отладка программ, написанных для ЦСП, производится с помощью специальных средств, включающих программные имитаторы и эмуляторы. В их состав также часто входят средства профилирования (измерения скорости выполнения блоков кода). [3]

# Особенности системы команд и программирования ЦСП

Система команд сигнальных процессоров имеет многие черты систем команд универсальных микропроцессоров (особенно с RISC-архитектурой) и однокристальных микроконтроллеров. Она включает в себя основные арифметические и логические операции и команды переходов, но в меньшем, чем в универсальных МП, количестве. Число режимов адресации операндов также относительно невелико. Команда имеет простой, четко заданный формат. Длина команды составляет одно, реже два 16-разрядных слова. Однако наряду с использованием сокращенного набора команд, в DSP-процессорах применяются и такие характерные для MMX-обработки аппаратно поддерживаемые инструкции, как команды поиска минимума и максимума, получения абсолютного значения, сложения с насыщением, при котором в случае переполнения при сложении двух чисел результату присваивается максимально возможное в данной разрядной сетке значение. Это ведет к уменьшению количества конфликтов в конвейере и повышает эффективность работы процессора. [4]

С другой стороны, ЦСП содержат ряд команд, наличие которых обусловлено спецификой их применения и которые вследствие этого редко присутствуют в микропроцессорах других типов. Прежде всего это, конечно, команда умножения с накоплением суммы MAC, лежащая в основе ЦОС: А = В\*С+А. В системах команд некоторых сигнальных процессоров можно при программировании указать число выполнений этой команды в цикле и правила изменения индексов для адресации операндов В и С. При этом в отличие от команд повторения обычных процессоров сигнальный процессор может аппаратно поддерживать проверку условия завершения цикла. Сюда же можно отнести и команды сдвига (перезаписи) в соседнюю ячейку ОЗУ данных, поддерживающие работу циклического буфера для подготовки умножения в следующем такте.

Для эффективной реализации алгоритмов БПФ в систему команд некоторых DSP-процессоров включены возможности адресации с реверсированием бит адреса.

Программирование микропроцессоров этого класса также имеет свои особенности. Значительное удобство для разработчика, обычно связываемое с использованием языков высокого уровня, в большинстве случаев оборачивается получением менее компактного и быстрого кода. Так как особенности ЦОС предполагают работу в реальном времени, это приводит к необходимости использования для решения тех же задач более мощных и дорогих DSP. Такая ситуация особенно критична для крупносерийной продукции, где разница в стоимости более производительного DSP или дополнительного процессора играет важную роль. В то же время в современных условиях скорость разработки (и, следовательно, выхода нового изделия на рынок) может принести больше выгод, чем затраты времени на оптимизацию кода при написании программы на ассемблере.

Компромиссным подходом здесь служит использование ассемблера для написания наиболее критичных с точки зрения время- и ресурсоемкости участков программы, в то время как основная часть программы пишется на языке высокого уровня, как правило, Си или Си++.

Предположим, что программа на Си пишется в 5 раз быстрее, чем на ассемблере, а получаемый в результате трансляции код работает примерно во столько же раз медленнее. Тогда в случае, когда некоторый участок программы объемом 20 % кода занимает 80 % времени выполнения программы, что бывает достаточно часто ("правило 80/20"), получим, что программирование этого фрагмента на ассемблере приведет к увеличению времени программирования на 80 %, но уменьшит время работы программы почти втрое (рисунок 2).

Изображение выглядит как стол

Автоматически созданное описание

Рис. 2. Анализ эффективности различных методов программирования

Вот почему программы для процессоров, имеющих критичное значение времени работы, хотя в большинстве случаев и написаны на Си, но пестрят ассемблерными вставками.

К интересным особенностям DSP-ассемблеров можно отнести наличие двух форм записи многих команд - мнемонической и алгебраической. Мнемоническая форма аналогична записи команд для обычных микропроцессоров, например, ADD dst, src. Другая, алгебраическая форма в ассемблерах микропроцессоров других классов используется реже, в то время как на DSP-ассемблере упомянутая команда может быть записана в виде DSt= DSt+ src. Обычно ассемблеры DSP понимают обе формы записи, но, например, ассемблеры ADI иLucent применяют только алгебраическую запись.

Спектр компаний, представленных на рынке DSP-процессоров, более широк, чем среди производителей универсальных микропроцессоров.

В настоящее время доминирующее положение на рынке ЦСП занимает фирма Texas Instruments. Среди остальных производителей этой продукции следует выделить фирмы Freescale Semiconductor (ранее подразделение Motorola), Analog Devices, Phillips Semiconductors и Agere Systems (Lucent MicroelecTRonic) (рисунок 3). Следует отметить, что данный рынок является очень подвижным, что приводит к его постоянному перераспределению между компаниями и появлению на рынке новых игроков.

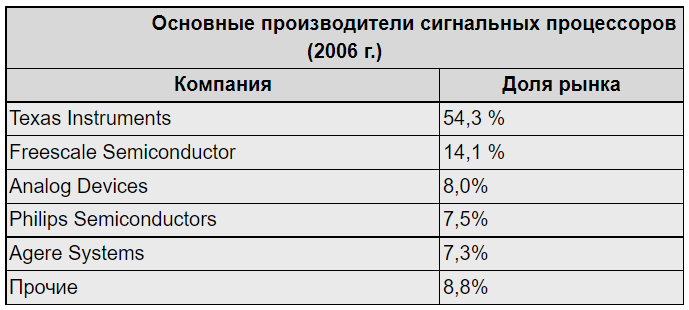


Рис. 3. Основные производители сигнальных процессоров (2006г)

С ростом числа областей применения ЦОС и сложности алгоритмов обработки возрастают требования к сигнальным процессорам в плане повышения быстродействия и оснащенности интерфейсными и другими специализированными узлами. В настоящее время на рынке присутствует множество типов ЦСП, как универсальных, так и ориентированных на достаточно узкий круг задач. Естественно, ни один из процессоров не может быть оптимальным для всех приложений. Поэтому первая задача разработчика - выбор процессора, наиболее подходящего по производительности, цене, наличию определенной периферии, потребляемой мощности, простоте использования и другим критериям.

Например, для таких портативных устройств, как мобильные телефоны, портативные цифровые плееры, первостепенными являются стоимость, степень интеграции и потребляемая мощность, а максимальная производительность зачастую не нужна, так как обычно влечет за собой значительное повышение потребляемой мощности, не давая преимуществ при обработке относительно низкоскоростных аудиоданных. В то же время для гидроакустических или радиолокационных систем определяющими параметрами являются скорость работы, наличие высокоскоростных интерфейсов, а стоимость является второстепенным критерием.

Хотя большинство фирм выпускает широкую номенклатуру процессоров, которые могут быть использованы для различных применений, среди них наблюдается определенная специализация. Так, следует отметить предпочтительность процессоров Analog Devices для приложений, требующих выполнения больших объемов математических вычислений (таких как цифровая фильтрация сигнала, вычисление корреляционных функций и т. п.), поскольку их производительность на подобных задачах выше, чем у процессоров компаний Freescale и Texas INsTRuments. В то же время для задач, требующих выполнения интенсивного обмена с внешними устройствами (многопроцессорные системы, различного рода промышленные контроллеры), предпочтительнее использовать процессоры Texas INsTRuments, обладающие высокоскоростными интерфейсными подсистемами. Компания Freescale является лидером по объему производства относительно дешевых и достаточно производительных 16- и 24-разрядных сигнальных процессоров с фиксированной точкой.

# Принцип работы

## Архитектура программного обеспечения

По стандартам процессоров общего назначения наборы команд ЦСП часто очень нерегулярны; в то время как традиционные наборы команд состоят из более общих команд, которые позволяют им выполнять более широкий спектр операций. Наборы команд, оптимизированные для цифровой обработки сигналов, содержат команды для общих математических операций, которые часто происходят в вычислениях ЦСП. Как традиционные, так и оптимизированные для ЦСП наборы команд могут вычислять любую произвольную операцию, но операция, для выполнения которой может потребоваться несколько команд ARM или x86, может потребовать только одну команду в оптимизированном для DSP наборе команд.

Одним из следствий для архитектуры программного обеспечения является то, что оптимизированные вручную процедуры ассемблерного кода обычно упаковываются в библиотеки для повторного использования, вместо того чтобы полагаться на передовые технологии компилятора для обработки основных алгоритмов. Даже при современных компиляторах оптимизированный вручную ассемблерный код является более эффективным, и многие общие алгоритмы, участвующие в вычислениях ЦСП, написаны вручную, чтобы в полной мере использовать преимущества архитектурных оптимизаций.[5]

## Наборы команд

1. Операции по совмещенному умножению-сложению.

* широко используется во всех видах матричных операций:
* свертка функций для фильтрации;
* скалярное произведение;
* полиномиальная оценка.
* фундаментальные алгоритмы ЦСП сильно зависят от производительности многократного накопления
* КИХ-фильтров;
* Быстрое преобразование Фурье (БПФ).

1. Команды по увеличению параллельности:

* SIMD (одиночный поток команд, множественный поток данных, ОКМД);
* VLIW («очень длинная машинная команда»);
* суперскалярность.

1. Специализированные команды для сравнения по модулю в кольцевых буферах и режиме адресации с обратным битом для перекрестных ссылок FFT.
2. DSP иногда используют нестационарное кодирование для упрощения аппаратного обеспечения и повышения эффективности кодирования.
3. Несколько арифметических блоков могут потребовать архитектуры памяти для поддержки нескольких обращений за цикл команд.
4. Специальные элементы управления циклом, такие как архитектурная поддержка для выполнения нескольких командных слов в очень плотном цикле без накладных расходов для выборки команд или тестирования выхода.

## Команды по данным

* Насыщенная арифметика, при этом операции, которые производят перерасход, будут накапливаться с максимальным (или минимальным) значением, которое регистр может хранить, а не обходить (максимум+1 как во многих процессорах общего назначения, а не остается на максимальном уровне). Иногда доступны различные режимы работы sticky-бит.
* Арифметика с фиксированной точкой часто используется для ускорения арифметической обработки.
* Одноцикловые операции для увеличения преимуществ конвейерной обработки

## Программы

* Блок с плавающей запятой интегрирован непосредственно в операционный автомат;
* Конвейерная архитектура
* Высоко параллельные операции по совмещенному умножению-сложению (блоки MAC);
* Аппаратное управление циклом, чтобы уменьшить или устранить усилия, необходимые для операций цикла.

## Аппаратная архитектура

В инжиниринге аппаратная архитектура означает набор физических компонентов системы и их взаимосвязь. Это описание называют моделью проектирования оборудования. Оно позволяет разработчикам оборудования понять как их компоненты вписываются в системную архитектуру и предоставляет разработчикам компонентов программного обеспечения важную информацию, необходимую для разработки и интеграции программного обеспечения. Четкое определение аппаратной архитектуры позволяет различным традиционным инженерным дисциплинам (например, электротехнике и машиностроению) более эффективно работать вместе для разработки и производства новых машин, устройств и компонентов.

Аппаратное обеспечение также является выражением, используемым в компьютерной индустрии, чтобы отличить аппаратное обеспечение от программного обеспечения, которое работает на нем. Но аппаратное обеспечение, в рамках дисциплин автоматизации и программной инженерии, не обязательно должно быть просто компьютером. Современный автомобиль использует гораздо больше программного обеспечения, чем космический корабль «Аполлон». Кроме того, современный самолет не может функционировать без выполнения десятков миллионов компьютерных команд, встроенных и распределенных по всему самолету и расположенных как в стандартном компьютерном оборудовании, так и в специализированных аппаратных компонентах, таких как проводные логические элементы IC, аналоговые и гибридные устройства и другие цифровые компоненты. Необходимость эффективного моделирования того как отдельные физические компоненты объединяются в сложные системы важна для широкого спектра применений, включая компьютеры, персональные цифровые помощники (КПК), сотовые телефоны, хирургические приборы, спутники и подводные лодки.

## Архитектура памяти

ЦСП обычно оптимизированы для потоковой передачи данных и используют специальные архитектуры памяти, которые могут извлекать несколько данных или команд одновременно, такие как архитектура Гарварда или модифицированная архитектура фон Неймана, которые используют отдельные программы и память данных (иногда даже одновременный доступ на нескольких шинах данных).

ЦСП иногда могут полагаться на поддержку кода, чтобы узнать об иерархиях кэша и связанных с ними задержках. Это компромисс, который позволяет повысить производительность. Широко используется прямой доступ к памяти (DMA).

## Адресация и виртуальная память

ЦСП часто используют многозадачные операционные системы, но не поддерживают виртуальную память или защиту памяти. Операционные системы, использующие виртуальную память, требуют больше времени для переключения между процессами, что увеличивает задержку.

* Аппаратная адресация по модулю позволяет реализовать кольцевой буферы, не тестируя на оборот.
* Бит-обратная адресация, специальный режим адресации памяти полезна для вычисления БПФ.
* Исключение блока управления памятью;
* Устройство для формирования адреса.

# Ассемблеры

## Программа Ассемблер

Ассемблер (assembler— сборщик) — компьютерная программа, компилятор исходного текста программы, написанной на языке ассемблера, в программу на машинном языке. [6]

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 объектныймодуль, 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), использующийAT&T-синтаксис, в отличие от большинства других популярных ассемблеров, которые используютIntel-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 .

## Язык Ассемблер

Язык Ассемблер— 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 ихаргументов. При этом одной команде языка ассемблера может соответствовать несколько вариантов команд процессора, в зависимости от операндов.

Кроме того, язык ассемблера позволяет использовать символические меткивместо адресов ячеек памяти, которые при ассемблировании заменяются на автоматически рассчитываемые абсолютные или относительные адреса, а также так называемыедирективы(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 макроопределенияс параметрами.

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

## 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 (рисунок 4) уходит в прошлое. [7]

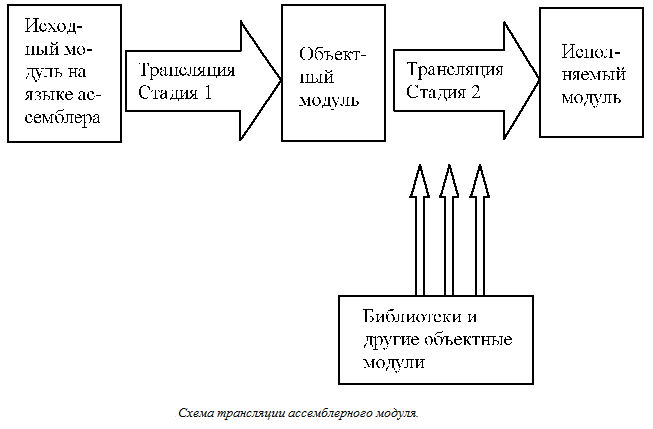


Рис. 4. Схема трансляции ассемблерного модуля

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. [8]

* Редакторы.

Начнём 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 — это EAS.EXE (Easy Assembler Shell). Редактор, а точнее оболочка, позволяет создавать и транслировать довольно сложные проекты, состоящие из ASM-,OВJ-,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.

* Нех-редакторы.

Нех-редакторы позволяют просматривать и редактировать загружаемые модули в шестнадцатеричном виде. Их великое множество, к тому же отладчики и дизассемблеры, как правило, имеют встроенные НЕХ-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 шаблоны, 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.

## Api функции

Ниже (таблица 1)перечислены основные функции API, применяемые в ассемблере.

Таблица 1. Основные функции API

| 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. |

## Сообщения Windows

Ниже (таблица 2)перечислены основные сообщения Windows, применяемые в ассемблере.

Таблица 2. Сообщения Windows

| 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 свойствоLBS\_WANTKEYBOARDINPUT. |

## Версии ассемблеров

* 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. [9]

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.

* 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 сFASMне распространяется.

Компиляция программы в FASMсостоит 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-синтаксисзаписи 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 простоlabel\_nameозначает адрес (то есть порядковый номер) ячейки. Это позволяет обходиться без ключевого словаoffset. Также в FASM при переопределении размера операнда вместоbyte ptrпишется простоbyte, вместоword ptr—wordи 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 FASMподдерживают многие специализированные ИСР, такие как RadASM, WinAsm Studio, Fresh (IDE) (специально спроектированный под FASM) и т. д.

* NASM (Netwide Assembler)

NASM —свободный(LGPLилицензия BSD)ассемблердля архитектурыIntelx86. 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.

* 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.

* 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 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.

# Вывод

Цифровая обработка сигналов, еще несколько десятилетий назад представлявшая собой лишь экзотическую ветвь науки, в настоящее время является одним из магистральных направлений развития практической радиоэлектроники. Благодаря успехам микроэлектроники, положившей начало бурному прогрессу цифровых технологий, системы ЦОС не только стали реальностью, но и прочно вошли в нашу повседневную жизнь в виде цифровой мобильной связи, цифровых аудио-видео систем, цифрового телевидения и многого другого.

Соответственно, существенно изменились и требования, предъявляемые к подготовке инженеров-электронщиков, программистов и радиоинженеров. Умение работать с ЦОС является сейчас одним из важнейших условий успешной профессиональной деятельности специалистов вышеназванного профиля.

# Список литературы

1. https://studfile.net/preview/1047153/page:9/
2. https://intuit.ru/studies/courses/604/460/lecture/10351?page=2
3. https://ru.wikipedia.org/wiki/%D0%A6%D0%B8%D1%84%D1%80%D0%BE%D0%B2%D0%BE%D0%B9\_%D1%81%D0%B8%D0%B3%D0%BD%D0%B0%D0%BB%D1%8C%D0%BD%D1%8B%D0%B9\_%D0%BF%D1%80%D0%BE%D1%86%D0%B5%D1%81%D1%81%D0%BE%D1%80#%D0%9F%D1%80%D0%BE%D0%B3%D1%80%D0%B0%D0%BC%D0%BC%D0%B8%D1%80%D0%BE%D0%B2%D0%B0%D0%BD%D0%B8%D0%B5\_%D0%A6%D0%A1%D0%9F
4. https://studfile.net/preview/1047152/page:12/
5. http://www.npcas.ru/wiki/tsifrovoy-signalnyy-protsessor-dsp.html
6. https://studfile.net/preview/4599904/page:7/
7. https://infopedia.su/25x3eb0.html
8. https://pro-prof.com/forums/topic/simpleasm-assembler-ide
9. https://www.tadviser.ru/index.php/%D0%A1%D1%82%D0%B0%D1%82%D1%8C%D1%8F:%D0%90%D1%81%D1%81%D0%B5%D0%BC%D0%B1%D0%BB%D0%B5%D1%80